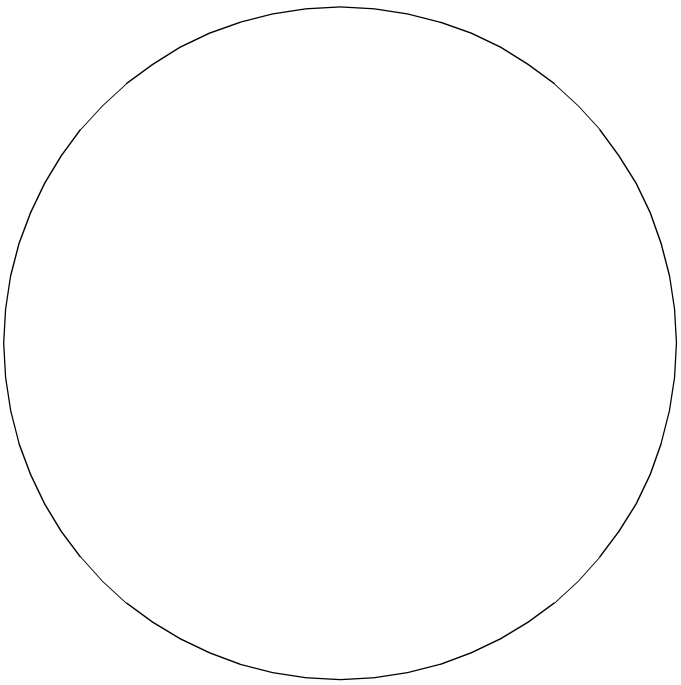
Practitioners' verification of SDL systems

Øystein Haugen

Thesis for the Dr. Scient. degree

University of Oslo
Faculty of Mathematics and Natural Sciences,
Department of Informatics



Practitioners' Verification of SDL systems

by

Øystein Haugen

Doctoral Dissertation

Submitted to
the Faculty of Mathematics and Natural Sciences
at the University of Oslo
for the fulfillment of the degree of Dr. Scient. in Informatics

April 1997

Supervisor: Professor Dag Belsnes

Table of Contents

1 Introduction

T.T.T.

Put up in a place
where it's easy to see
the cryptic admonishment
T.T.T.

When you feel how depressingly
slowly you climb,
it's well to remember that
Things Take Time

1. Introduction

Here we give summaries of the thesis and background for the study. We summarize the thesis in an abstract, an executive summary and a technical summary. Then we give a short introduction to the language SDL and its use. We elaborate on our motivation for the work and give some background in other approaches. Finally the introduction ends up with a guide to the reading of this thesis and acknowledgments.

1.1 Abstract

This thesis aims to give a skeleton of a bridge between theoretical validation of communicating finite state machines, and the practical quality improvement of designing real systems in SDL.

Mn-procedure The backbone of the bridge is the Mn-procedure which reduces an SDL-system to an SDL-process by eliminating internal communication. The Mn-procedure tries to establish that a progressive SDL system is confluent, and then a simple reduction algorithm produces the reduction.

We show that the Mn-approach for reduction is fruitful on well known examples: Alternating Bit Protocol, The Brock-Ackerman anomaly and the RPC-Memory specification problem.

Monolithic The Mn-approach is purely monolithic as it produces SDL descriptions from SDL descriptions. We argue that the reductions can result in:

1. A more compact functional specification; (This may help when other techniques are used to analyze the total system.)

2. Improved overview and understanding; (This helps for the designers and reviewers of the system, and it helps achieving reuse.)
3. Simpler analysis on higher levels. (The Mn-procedure is compositional.)

We show how the Mn-approach can be applied to proving refinement.

SDL extensions

In order to cope with (extreme) fairness and non-determinism we introduce a few new features to SDL, fair non-deterministic decision and spontaneous save. We also introduce concepts to harmonize signals and variables to be able to express more general signal handling.

Conditional reduction

We define modifications on the Mn-procedure which result in conditional reduction. Conditional reduction appears when the confluence is dependent upon the system executing without run-time errors, which are caught by a monitoring layer. This means that either the system executes as the reduction, or it turns into a run-time error.

We also note that conditional reduction based upon limited resources such as finite limits on the lengths of the channel queues may be very attractive. This makes confluence decidable, and a simplified version of the Mn-procedure suffices.

Mn-metric

Based on the Mn-procedure we develop a metric to indicate complexity areas of the system, and thereby a strategy to decrease the complexity and improve the system quality.

Confluent design

We argue that the Mn-procedure scales well and will work also on selected components of real systems. Therefore we end up specifying a method which we call “confluent design” based on aiming for confluence in the design of every constituent part of a system.

1.2 Executive Summary

In this short section we shall give a brief summary of the background and aims of this dissertation, and the results which we claim to reach.

The background for our work is the discrepancy between the sophistication of the available verification methods and the perceived complexity of the systems which are being analyzed. We feel that the intricacy of the analysis should not exceed the mental capacities of the designer. Still we recognize the powers of computers to perform repeated tasks and want to exploit the possibilities of automating trivial verification steps.

1.2.1 Major aims

We provide an approach (the Mn-approach) to supplement systems engineering. Its core part is a technique (the Mn-procedure) for reduction of SDL-like systems. This technique we show can be used both for theoretical purposes and on real systems. The Mn-approach also provides a methodology which reaps the benefits of the Mn-procedure for real systems.

The name “Mn”[1] simply means “Machine n” referring to the use of several machines (automata) or generations in the Mn-procedure.

1. “Mn” is actually very mnemonic since it is the two first letters of “mnemonic”.

1.2.2 SDL systems

We have limited ourselves to SDL-like systems, but our approach is generally applicable to the class of systems defined by asynchronously communicating finite state machines.

An SDL system is a system which has been specified by the specification language SDL [25; 78; 83]. SDL systems consist of interacting components which in the end are processes which are described as finite state machines extended with data variables. The communication is asynchronous and we have no means to describe accurately the delays of the channels or the execution times of the individual transitions. SDL is a specification language which is being used extensively in the telecom area and some of the largest software systems in the world are specified in SDL. In a nutshell we claim that SDL systems are reactive, asynchronous, large and concurrent, which are all aspects which are known to add to complexity and to highlight the need for improved methods. On the other hand we claim that most SDL systems are fairly simple wrt. data, or that the data aspects can be handled isolated from the aspects which make up the core of SDL (as mentioned above).

1.2.3 Progress, Confluence and Reduction

The reduction technique that we have defined in this dissertation eliminates the internal communication within a subsystem. Thus the result is a process definition of a subsystem that describes a behavior which, from the outside, is non-distinguishable from that of the original subsystem. The reduced process can be used in other analyses of enclosing systems, but it is not meant to replace the original in the system development towards the final implementation.

Thus the reduction may result in:

1. A more compact functional specification;
2. Improved overview and understanding;
3. Simpler analysis on higher levels.

It is, however, not certain that the reduction algorithm is applicable to a given system. Our technique examines the system for *progress* and *confluence*. A system which is progressive and confluent is reducible.

Progress is related to termination of programs. A progressive system will not turn into deadlocks or livelocks. This thesis is not very preoccupied with determination of progress other than the fact that progress is a prerequisite for determining confluence.

Confluence is that different execution paths lead to the same end result. Said differently, confluence between an external and an internal channel means that the end results are insensitive to the order in which signals are handled on the two independent channels. Our Mn-procedure may determine confluence in cases where we are certain to have a finite execution tree from every system state (weak progress). Why it is not trivial to assert the confluence of an asynchronously communicating system is the fact that internal signals may trigger more internal signals which in turn trigger other internal signals etc. These internal signals may freely travel across the communicating system and their interleaving and concurrency cannot be determined in advance.

When reducibility has been established, the reduction algorithm is simply to choose the simplest strategy for execution (since all eligible strategies lead to the same end result) and eliminate all other possibilities. The chosen strategy is to execute all internal signals before the next external one is consumed.

The result of the reduction algorithm, the reduction, can be described as an SDL-like process.

1.2.4 SDL is a real language

SDL is a language which has been in real use in industry for a number of years. This means that it contains features and mechanisms which are beyond the simple core of communicating finite state machines. We show that our Mn-procedure with some extension and modification, can cover also the extra features of SDL.

We consider:

1. save – the active withholding of the next signal for consumption;
2. timers – an imperative way to cope with time;
3. procedures – structured way to reuse behavior patterns;
4. pure types – general structures for reuse;
5. inheritance – the object-oriented way to describe concept hierarchies;
6. virtuality – the object-oriented way to describe polymorphism.

We have also considered what complications can be introduced by data.

1.2.5 Desired non-determinism

SDL-92[78] introduces non-determinism through spontaneous transitions and anyvalue expressions. We show how these concepts can be included in our strategy and we show how we can improve the SDL mechanisms for better specification of fairness.

We also cover the fact that implicit non-determinism may be desirable. Explicit non-determinism is when there are specific language constructs which clearly express that “here there is non-determinism” such as spontaneous transitions and anyvalue decisions. Implicit non-determinism stems from the race condition between signals of different channels. We suggest a new SDL mechanism (spontaneous save) which makes it possible to describe reductions also in cases with implicit non-determinism.

1.2.6 The Mn-approach — does it scale?

We have showed that the Mn-procedure adapts easily to most significant features of SDL, which in practice means most significant features of reactive systems. But it is important to consider how well the Mn-procedure may work on large systems. Is the method such that its supposed applicability is prohibited by the need for an enormous amount of resources of speed and storage?

The analysis of concurrent systems are usually badly crippled by the “state explosion” syndrome. The number of possible states quickly exceeds any number which can be processed in available time and space.

Fortunately our Mn-approach scales remarkably well. Very large bulks of the work is linear wrt. the number of components in the system under analysis. The complexity rises with the existence of feedback loops, but the complexity rises in a fairly controllable way.

1.2.7 The Mn-approach as a base for methodology

The Mn-approach can be seen as a technique for verification, a technique for validation, a technique for documentation, a technique for reuse, a technique for evaluation or the theoretical background for design guidelines.

While many other verification and validation techniques appear to give verdict “correct” or “erroneous” to a given program or system, our approach is to bring into play design guided by analysis results.

Even though correct programs are highly desirable, incorrect programs may have a higher market potential since they are earlier in the marketplace. We try and focus on more aspects of systems than the binary distinction between correctness and faults.

1.2.7.1 A technique for verification

Practitioners hate formal verification. They believe it is difficult and time consuming and error prone. Most often they are right. The Mn-approach makes it possible to examine the reductions rather than the full system. In a reduction the questionable properties may be trivial to assert. If the reducibility has been established through automatic means, formal verification (almost) has been obtained without sweat.

1.2.7.2 A technique for validation

Validation is to assert the value of something. In our case we want to assert the validity of some software. While a full system may be difficult to overview and to play with, a reduction is more manageable. Its strong and weak points are more easily spotted and its undesirable behavior more easily traced. The interesting aspects of the reduction may then be traced backwards through the Mn-approach to their origin in the full system.

1.2.7.3 A technique for documentation

A reduction does not always appear simpler than the original even though internal communication has been eliminated, but sometimes the reduction does yield something which is simpler to overview and to explain. Since the reduction is in principle automatic, the reduction is not “yet another description” that needs consistency checking with the original.

1.2.7.4 A technique for reuse

A prerequisite for reuse is certainty that the candidate fulfills its purpose in the new context. A reduction may be a compact description of a reusable component. By examining the reductions the potential reuser can be certain that the component is applicable without looking into the full amount of details of the reusable components.

1.2.7.5 A technique for evaluation

We claim that real systems typically have complexities in a few specific parts while other large portions are fairly straight forward. It may not always be obvious where the real complexities are. The Mn-approach discloses concurrency complexities, and we have devised an evaluation scheme which makes it possible to create an “Mn-profile” of the system which indicates areas of complexity and the priority in which these areas should be considered.

1.2.7.6 Background for design guidelines

Accepting the Mn-profile as a fruitful measurement of complexity, it is reasonable to look into ways which make it more probable to avoid unintended complexities. We present a set of concepts and guidelines which help minimize the problems of the Mn-approach. Thus if the designer follows these guidelines chances are that the Mn-approach may be applicable and fruitful reductions obtainable. We call this “confluent design”.

1.2.8 The Mn-approach integrated with other techniques

The Mn-approach is a very friendly technique. It integrates well with other techniques both inwards (using other techniques to achieve reducibility) and outwards (using reductions in analysis of enclosing systems).

1.2.8.1 Using auxiliary techniques inside the Mn-approach

Our Mn-approach relies on deciding that the system under analysis is progressive. For this purpose we shall often use methods which are not necessarily part of this thesis. There is a vast literature of proof techniques to assert termination e.g. from rewrite systems.

Furthermore we may run into situations which appear non-confluent in our Mn-procedure, but which through more thorough analysis can be proven to be unreachable. Backward execution is a possible way to assert that a given complete state is not reachable.

Symbolic execution in general is used to cope with data. We do not introduce any special techniques for this in this dissertation.

1.2.8.2 Using the Mn-reductions in analysis of enclosing systems

The Mn-approach turns a system component like a chameleon into something else more suited to the environment. Other techniques applied to an enclosing system may preferably use reductions of the components because then the complexity of the analysis on this level may become more manageable. Reachability techniques implemented in commercial tools may handle larger systems without meeting the state explosion boundaries. Techniques which check for specific properties e.g. a temporal logic formulae spend less time and resources to come up with the answer.

1.2.9 Concluding executive summary

Our Mn-approach is a flexible approach. It integrates well with other techniques. It focuses on problems of concurrency and leaves algorithmic problems of data alone.

Our approach can be applied to small, interesting and difficult theoretic problems as well as large, complex and trivial industrial problems. The technique seems to scale well.

We include in the Mn-approach methodology which helps the designers achieve reducible systems and to utilize reduction for a number of attractive purposes.

We suggest added value to some SDL language features to facilitate describing systems in a way which is compatible with the Mn-approach.

1.3 Technical summary

Take an ambitious urge and a simple approach, and study how far the simple approach can be carried and extended to meet the ambition. This is what this thesis is about.

We had the ambition to use SDL as the specification language of larger SDL systems. This was motivated by the idea that a practitioner would like to express his design in as few languages as possible. In order to use SDL as its own specification language, it was necessary to reduce the large SDL descriptions to simpler ones which eliminated the internal aspects of the SDL system.

We also had the ambition to use this monolithic SDL approach as the fundament for a bridge between theoretical verification and practical validation of SDL systems. This thesis gives a skeleton for such a bridge.

1.3.1 The bridge

The theorist studies small, but intricate problems. He believes some day he will find a way to scale the method to larger and more realistic situations. The practitioner engineers large, but mainly trivial problems. He believes that solving big problems is a matter of working habits and notation rather than formal theory. He believes that some day formal techniques may be applicable to his field of work, but he does not really believe he will live to see it.

This thesis takes the simple idea of structural reduction and shows that it can be used on a couple of the small, but intricate problems of the theorists. It takes the same simple idea and shows that it is possible to device a method with guidelines for making realistic systems with improved quality. We define metrics to measure real systems based on the same simple idea.

The simple idea can be summarized in the following conjectures:

1. Good quality systems are such that each structural concept is reducible to a process.
2. The effort needed to establish that a unit is reducible is a good measure of how complex the unit is.
3. The most important resource for making good quality systems is the designer himself, and our approach is a way to support him validating in parallel with designing.

The bridge from theorists to practitioners by our Mn-strategy consists of the following building blocks:

1. The Basic Mn-procedure
2. The General Mn-procedure
3. The Mn-approach to validation
4. The Mn-metrics, complexity profiles
5. The Mn-method “Confluent Design”

1.3.2 The Basic Mn-procedure

Our simple approach to determine reducibility is to study all potential *race conditions*. A race condition is when signals from more than one input channel “race” to be consumed first, and it is significant for the final result which one “wins” the race. We present a simple reduction algorithm which applies if the original system is confluent. Confluence means that all possible executions from a given complete state give the same final result. We show that confluence can be determined by examining all the potential race conditions. This idea is not new to practitioners, but this thesis shows how you can apply this idea systematically, and be certain that the system is confluent.

Assumptions:

1. A system of one process only, with one external input and one internal input channel and one external output channel.
2. The system is progressive which means that it terminates for any finite external input.
3. The following features of SDL are not used: save, non-determinism, data, timers, procedures, object orientation, services, priority signals.

Results:

1. Whenever the Mn-procedure returns with success, the system is confluent, meaning that all race conditions are insignificant wrt. the final result.
2. If the system is progressive and confluent, it is reducible. The reduction is easily reached by executing the system systematically giving absolute priority to internal signals.

Problems:

1. The Mn-procedure may return with failure even though the system is confluent.
2. The Mn-procedure may not terminate. This can be remedied by simple pragmatics.

Figure 1: Basic Mn-procedure

The algorithm of the basic Mn-procedure as summarized in Figure 1 (p. 8) is based on examining all possible race conditions. We show that it is sufficient to check all possible minimal non-confluence patterns, which are based on start situations consisting of one

basic state, one external input signal and a sequence of internal input signals. We construct a transition system M0 which has as initial nodes pairs of complete states. The pair has one element which is the start situation where the external signal is executed first and then the first internal signal. The other element of the pair is the start situation where the first internal signal is executed first and then the external signal. From these initial nodes transitions correspond to execution of an internal signal. If the nodes have equal elements in the pair, the branch can be pruned. If all branches are pruned, the procedure concludes that the system is confluent.

It is not always sufficient to study only the transition system M0. Since we are only interested in differences which can be observed externally, we accept that the two complete states of the Mn-node have differences in the internal channels. Such internal sequence permutation leads to the construction of transition systems on higher generations M1, M2 etc. Higher generation transition systems represent consumption of internal signals produced on lower generations.

The Mn-procedure examines all potential non-confluence patterns, not necessarily only the reachable non-confluence patterns. This is the reason why the Mn-procedure is not a decision procedure which determines exactly when the system is confluent. We show that by using other ad hoc methods to prove that the non-confluence patterns are not reachable, the Mn-procedure can be supplemented to conclude reducibility in a larger class of processes.

We also show that Mn-reductions are reductions in terms of Kwong as summarized in Figure 2 (p. 9).

Assumptions:

1. X is reducible because it is progressive and the Mn-procedure has succeeded in finding it confluent.
2. X-red is X reduced by our reduction algorithm.

Results:

1. X-red is a Kwong-reduction of X
2. From Kwong-reducibility it follows that such properties as deadlock freedom and homing are preserved.

Figure 2: Mn-reduction is Kwong reduction

1.3.3 The General Mn-procedure

The Basic Mn-procedure is a theoretical framework which is too restricted to be of much use other than with rather uninteresting systems, even though we prove reducible a system which is not so simple to see is reducible and where a proper invariant to prove reducibility is not so obvious, either.

We show, however, that the Mn-procedure with minor modification can be extended to work for much more general systems. We find that full SDL systems can in principle be processed provided that the data expressions can be handled by symbolic execution as indicated by our summary in Figure 3 (p. 10).

To cope with non-determinism we suggest two new features to SDL, fair non-deterministic decisions and spontaneous decisions. We show that this greatly enhances the expressiveness of SDL in connection with reductions.

Assumptions:

1. The system is progressive.

Results:

1. *Multiple processes and multiple channels*. We show that systems with more processes and channels still can be seen as one process. The Mn-procedure is basically linear wrt. number of components because the general Mn-procedure to a large extent can be executed piecewise component by component.
2. *Save*. By introducing semi-stable states containing saved internal signals, we find that save is very practical for ensuring confluence.
3. *Non-determinism*. The nodes of the transition systems Mn are expanded to tuples. We introduce fair non-deterministic decisions to help establish progress and spontaneous save to describe acceptable race conditions.
4. *Timers*. Timers can be handled as a special case of non-determinism. We acknowledge the lack of means to reason about durations and time constraints.
5. *Procedures*. By applying a simple transformation scheme procedures are easily managed in the general framework.
6. *Object orientation*. When types are reused (e.g. by inheritance), the Mn-procedure analysis of the type can be reused in the analysis of the entity in which the type is being reused.

Figure 3: General Mn-procedure

During our discussion of the general Mn-procedure we apply the Mn-procedure to a couple of well known examples from the literature, the Alternating Bit Protocol and the Brock-Ackerman anomaly.

The reader may be surprised to find as a result in Figure 4 (p. 11) that reducibility does not mean error-free. Our approach argues that reductions make it easier to see that the system contains problematic areas, and we find during the analysis of the Alternating Bit Protocol that certain assumptions has to be made for the timer, otherwise the reduction contains an internal error. This highlights that our approach is a monolithic approach which means that we concentrate on the SDL description alone and do not focus on dual descriptions which should be proven consistent with the SDL description.

The Brock-Ackerman anomaly summarized in Figure 5 (p. 11) was used to show that our reduction strategy seems to capture the essence of SDL systems.

1.3.4 The Mn-approach to validation

The Mn-procedure is a procedure which determines reducibility. The Mn-approach is to apply the Mn-procedure and derived reductions in validation of systems.

Purpose:

Communication over lossy channels with only one control bit.

Included features in our version:

1. fair non-deterministic decision
2. timer
3. save

Results:

1. The system is automatically proven reducible when progress is established manually.
2. The reduction shows that the function of the system is exactly what it should be namely to relay the input message.
3. The reduction shows also that the system is strongly progressive meaning that no internal signals will remain saved indefinitely.
4. Reducibility is not the same as error-free.

Figure 4: Alternating Bit Protocol by Mn-procedure

Purpose:

To show that history relations could not quite capture the essence of asynchronously communicating finite state machines.

Included features in our version:

1. Explicit fair merge modeled by spontaneous save
2. Object orientation with inheritance and virtuality

Results:

1. The reductions made through the Mn-procedure can be used compositionally in the analysis of enclosing systems. The difference which disappears when expressed in history relations is preserved in our extended SDL reductions.
2. The example exhibits how reducibility can be used in connection with object orientation.

Figure 5: Brock-Ackerman anomaly by Mn-procedure

The Mn-approach is basically monolithic which means that we concentrate mainly on one description. This description consists normally of a structure where each component may be eligible for our Mn-approach. Compositionality of our analysis approach becomes important as summarized in Figure 6 (p. 12).

Even in a monolithic approach the single description may come in several versions and it is interesting to determine whether the newer version is a refinement (implementation) of the former. We device a technique to utilize the Mn-reductions to establish refinement summarized in Figure 7 (p. 12).

Assumptions:

1. System A contains block B.
2. B is reducible and the reduction is B-red.
3. A-subst is the system that takes A and substitutes B by B-red.

Results:

1. A-subst is reducible **iff** A is reducible.
2. The reduction of A can be found by reducing A-subst.

Figure 6: Compositionality of the Mn-approach

Assumptions:

1. Versions V0 and V1 are both reducible
2. Eventual interface mappings can be described in SDL

Results

1. Refinement (possibly interface refinement) relation between V0 and V1 can be determined through a state by state, transition by transition comparison of the reductions.
2. The refinement establishment is simple to perceive and accept by practitioners.

Figure 7: Refinement by the Mn-approach

The Mn-approach can be made more pragmatic by combining it with other methods. In this respect the Mn-approach is very “friendly”, it seems that it cooperates well with a number of other quite different approaches. We summarize this in Figure 8 (p. 13).

1.3.5 Complexity profile and complexity estimates

As a way to estimate the complexity of the system itself we device a metric based on the M0 execution of the Mn-procedure summarized in Figure 9 (p. 14). The M0 execution is the first level execution of the Mn-procedure.

We also developed an estimation model for how many Mn-nodes the Mn-procedure would have to produce for a given system. This is summarized in Figure 10 (p. 15).

1.3.6 The Mn-method, “Confluent Design”

We present a framework for understanding real, reactive systems and place the Mn-approach in this setting. The result is an Mn-method which is basically a set of guidelines which we call “Confluent design”. We summarize in Figure 11 (p. 13) the framework for quality development which we have named the *software distillery*. In Figure 12 (p. 14) we summarize a classification of development comprehension profiles.

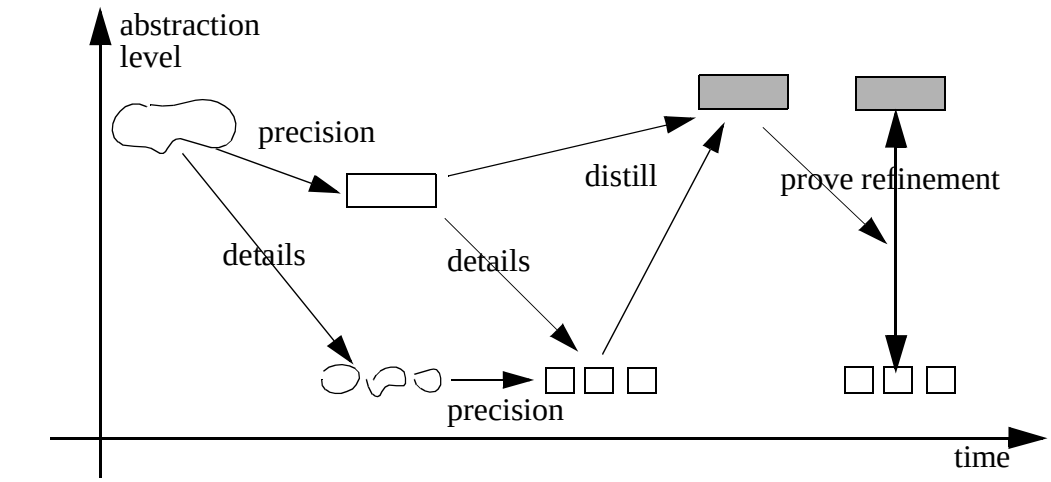
Purpose:

1. To utilize other methods to support the Mn-approach
2. To use the Mn-approach to support other methods

Results:

1. Use run-time checks to obtain a conditional reduction
   - 1.1 define exceptions onto monitoring layers for impossible transitions
   - 1.2 define exceptions when limits to resources are exceeded. If we define bounded channels, it suffices to apply M0 (the first level of Mn-procedure) to establish confluence. In principle exhaustive simulation could decide confluence.
2. Use ad hoc invariants to eliminate unreachable non-confluence patterns and other situations which cannot occur.
3. Use backwards execution to eliminate unreachable non-confluence patterns.
4. Use Mn-reduced components to make enclosing systems more manageable by common reachability techniques such as Supertrace.

Figure 8: Pragmatic Mn-approach



Precision = formalize, make more narrow, supplement

Details = decompose, break down, reveal

Figure 11: Software distillery

We provide a strategy (“algorithm”) for system development which should increase the chance of producing systems which are comprehensible and reusable. The idea is summarized in Figure 13 (p. 15). We apply the distillery approach to tie the descriptions into the SDL description. Reductions are used for a more compact but faithful “specification” of the component. Following maintenance of the SDL description, the establishment of reducibility leads to a new specification by reducing the description again. Much of the work related to establishing reducibility can be carried over from previous work.

Purpose:

To estimate complexity of system.

Technique:

1. Calculate the Z0, i.e. the set of initial nodes of M0 of the system.
2. Classify the nodes according to the following categories:
   - 2.1 Confluence
   - 2.2 Non-confluence
   - 2.3 Sequence permutation
   - 2.4 State different
   - 2.5 Omitted
   - 2.6 Double sided error
   - 2.7 Single sided error
   - 2.8 Warning
   - 2.9 Save-problem
   - 2.10 Non-determinism problem
3. Normalize the numbers
4. Apply some proprietary weights on each category and calculate a complexity index.

Figure 9: Complexity profile based on Mn-procedure

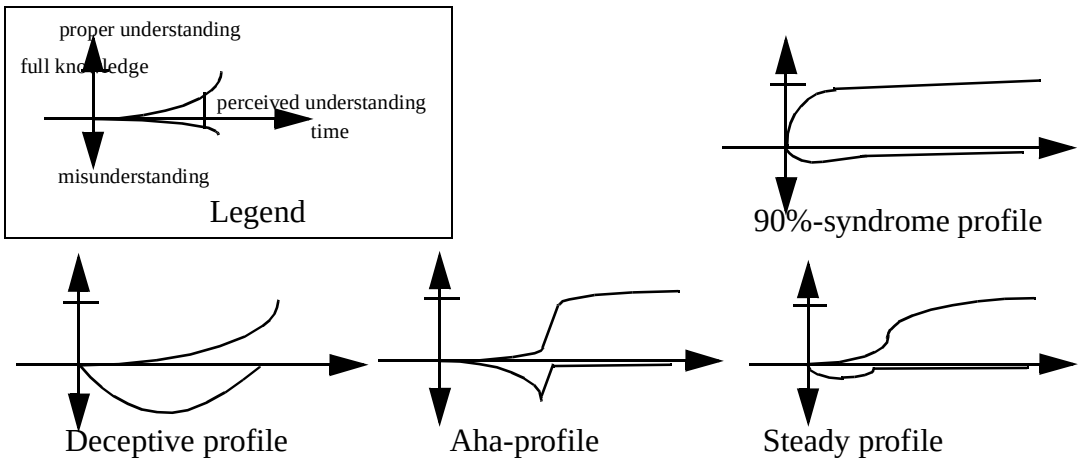


Figure 12: Comprehension profiles

We summarize the guidelines for how to make individual processes confluent in Figure 14 (p. 16).

We claim that adhering to “confluent design” increases the chances of experiencing that the development follows a steady profile as described in Figure 12 (p. 14).

Assumptions:

1. The number of processes is p.
2. The number of basic states per process is on the average s.
3. The number of external signals per process is on the average e.
4. The number of internal input channels per process is on the average c.
5. The number of internal signals per channel is on the average i.
6. Non-determinance factor is n. The non-determinance factor is how many more nodes there are on the next level of execution due to non-determinism. E.g. if every transition contains a non-deterministic decision which branches in two possibilities, the factor is 2.
7. The non-conformity factor is f. The non-conformity factor measure the number of nodes which need another level of Mn-procedure compared with the total number of nodes on this level.

Results:

1. The number of potential non-confluence patterns is $t=(p*s*(e*c*i + i*i*c*(c-1)/2))$
2. The number of nodes needing another execution level is t*f, and the result of another execution level from these nodes will result in (t*f*n*i*c) new nodes. The level factor is thus $a=f*n*i*c$.
3. If we accept 5 levels as the maximum we get the following total number: $T=t*(1+a+a^2+a^3+a^4)$.
4. If the parameters s,e,c,i vary considerably between the processes, t should be calculated as a sum over the actual processes.

Figure 10: The estimated complexity of the Mn-procedure

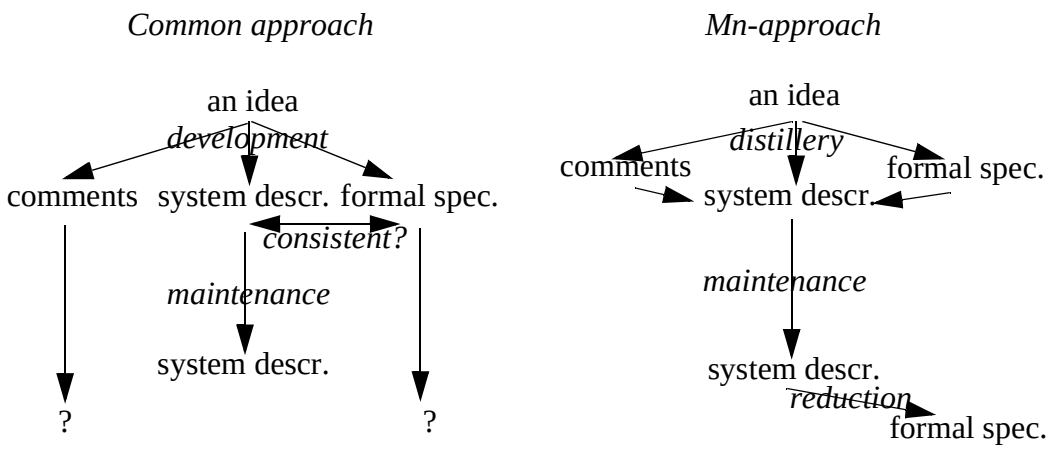


Figure 13: Mn-method for improved quality systems

1. *Categorize* the components according to this rough scheme:
   1.1 *One-input-channel* process (The process has only one input channel and therefore it cannot show any non-confluence.)
   1.2 *Multi-lane process* (The process is actually a collection of “lanes” with one input and disjoint output. The clue is that the outputs are never merged.)
   1.3 *Channel-state mapped* process (The process is such that for each basic state there is only one channel from which it accepts input.)
   1.4 *Merge* process (The process has potential non-confluence patterns which must be considered more closely.)
2. Make a *complexity profile* of each merge process
3. Order the merge processes according to a *complexity index*.
4. Take the most complex processes first and continue in the order of the complexity.
5. For each process proceed to analyze and possibly modify the critical points according to the following succession:
   5.1 Clarify the non-confluent situations
   5.2 Continue M_0 on the “state different” cases
   5.3 Perform generation change on the “sequence permuted” cases
   5.4 Try and see if external stuttering could be used on the generation changed cases which turned into non-confluence
   5.5 Analyze the auxiliary category situations

If confluence cannot be obtained this should be properly documented. A case which shows that there is actually an error should be produced.

Figure 14: Confluent Design

1.3.7 Future research

Even though we have conducted a rudimentary industrial case study, we realize that the Mn-approach to system validation and the Mn-method for synthesizing systems need further pilot studies. The empirical base for stating that the Mn-approach is applicable to real systems should be made more solid. We have realized that it is necessary to support the Mn-approach with an Mn-tool which most effectively should be built on existing SDL tools.

There are some areas which this thesis has chosen to neglect regarding the Mn-approach. More practical ways to determine progress should be included in the method. There should be much to gain by systematically combining the Mn-approach with traditional proof approaches taking advantage of proven invariants in the elimination of unreachable, but problematic race situations. There is a need to look into the theory and practice of handling data symbolically.

Since we concentrate on real, reactive systems, the improved handling of real time constraints is interesting. Can the focusing on confluence and reduction be combined with real time constraints?

1.4 The nature of SDL systems

Here we give an introduction to SDL – the language and conceptual framework of this thesis.

1.4.1 SDL – the language

SDL (Specification and Description Language) was developed as an *answer* to *questions* in ITU (International Telecommunication Union) in their consultative committee (CCITT) on languages. The first version was standardized in 1976 and included not much more than a few graphical symbols in the domain of telecommunication.

New versions of the recommendation Z.100 were presented in 1980 and 1984. The 1984 version was very much a full fledged language, and tools emerged to support it. In 1988 a major revision was undertaken and the language got a more formal semantics definition and a precise data concept (ACT ONE) [25].

In 1992 another important revision took place as object orientation was smoothly introduced. It was also made possible to express non-determinism [78]. In 1996 only minor corrections and supplements were put in an addendum to the 1992 recommendation [83].

The semantics of SDL is defined in the recommendation Z.100 through informal English. There is also a formal semantics [79], which is based on MetaIV which is a variant of VDM [87]. The formal semantics has played an important role in two ways:

1. The tool vendors consult the formal semantics when they are uncertain about the interpretation of a construct.
2. The making of the formal definition revealed a number of inconsistencies in the informal semantics.

Contrary to what one could expect the formal semantics has not been used much in verifying SDL systems.

1.4.2 SDL – the use and the users

SDL was brought forth in the area of telecommunication and it still has most of its supporters in that domain. Telecommunication was one of the first areas to make practical use of concurrent processing and the need for a precise attitude towards the perils of concurrency was critical for the success of some of the largest pieces of software ever created. In recent years other areas of computing have also approached concurrency and real time and therefore SDL and SDL-like approaches are becoming more popular.

In Norway there has been a very active SDL user group for a number of years. SDL methodology was developed in Norway for the production of MAREIK, the world's first system to provide fully automatic telephone and telex services to ships through sat-

ellites in the INMARSAT system [11] already in 1979-1981. From that time, SDL has been popular in the advanced telecommunication projects in Norway, and throughout the national Norwegian technology transfer program SISU [62] SDL methodology was central. New versions of the SDL methodology was developed in the SISU project resulting in a textbook [11] and an interactive CD-rom [12]. Norway also played a very active role in the development of the language itself especially during the introduction of object orientation in the study period 1989-1992.

The SDL users have a biannual conference which shows that the use of SDL spreads also to other areas than telecommunication and that the use of SDL is taken up also by universities and engineering schools [45; 46; 47; 120].

SDL is a language and it does not secure that its use by necessity leads to perfect results. The need for guidance and methodology became evident at an early stage and textbooks have appeared [5; 11; 106; 42]. Furthermore ITU has laid down certain guidelines for methodology in [26; 80; 84].

1.4.3 SDL – the supporting tools

The forces behind SDL, the telecommunication administration, were traditionally very affluent and could support large technology endeavors. Some of the largest software projects evolved in the telecommunication area. In this situation it was reasonable that tools were developed to support the use of SDL. The early tools were mainly graphic editors which supported drawing boxes with attached lines. In due turn these tools developed according to the development of the language itself into more language-oriented tools as they performed syntactic analysis and also static semantic checks. As the tools became more advanced, fewer tools were left in the marketplace. The large telecom actors like AT&T and Siemens made their own SDL tools which they kept in-house while they were trying to convince competitors how wonderful they were.

When SDL was formalized in 1988 code generation directly from SDL became the next field to cover. With it came also the possibility to simulate the SDL system execution on a “host computer” with more resources than the final target system. The simulation became a substantial contributor to the improvement of the development process and code generation helped to lift the focus from implementation to design.

The recent most advanced tools also include validators which perform reachability analysis of the SDL system. This again promises to improve the reliability of SDL systems. Along the same lines we find the tool development of integrated design and test tools for SDL.

As mentioned above, the development towards more advanced tools has left fewer commercial tools in the marketplace. Full fledged tools for SDL-92 include Geode from Verilog, France and SDT from Telelogic, Sweden. The general trend seems to be that these commercial tools make the in-house tools too expensive for even the big companies to maintain and improve.

In order to avoid a monopoly situation on the tools market, ITU has defined a Common Interface Format [85] which makes it possible to transfer SDL diagrams from one tool to another. This format makes it possible to have projects where both large tools are

used. It also makes it possible for small tool vendors to specialize in one area such as graphic editor, code generation or test generation, and attach to the more complete tools. It should also be possible to build up a complete tool from such specialized small tools.

1.4.4 SDL – main concepts

SDL is a real language made by a committee over many years. From this it is obvious that SDL is not a small, beautiful language on which the most perfect theories can be made. On the other hand, the theories which can be made with SDL have the chance of having practical impact since many very interesting systems are specified with SDL.

In this section we shall go through the most central mechanisms of SDL. We will not go into great detail and we do not spell out all the options and variants.

1.4.4.1 SDL system

An SDL *system* is the highest aggregation level which SDL covers. The SDL system communicates with its *environment* by asynchronous *signals*. A system consists of a set of communicating *blocks* (or *block instances*). The blocks communicate via *channels* by means of asynchronous signals.

1.4.4.2 SDL channels

The SDL channels deliver the signal from the Sender to the Receiver without loss. The channel may be either *delaying* or non-delaying. A delaying channel delays the signal some unknown duration of time from the sending to the reception. A non-delaying channel delivers the signal to the receiver in the same moment in time as it was sent. There is no guarantee, however, that the Receiver will consume the signal immediately.

Signals cannot overtake each other on the same channel.

The channel may be either unidirectional or bidirectional. The bidirectional channel is just the combination of two unidirectional channels. A channel has a *name* (which may be omitted) and for each direction there is associated a *signal list* which specifies what signal types may pass over the channel.

1.4.4.3 SDL blocks, block types and block instances

An SDL block is defined by a block definition and it is a singular object. To define patterns for several similar block instances, we use block types. Block instances are specified statically, i.e. there is no way SDL can create block instances during execution.

A block (block instance) may either contain a set of blocks and channels (such as a system), or it may contain a set of *processes* (*process instances*) and *signalroutes*.

A signalroute is identical to a non-delaying channel and in this thesis we shall refer to them also as “channels” to avoid unnecessary confusion.

The processes communicate asynchronously in the same manner as the blocks. In fact a block does not have its own behavior. It has the combined behavior of its constituents.

The processes are by definition concurrent.

A block type has *gates* to specify the communication interface between the instances of the block type and their surroundings. A gate has a name and specifies the input and output signals permitted to pass through the gate. The gate may also have a gate endpoint constraint which specifies what the gate may connect to.

1.4.4.4 SDL processes, process types and process instances

An SDL process is defined by a process definition and can be seen as a singular object. To define patterns for several similar processes, we use process types. Process instances are objects from a process type[1]. Processes are defined by a *process graph* or as a set of communicating *services*. In this thesis we shall concentrate on the process graph version.

A process type has *gates* to define the communication interface in the same manner as block types.

1.4.4.5 SDL process concept

An SDL process is defined by a finite set of *basic states*. In SDL they are called plainly "states", but we use "basic state" to distinguish them from "complete states" which also take other aspects into account.

The SDL process has a *valid input set*, which consists of all the signal types of the signals which may be received by the process.

The basic state and the received signal (which must be a member of the valid input set) together determine which *transition* the process will execute. A transition brings the process from one state to the next (which may or may not be the same state).

A transition starts by *consuming* the first signal of the *input port*. The input port is the queue of all signals received by the process. If the process has more than one incoming channel, the signals will be merged into the input port in a FIFO way at reception. There is theoretically no limit to the size of the input port.

A transition may *output* new signals to other processes (via gates and channels).

Furthermore a transition may change the internal *data variables* of the process. An SDL process is actually an *extended finite state machine*. The set of basic states is finite, but the complete state of the process is not finite because most data variables have an infinite value space (and the size of the input port is unbounded). In this thesis we shall refer to the data variables of the process as "data". The data capabilities of SDL is not a major point in this thesis.

1.4.4.6 SDL data

SDL data variables are objects of data types. Data types are defined through the language ACT ONE or through a special recommendation Z.105 in ASN.1[81; 135]. Data variables look in SDL very much like in any other programming language such as C++[130] or Simula[7].

1. or for historical reasons also from process definition but this will not be covered here

1.4.4.7 SDL timers

SDL *timers* are signals which theoretically come from the process itself at certain pre-specified times. A timer may be *set* to a specific time, *reset* if there is no use for it any more, *set again* if the time information changes, or it may *expire* (time out). When the timer expires a signal with the name of the timer is received just as an ordinary signal by the process.

1.4.4.8 SDL procedures

SDL processes may also contain *procedures.* Procedures contain a process graph and defines a sub-behavior of the process. A procedure may be *called* inside a transition and when it returns it will *return* to where it was called. This is very similar to functions in C or procedures in Simula.

The difference compared with common programming languages is that an SDL procedure may contain states. This means that the process may halt and wait in the middle of the execution of a procedure.

1.4.4.9 SDL services

A *service* is a constituent part of a procedure which is divided into communicating, alternating services. The services of a process alternate, which means that only one of them will execute at any one point in time. The valid input sets of the services must be disjunct and thus the signal received by the process determines which service will execute it.

1.4.4.10 SDL save

A signal may be *saved* instead of consumed in a state. To be saved means that the signal will not be handled as long as the process is in this state. When the state changes, the saved signals will be the first to be consumed.

This is (almost) the only way SDL can permute the order in which the signals are consumed. There is also a priority input mechanism, but this will not be used here.

1.4.4.11 SDL non-determinism

Non-determinism in SDL comes in two flavors: *spontaneous transitions,* and *any-value expression.*

The spontaneous transition is a transition with the signal name **none**. **none** is not a real signal name, but a keyword. It means that this transition *may* occur, but it is not certain that it does occur, when the process is in the state where the transition is specified. The spontaneous transition needs not consume a signal in order to trigger.

The any-value expression is a construct which returns any value of the data type specified. SDL does not require that the any-value expression is implemented by a stochastic distribution such that the implementation may just as well choose one constant value every time the any-value expression occurs. Any-value expressions occur often in *decisions* to specify randomized variants of the transition.

A decision is a construct which branches according to the value of the data expression in it. Its branches are labelled with different ranges of the type of the expression.

1.4.4.12 SDL object orientation

SDL introduced object orientation in 1992, and object orientation with inheritance and virtuality can be applied to almost all types of SDL.

Inheriting a block type means to add more blocks (processes) and channels. The new entities may be connected to the old ones.

Inheriting a process type means to specialize the behavior by introducing new transitions because new states or signals are introduced. SDL is one of the very few languages which specifies the inheritance of behavior in a useful way.

Virtuality means to redefine in specializations patterns of the type which is inherited. This is very practical to make small modifications to the type inherited in the new specialization.

The SDL approach to object orientation is typically in the European tradition such as Simula and Beta[97].

1.4.4.13 More?

There are more features to SDL, but the reader is referred to other sources to perfect himself in SDL.

For a general tutorial to SDL and how it should be used, the reader should consult the special issue of the journal CN&ISDN [57]. This gives a brief introduction, and complementary education should be sought in the textbooks [5; 11; 106; 42].

For a full definition of SDL there is only one place to look and that is in the Recommendation itself [78; 83].

1.4.5 SDL – pragmatics

The language which is used to prescribe a system, restricts the way the system will behave. On the other hand certain domains have the need for certain features and will look for languages which include these features.

Traditionally SDL systems are:

1. reactive,
2. concurrent,
3. asynchronous,
4. large,
5. often simple wrt. data.

None of these characteristics are absolute. It is possible to find SDL systems which violate one or more of these properties. Still they represent experience from many years and many systems. We go in greater detail into this in Section 5.1.2 (p. 178).

That a system is reactive, concurrent and asynchronous can be said to be due to the language SDL since this is the way SDL sees the world. On the other hand, the relative success of SDL in the area of telecommunication indicates that such systems favorably may be seen as reactive, concurrent and asynchronous.

That some of the largest systems in the world (telephone switches) with millions of code lines have been specified in SDL, indicates that SDL is a language which “scales”. Similar to the distribution of the system itself, an SDL description is well distributed and a number of developers can work in parallel towards a common cooperating system.

That data is often simple, is the most dubious statement. We can find SDL systems with considerable data and where data variables are absolutely necessary for the functioning of the system. Still there are not many SDL systems where SDL is used to define complicated algorithms, or SDL systems which specify administrative database solutions. The point of using SDL is to handle the concurrency and the flow of control, not the complexity of data. SDL has the power to simulate a Turing machine even without the data variables [13] and *with* the data variables it is even more trivial to specify very complex data problems.

However, this thesis does not intend to solve data complexities.

1.5 Motivation

In this section we try to give some motivation for why we believe that the Mn-approach is fruitful in practical software development.

Having worked with concurrent systems specified in SDL for many years, having created SDL tools [55] and having been involved in creating SDL methodology [11; 12], we have reached some assumptions about real reactive systems and their development. As a student of more formal computer science and attempting to convey such ideas to practitioners we have reached other assumptions about developers.

1. There is a need for formal verification and pure testing just cannot do the job properly.
2. What the designers claim to perceive, should be manageable through formal analysis on a modern computer.
3. Designers and programmers like to reason in imperative terms.
4. A variety of notations is not a goal in itself. The amount of inter-notational conflicts in concept and understanding increases with the number of notations that must be used.
5. Reuse requires compact, but verifiable, correct specifications.
6. For formal verification and validation to become commonplace, the results must be believable to the practitioner and automated by tools. Furthermore it must be conceivable that real size systems can be handled.
7. For any type of analysis, it is important that earlier results can be reused in the analysis of new ones or modifications of the old. We need compositionality of the analysis.
8. Large systems are largely trivial, but may contain intricate parts.
9. The designer (programmer) is the most important resource in creating correct programs. It is not difficult to make systems which are virtually impossible to analyze. Methodology for the creation of verifiable systems integrated with formal evaluation of the system is cost effective.

These statements are definitely not provable, but we shall go through them one by one and give some explanation for our attitude.

Need

The need for formal validation has been demonstrated by a number of scholars [34] and more frequently now than before successes of formal methods in producing better quality are reported [10].

Capacity of designer

When a computer system is finished, it is the belief of the designers that the system is correct. They believe that they have been able to perceive the effects of their design. Very often this is not the case, and it is proven through testing and verification that the designers were not quite able to manage the task. Still there is a big discrepancy between the complexity the designers see in their design and the complexity sometimes demonstrated by formal verifiers! Validators applying reachability techniques suffer from severe state space explosion problems, and proof assistants produce a massive number of proof obligations which must be manually proven. This does not correspond well with the feelings of the designers.

Our stand is that when the automatic validators must cover a multitude of cases, this is probably a sign of unmanageable complexity of the analyzed system. There is a fair chance that the designer should look into the design again.

Imperative languages

Even though there are scholars who advocate the adverse, our experience is that programmers and designers prefer to express their thoughts in sequences of actions or imperatives. Sometimes combining such action sequences with more declarative statements is reasonable, but the notions of sequencing and of cause/effect seem to guide the thought more effectively. Even programmers of such declarative languages as PROLOG and Z acquire a programming style which shows that they think imperatively when they program.

SDL is a language which is imperative with a dash of declarativeness. The action sequences of the transitions are balanced by the declarations of basic states in the processes which actually represent invariants. Often the basic states can be interpreted to describe the whole history of actions leading to it. Thus only the knowledge of the basic state is sufficient for the future execution.

Few notations

It has been used as an argument in favor of object orientation that the designer does not need to change paradigm during the software development. The more different notations needed in a system description, the bigger the chance that either the designer misunderstands a notation or that there are inter-notation discrepancies. More than one notation may be needed to express different aspects of the system, but preferably not more than one notation should be used to express the exact same aspect.

In SDL systems, we consider it an advantage if SDL can be used to specify the system by process behavior. Orthogonally MSC could be used to express inter-process interaction and the two descriptions can be compared for consistency. MSC is a language for message sequence charts standardized by ITU [86].

Reuse

Reuse has become a buzzword in computer science and it comprises a number of different aspects. Here we only want to make a point regarding the possibility to judge whether a given candidate for reuse is appropriate.

If we assume that the candidate for reuse is an SDL block of reasonable size, the designer may not want to look in full detail into the SDL details. The task of finding out what the SDL block actually does, may be quite time consuming and unreliable if the designer himself should be forced to look into the full SDL description. On the other hand relying solely on the informal comments to the block, may prove to be too imprecise. Other notations such as MSCs can also be used to supplement the original description, but they are often as incomplete as the informal description.

A reduction, however, which is made automatically from the original, and which is insensitive to the potential usage, presented in the very same language SDL, seems more attractive.

Credibility

It is not sufficient to *claim* that formal verification has taken place. The users of the system must *believe* that the verification has been performed. It is not obvious how this is achieved [107]. The practitioners of SDL will have serious doubts when a theorist claims that he has proved an SDL program correct. Often the practitioner will be right in his doubt since it turns out that what the theorist has proved correct was not the SDL program itself, but a simplified model (an abstraction). Is the proof still valid for the full SDL program?

A practitioner will be even more suspicious if he is presented a manual proof of the SDL program. This will normally include notation which is unfamiliar to him supplemented with informal statements that he can understand, but which he does not know whether he believes. If we add that the theorist probably is unfamiliar with SDL as a notation, the practitioner is uncertain whether the theorist has really understood the subtleties of the problem.

A practitioner is more susceptible to accepting a machine-generated proof since he is accustomed to accepting automatic means such as compilers and code generators. Still he will prefer that the intermediate results are given in a form that he can relate to, which means in forms close to SDL or MSC.

The optimal situation from the viewpoint of the practitioner is if he could understand the verification steps and read the verification results in the same language as he has made his description (i.e. SDL and MSC).

Compositionality

Since systems are never finished, but are being maintained almost before they are released, it is important that the efforts of validation is not lost once a single change is introduced into the system. For some of the available methods this is in principle the case. It is important that pieces of the system can be analyzed (partly) in isolation such that the results of the analysis can be applied directly in analyzing larger parts as long as the isolated piece has not been changed. This is what we call compositionality.

Variation in systems

A real system usually contains parts which are trivial as well as parts which are intricate. Assuming that this reflects the complexity of the problem itself and not the competence of the designers, this indicates that the same method for validation may not be applicable to all parts in the same way.

For a project leader it would be desirable to be able to assign the complicated, but small parts to a group of experts in that field. Other parts which were more like “digging a ditch” could be assigned to the more average software engineer. The experts could use whatever methods suitable for their isolated problem, while the software engineers used the default validation techniques.

The important issue would be that the methods of validation could be combined in flexible ways corresponding to the challenges of the problem and the software.

Methodology

It is a well known fact that the earlier in the development process an error or deficiency is discovered, the less expensive it is to correct the problem. This contrasts the fact that validation normally takes place closer to the end of the development. It would be easier if the designer created software which was easily validated, and that he actually performed certain validation efforts along with his designing.

The ambition

Our ambition is to contribute to the narrowing of the gap between theorists in the field of validating concurrent systems, and practitioners in the field of engineering reactive systems.

We want to make a method which has a low threshold for the practitioners such that he can experience positive effects of his validation efforts without having to put in a lot of time and resources.

On the other hand we want a method which is advanced enough to produce strong analysis results which can be reused later, and possibly also within other validation methods.

We also want to make the validation results available in SDL itself.

We acknowledge the fact that our method may not always succeed, but want that even when it does not succeed totally, we shall still reap valuable experience from the effort.

1.6 Background

In this section we want to give some insight into the work which has influenced our Mn-approach without going into detail about every theoretical approach which has played a role in forming the Mn-approach. We shall characterize the Mn-approach by its similarities and differences with other approaches.

1.6.1 The Mn-approach is validation-oriented

The distinction between “verification” and “validation” is used by some and rejected by others. Following Boehm [8; 11] “verification” is to establish the truth of correspondence between a software product and its specification, while “validation” is to establish the fitness or worth of a software product for its operational mission. “Verification” originates from “veritas” which is Latin for “truth”. “Validation” originates from Latin “valere” which means “value” or “worth”.

Opponents of applying the distinction argue that in order to assess value, one must describe the evaluation criteria, and therefore we are back to a specification, and validation reduces to verification anyway.

1.6.1.1 Dual descriptions

In most verification approaches there are a specification and a system description. The latter is normally prescriptive like a software program. The specification is normally declarative and often in the form of predicates. One may call this a “dual” description of the system and verification means to determine whether the two descriptions are consistent. Normally we try to find whether the specification is *satisfied* by the system model (the program).

The earliest approaches in program verification [69; 32; 70] had predicate logic specifications interleaved in the program text. The interpretation should be that the predicates should hold at these places they were put inside the program. A logical system (Hoare-logic) defined the logical relations between specification predicates and program statements. The predicate specifications were also very often called “invariants” as they described properties which were invariant whenever the program control passed this point in the program.

This tradition made its way into methods for software development through VDM [87] and into concurrent programming through the advent of CSP [70; 71], CCS [103] and LOTOS [9; 77]. For more on this tradition in the verification of parallel programs, we refer to Barringer’s survey [2].

Dual descriptions can also appear as two separate descriptions which are compared. This is often the case in developing SDL systems where MSC[1] descriptions and SDL descriptions appear side by side and their consistency is checked [40]. More about this in Section 1.6.2.1 (p. 30).

1.6.1.2 Monolithic descriptions

Even though we argue that the Mn-approach can be helpful for verification purposes our approach is more “monolithic” than “dual”. Our goal is more to explore the possibilities of the one description than comparing one description with another.

To explore the single description we apply transformation of the description. This is similar to program transformation. Program transformation may have different aims, but traditionally there has been two purposes of program transformation:

1. to make the program more efficient,
2. to transform a description in a wide set of concepts to a description in a more narrow set of concepts.

The first purpose was used in compilers to optimize the object code, in “peephole” optimization. The peephole is a segment of the program which then is massaged into a more effective sequence of instructions.

The first purpose is also the motivation of Darlington and Burstall [33; 22] when they argue that simple and comprehensible programs can be transformed automatically to more efficient executable programs. The idea is that what is comprehensible is not effectively executable. Modern systems still suffer from inefficiency and need optimization, but their incorrectness and general lack of reliability are more worrying than their speed.

1. MSC = Message Sequence Charts, standardized in [86].

Therefore the goals of the Mn-approach are rather opposite of the Darlington-Burstall aims as we want to make systems *more* comprehensible through system transformation. Efficiency is not the main issue as our transformed systems are not executed on the target configuration.

The second purpose of program transformation, where descriptions of rich conceptual frameworks are transformed into descriptions of restricted languages, are often used to make proofs and statements valid for restricted conceptual frameworks valid also for greater domains. The definition of SDL-92 [78] contains a comprehensive transformation section on how to transform all mechanisms which are not “basic SDL” into “basic SDL”. Then “basic SDL” is given a formal semantics in [79]. The formal semantics then applies to the whole SDL-92. For the Mn-approach we use this technique to explain how SDL procedures can be considered a small system of SDL processes and as such be covered by our general results (see Section 3.8 (p. 128)).

Monolithic approaches in the tradition of axiomatic specification include the specification languages Z [64] and Focus [15; 20]. Their approach in this context can be characterized by a belief that small steps in the development of specifications will minimize (or almost eliminate) the risks of describing undesirable features. Formal refinement of the specification in small steps should lead to a specification which is realizable on a computer system.

This attitude is similar to ours since one of our main postulates is that a multitude of different languages and paradigms does not help the designer in his effort to create a good system. He wants rather one powerful language which is associated with powerful tools and techniques to aid his understanding of the subject matter.

1.6.1.3 Reduction

The Mn-approach to system transformation is that of *reduction.* Reduction means that some parts of the original is removed, but the essence remains. What the “essence” is may change from situation to situation, but the idea is that the reduction should be able to play the role of the original in the discussion.

Our Mn-approach is in the tradition of Kwong [94] where the essence to be preserved is properties like deadlock freedom and homing[1] which are generally desirable properties of systems. The idea is that proving e.g. deadlock freedom of the reduction implies deadlock freedom in the original. Supposedly to prove the property is easier in the reduction than in the original.

We show in Section 2.5.3 (p. 75) that our Mn-reduction is a Kwong reduction and therefore our reductions also preserve these general properties.

In our Mn-approach we define that the “essence” of a system is its behavior as a machine which handles input and produces output. The internal distribution and communication are considered less important. We may say that the “functional” aspects of the system are focused.

What is the essence of the system may also be user specified. Specifying the essence of a system may get the form of a *filter*. The system description is filtered through this specified filter and a reduction is filtered out which preserves certain properties of the original depending on the filter. Seltveit has given a thorough survey of such specified

1. Homing means that there is a set of complete states which can be reached from any other complete state.

filters to reduce complexity in administrative systems [121]. Her approach is more structure-oriented than functionally oriented, but her goals are very similar to ours as the perceived simplicity of the reduction is considered important.

Reduction of complexity is also extremely important in automatic verification tools in the SDL domain. Since state explosion is the major obstacle to verifying real systems the reduction of the state space needed to be explored is decisive for the success. The most advanced reachability tool SPIN uses a partial order reduction method which reduces the state space relative to the temporal logic formula which is checked [75].

Other reduction techniques which are used include compression of data held by the algorithm and smarter ways to store the necessary information such as with binary decision trees and binary decision diagrams (BDDs) [28].We have not studied how such implementation oriented reduction techniques could improve the efficiency of the Mn-approach as the Mn-procedure has not seriously been implemented, yet. The techniques could probably be applied with the Mn-procedure as well.

1.6.1.4 Testing

Testing is the most common way to assess the value of an SDL system, or rather the implementation of an SDL system. Testing in the form of simulation is an increasingly popular way to assess the worth of the SDL system before it is implemented. Testing is different from verification as it relies on observing the results of executions. The specification can be used to define the test cases and the desirable result.

Testing in SDL environments is well covered in [52; 105; 24]. In general, testing has been closely connected to reachability analysis [141; 76; 123; 138; 124; 44].

The connection between testing and the Mn-approach is only indirectly and concerning the general philosophy of the approach. Testing, as well as the Mn-approach, examine cases which should not occur. The Mn-approach considers all potential problematic patterns regardless of their reachability. Testing may also run non-reachable situations for the same reason: it is in general impossible to know that a given complete state is reachable, and for robustness, such cases should be tested.

1.6.1.5 Evaluation of systems

The Mn-approach is different from the dual and other monolithic approaches by its explicit focus on evaluation. While the verification-oriented approaches concentrate exclusively on finding a proof that the program satisfies the specification and give a binary “yes” or “no”, we also make an effort to evaluate the program along a more informative scale. Verification-oriented approaches fail to appreciate the development of the program as such, but recognize the fact that failure to find proofs often results in improvements of the program when eliminating the counterexample.

We advocate to use the Mn-approach metrics to govern the analysis and to give an indication about the overall complexity of the system (part).

1.6.2 The Mn-approach is automata-based

Different approaches to system validation may have different semantic bases. The approaches based on proof theory are based either on process algebra, mathematical relations, or basic logical elements often defined syntactically. The approaches based on model checking are most often based on automata. The Mn-approach is in the latter tradition even though it borrows techniques also from rewrite systems.

1.6.2.1 Model checking

The term “model checking” was coined by Clarke et al. in the early 80’s [27]. The term refers to the technique to define both a model (by a program) and a specification (by a specification language) and to check in the model that the specification is satisfied. The common semantic base of the model and the specification is the automaton. The model and the specification are transformed into special kinds of automata (Büchi-automata) and the combined automaton is analyzed.

The model is normally a finite state automaton, which makes the model checking problem decidable. The specification is turned into a Büchi-automaton which is a finite automaton which takes infinite input. The model checking problem can be expressed as nonemptiness of the language accepted by the combined automaton.

We shall go in some more detail without being too formal, but for those who want a more thorough introduction to these matters we refer to [133].

An automaton has an input *alphabet* comprised of *symbols*. The automaton is in one of a finite set of *states*. The automaton starts from a state in an *initial set of states*. A *word* is a sequence of symbols. The input to the automaton is a word. The automaton will consume the first unconsumed symbol and depending on its current state it will perform a transition to a next state. Thus there is a set-valued function from the set of states and the alphabet to the powerset of states. This is a *non-deterministic* automaton. In a *deterministic* automaton the transition function yields only one state. A subset of the states is defined to be *accepting states*.

An automaton *accepts* a finite word if the automaton is in an accepting state when the last symbol of the input word has been consumed. A Büchi-automaton accepts an infinite word if the automaton enters accepting states infinitely many times during the infinite consumption of the word.

The set of words accepted by an automaton is called the *language* of the automaton.

When a word is accepted by an automaton there is a *run* consisting of a sequence of transitions from an initial state to an accepting state. We say that the accepting state is reachable from the initial state. In a Büchi-automaton acceptance means that an accepting state is both reachable from an initial state *and* reachable from itself (an accepting cycle). This shows that graph reachability is important in the analysis of automata.

Reachability in a finite graph is decidable. This is the reason why most practical techniques of model checking are confined to finite state automata to represent the model.

Checking whether a finite-state program P satisfies an LTL[1] formula φ can be done in time $O(|P| \cdot 2^{O(|\varphi|)})$ or space $O((|\varphi| + \log|P|)^2)$ [133]. $|P|$ and $|\varphi|$ represent the sizes of the program and the LTL specification respectively. We see that the complexity is very sensitive to the size of the specification. In practice the specification is much, much smaller than the program, which makes the complexity manageable after all.

An SDL system is not directly a finite state automaton. Since there may be any number of signals in the channels and since we have no scheduling algorithm given, the number of (complete) states is unbounded. The value ranges of variables in SDL processes are also infinite. Still the approximation of an SDL system by a system with bounded channel buffers and bounded ranges is a reasonable restriction. The available tools for SDL validation are all based on this restriction [40; 136; 75].

Our Mn-procedure does not assume that a system can be approximated by a finite state automaton. It applies fragmented reachability techniques such that each reachability graph is (hopefully) finite. If the assumption is made that buffers are bounded, the Mn-procedure collapses to the first phase of the general Mn-procedure as shown in Section 4.5.2 (p. 172) and Section 5.2.2.1 (p. 193).

The differences between the automata-oriented model checking approaches are firstly due to the difference in specification languages and secondly to algorithmic maturity.

CTL is a language defined by Clarke and Emerson and which has been shown to have fairly effective model checking algorithms. CTL and derived notations have been used to model check real hardware constructions with a considerable amount of synchronous communication [27; 134; 28; 43]. There is a slightly absurd competition about how many states the different methods handle since the state explosion problem is definitely the major drawback with these methods. The CTL-oriented methods claim to be able to cover problems where 10^{130} states are covered. This is hardly the number of states actually visited, but an estimate of the number of states in the total state space covered.

The significance of counting states is challenged by the most recent advances in verification of hybrid automata [67]. A hybrid automaton includes variables with continuous and infinite domains. Furthermore changes of the (complete) state is considered continuous according to a set of functions over time and transitions occur due to jump conditions on the variables. It is obvious that every domain covered will have an infinite number of states, but this is not the point since individual states are not visited. The analysis of the hybrid automata consists of solving equations. This approach seems very promising in fields of real-time continuous systems. An experimental tool UPPAAL has been made in a joint research project between universities in Aalborg (Denmark) and Uppsala (Sweden). They have chosen a fairly practical approach and have reported a number of successful case studies already. The validation of the Philips Audio-Control Protocol with bus-collision is the most comprehensive study done so far [6]. The resemblance to our work is that we also apply symbolic execution (e.g. for data variables) as a finite way to represent an infinite set.

1. LTL = Linear Time Logic

LTL is another family of specification languages including also PROMELA which is the language of SPIN – the experimental and practical tool built by Gerard Holzmann at Bell Labs, AT&T. This is probably the most well founded tool in the area of asynchronously and discretely communicating finite state machines, and it can show off a number of successful verification efforts [72; 76; 73; 75; 74].

One important idea for the success of SPIN is *Supertrace*, an algorithm for partial coverage of a state space based on hashing. In principle every state visited is given an address in the available storage address space. The address is assigned through a carefully chosen hashing algorithm. Whenever a state is encountered there is a need to find out whether it is visited before. The address corresponding to the state is calculated by hashing and the corresponding bit is checked in the storage. If it is not set, Supertrace knows that the state has not been visited and sets the bit and continues. If the bit is set already, Supertrace assumes that the state has been visited and discontinues the search along this execution branch. This latter assumption will sometimes fail when there are more states than addresses, but the coverage becomes random. By applying more than one analysis with different hashing functions, it is possible to show that the actual coverage can get very close to 1 even with address spaces down to 0.01 of the total space [75].

Supertrace is used in the commercial SDL validators from Telelogic and Verilog [40; 136].

The main differences between the model checking approach pioneered by Holzmann and our Mn-approach are that we do not assume a finite state model, and we do not restrict ourselves to execution from the initial state.

From a practical point of view, reachability from the initial state is not very robust. Since most of the design is done with only fairly local knowledge, the designers will not be able to overview whether the set of reachable states are affected by a change which they want to introduce. In fact whenever there has been a change in the system, the whole reachability analysis must be repeated.Contrary to this our Mn-approach is more robust as it considers the whole set of complete states and not only the set of reachable states.

This concludes our discussion on model checking. Their major drawback is that the analysis is normally not compositional meaning that earlier results cannot be applied in future analysis of other parts of the system or new versions of the system. In contrast our Mn-approach based on reducibility is compositional as shown in Section 4.1 (p. 143).

1.6.2.2 I/O relations

Even though automata are attractive as basic semantics building blocks, many proof theory scientists choose otherwise. In the study of communicating finite state machines also relations between the input sequences and the output sequences have been used as basic formal model [91].

In FOCUS this principle is called “stream processing functions” [15; 20]. The approach is a very general one based on formal proof theory rather than model checking. A *stream* is an infinite sequence of signals either input to or output from a process. The *process* is defined as a set of functions which processes these streams. Even though the processes

are not automata there is no problem to define automata processes in FOCUS through the use of simple templates. In this way SDL semantics can be expressed (in principle) in FOCUS [16; 17; 128; 68].

Time is introduced in FOCUS as special time ticks appearing as signals in the streams. In this way FOCUS is more expressive than SDL where no declarative reasoning can be done regarding time and duration.

FOCUS is a monolithic approach to system specification and as a method for system development its strategy is to define a series of refinement steps from a very abstract specification to a realizable one [18; 129; 20]. The refinement approach is well combined with compositionality which makes the approach attractive. The FOCUS approach to interface refinement has been adopted by the Mn-approach as shown in Section 4.2.1 (p. 147).

Another approach which has certain resemblance to our approach is the one by Jonsson [88; 89] where he uses the concept of *trace generators* in a compositional model of i/o systems expressive enough to capture the Brock-Ackerman anomaly. The compositionality makes it possible to achieve reductions comparable to ours, but the reduction is made through formal proofs and not an automatic procedure.

1.6.2.3 Rewrite systems

Rewrite systems are directed equations used to compute by repeatedly replacing subterms of a given formula with equal terms until the simplest form possible is obtained [37].

The execution of a process can be seen as the computation of a rewrite system. The transition table can be seen as the substitution rules. The consumption of an input signal and producing output signals can be seen as substituting a ground symbol prefixing a sequence by sequences of ground symbols appending sequences.

$$(S;eE;I;O) \longrightarrow (T;E;Iij;Ox)$$

Figure 15: Substitution rule

In Figure 15 (p. 33) we have given an example of a substitution rule where the external signal e is consumed producing the internal signals ij and external output x. E,I,O are variables. S and T are basic states. As we shall see in Section 2.1.3 (p. 44), this is identical to our notion of an unlabeled transition representing a possible execution step of the communicating process.

Therefore it should be no surprise that our Mn-procedure is influenced by the theory of rewrite systems. Our notion of confluence (see Section 2.4 (p. 51)) is very similar to confluence of rewrite systems. Our Mn-procedure has a concept of potential non-confluence patterns, and this is similar to critical pairs in determining rewrite confluence.

The search for confluence corresponds to determining Church-Rosser property of the transition system. Keller [92] shows that *commutativity* can be used to show Church-Rosser. His notion of commutativity implies trivially our notion of confluence and a

commutative system is determined confluent by the Mn-procedure in its simplest form (M0-execution, see Section 2.4.4 (p. 56)). Sethi [122] also makes the search for Church-Rosser a central point in his *replacement systems* used to optimize programs.

Our notion of progress (see Section 2.3 (p. 50)) corresponds to termination in rewrite systems and we advocate to apply techniques from rewrite systems for the determination of progress. Thus reducibility in Mn-terms (see Section 2.2 (p. 47)) is similar to convergence in rewrite systems. In fact reducibility in Mn terms corresponds reasonably well with *ground convergence* of the rewrite system derived from the process.

Ground convergence means that every *ground term* rewrites to a unique normal form. A ground term is a term consisting of only *ground symbols*. In our terms the ground terms are the signals.

1.6.2.4 Proof systems

There is an important division concerning verification. Are the proofs automatic, or do we need human intuition? Most formal approaches have a considerable portion of manual proof construction, but the picture is changing – rapidly.

The proof-oriented methods have (at least) two major drawbacks seen from a practitioner:

1. Manual proofs are tedious, time-consuming, error-prone and incomprehensible.
2. To find the invariants which are strong enough to make the proof work, requires experience which is beyond the normal competence of a software engineer. When the invariants are found, they are great to use, but finding them is a big hassle.

These negative opinions by the practitioner are partly due to reality and partly due to myths. There is no doubt that the perceived complexity of a formal proof often reflects the complexity of the problem or the solution, and any explanation failing to realize this will often miss important aspects of the problem. It is also obvious that practitioners should be encouraged to express invariants and to use them in their arguments for the correctness of their programs. The search for strong enough invariants often reveals new aspects of the problems. Still our opinion is that stronger emphasis on automatic techniques is a positive trend for future development of verifiable software.

Early formal theories for concurrency which were suitable for automation include Petri-nets [116; 93] which have been used for a number of verification purposes. Petri-nets led to trace theory [99] which became a research area of its own.

In the tradition of Hoare-logic we have VDM[87] which was an early attempt to apply verification in software engineering. Competent people made effective use of this technique, and tools have been developed, but it did not catch on as a general approach to software engineering. VDM has had influence on SDL as the formal semantics[79] of SDL is defined in MetaIV which is a variant of VDM.

In recent years general software for the assistance of theorem proving has emerged. The proof assistants are increasingly being used for more practical purposes [4; 100] and even though the theoretical complexity can be proved to be prohibitive, the actual complexity may be within the limits of modern computers. This is also very much the assumption behind our Mn-approach where the worst case is very bad, but where methodological reasoning indicates that the actual complexity should be manageable.

Since there are problem areas where general algorithms cannot be found, it is reasonable to assume that combined uses of proof assistants and manual proofs will emerge [126; 127]. The combination of the different automatic techniques combined with manual proofs of details or of certain generalizations becomes attractive [104].

This is very much in line with our Mn-approach where we encourage the combined use of different techniques rather than expecting the Mn-approach itself to solve all problems.

The reader is referred to [96] for an attempt to compare different approaches to the same experimental system. The survey gives some insight into the differences of the approaches, but the individual efforts are done by different people with differing competence in the specifications used such that the comparisons are not really very reliable.

1.6.3 The Mn-approach does not really address real-time

Even though SDL is being used extensively in the construction and design of real-time systems, SDL does not have means to describe real time other than by plain timers. Its dual specification language MSC does not have means to describe real-time either. This means that real-time reasoning in an SDL environment is difficult or impossible without applying extra information supplied in other formal or informal languages.

Approaches to real time are still quite formal with few success stories from real life, but research is progressing. We have already mentioned the theory of hybrid automata [53; 67] which seems very promising, and FOCUS [20] which also offers compositional attitudes to real-time reasoning. Furthermore much research has been done in the ProCos project [114; 38; 65] to define a “duration calculus”.

The Mn-approach addresses concurrency and asynchrony, but we find that reducibility preserving real-time properties such as minimal response times, is definitely more difficult (see Section 3.7.4.6 (p. 126)).

1.6.4 Mn-approach is integrated with design

The main target for the Mn-approach is to contribute to the design of reactive systems. We have named our design approach “confluent design” to emphasize the aim to create confluent units which may be functionally reduced.

Traditionally verification methods have concentrated on verification and not on the design of the system. Still experience from verification led to such slogans as “goto-less programming” referring to the problems of verifying programs containing goto-clauses. Structured programming was the answer to easier verification[31; 30]. The interleaving of specifications and program known from approaches with pre- and post-conditions based on Hoare-logic [69] could also be considered a motivation for verification-oriented design.

In the application of SDL the integration of verification and design is encouraged. In the appendix to the Recommendation in 1993 [80] verification of SDL systems is considered together with their design and implementation. In the tutorial collection for the SDL Forum 1995 [115] advocates that validation should be performed as an integral part of every software development activity. The Telelogic SOMT method[131] also intro-

duces tool support for inter-notational links (“imp-links”) which describes mappings which should be used during consistency checking. In a Supplement to the most recent Recommendation [84], validation is again highlighted. SDL as a formal language is emphasized in [42].

The CleanRoom method [39; 110] is based on small verifiable design steps. The steps are not necessarily automatically verified, and also walkthroughs are accepted as verification, but it emphasizes the integration of design and verification.

Regardless of these attempts in practice there is often a clear distinction in time between designing a system (program) and verifying it. Commercial companies may have disjoint groups of people to perform verification and design. The distinction is partly due to division of labor, and partly due to difference in competence. Our Mn-approach aims to let the designer be able to understand and to undertake some systematic validation effort during his design. The goal is to provide a smooth transition from tentative complexity evaluation through automatic analysis to full reducibility analysis.

The problem with methods applying informal notations such as OOA [29], OMT [118] etc. is that the consistency between different descriptions is not verifiable. We expect that UML [119] may lead to more precise semantics, but still SDL offers a lot more in terms of formalisms.

The method brought forth by SISU [62] presented on electronic form in [12] is a good framework for adaptation of the Mn-approach. The “distillery” conceptual framework presented in Section 5.1.3.1 (p. 180) provides the background for more verification-oriented design. The Mn-approach not only provides an answer to whether different parts of the design are consistent, but also indicates complexity and offers guidelines for improved verifiable confluent designs.

1.6.5 Comparison summary

We may summarize the relations between the Mn-approach and other approaches by the following table which tries to present in short the placement of the Mn-approach and the reasons for why it is like it is.

Table 1: The Mn-approach and other approaches

Mn-feature	similar to	different from	comments
asynchronous communication	FOCUS, SPIN	CSP, CCS	asynchrony is often perceived as better for the designer
monolithic	FOCUS, Z	most formal verification techniques	one paradigm is normally enough for a practitioner
compositional	FOCUS, Jonsson	CTL	reuse of analysis is preferable

Table 1: The Mn-approach and other approaches

Mn-feature	similar to	different from	comments
check also unreachable cases	Testing, rewrite systems	SPIN etc.	reachability from initial state is not robust
process as semantic base	automata-oriented techniques	proof theory	the semantic base is conceivable for practitioners
unbounded state space	hybrid automata, proof theory	model checking	often the finitude of the program state space is assumed in model checking
automatic	SPIN, etc. rewrite systems	proof theory	manual proofs are error-prone and time-consuming
integrated	SISU integrated methodology (TIMe), SOMT	OOA, OMT	Mn is formal enough for verification and pragmatic enough for designers

1.7 Reader's guide to the thesis

The thesis is built up as follows. In Section 1. (p. 1) we have given an overview of the thesis with an executive and a technical summary. We have also given motivation and background for our work.

Section 2. (p. 41) contains the theoretical foundations for our "Mn-approach". We present the theory in a context which is much simpler than we find in real systems, but which is complicated enough to exhibit most of the difficulties.

In Section 3. (p. 83) we see how the theory can be made more applicable by generalizing it to more realistic situations. SDL features such as multiple channels and multiple processes, non-determinism and timers, save and priority input, procedures and object orientation are covered. The Mn-approach is modified to cope with these areas.

Section 4. (p. 143) contains material which shows how the Mn-approach can be used for formal reasoning, while Section 5. (p. 177) contains a methodology "Confluent design" where the Mn-approach is central. The aim is to show that the Mn-approach can be used not only to reduce system parts, but also to give evaluations of the complexity of a system. The Mn-approach is seen to be of practical use even if it is unable to produce a reduction. We report from a rudimentary industrial case study.

During the presentation of the Mn-approach a number of examples are given, but we want to give an indication of how the Mn-approach works for a medium size toy example. Our choice is the RPC-Memory specification problem which was suggested by

Leslie Lamport and Manfred Broy before a conference at Dagstuhl, Germany in September 1995. In Section 6. (p. 229) we apply the Mn-approach to this problem and show successes and shortcomings of the Mn-approach.

Section 7. (p. 269) contains the conclusions and indications for how the Mn-approach may be improved by further research.

The references to literature follow in Section 8. (p. 275)

In Section 9. (p. 283) we give a summary of the extensions to SDL which we have used in this thesis in order to make SDL suitable for our purposes.

Appended to the thesis is a list of all the figures.

1.8 Acknowledgments

Since this thesis falls in between formal theory and practical engineering, we have had great benefit from a number of sources, and a lot of people have contributed positively to the progress of the work.

The take-off of our doctoral studies would have been more difficult without the explicit support of our former advisor, professor Kristen Nygård. Professor Sverre Spurkeland opened our eyes to symbolic verification and constituted a valuable discussion partner in the middle phases of the work. My advisor professor Dag Belsnes has given increasingly important criticisms to the work as the material has matured, especially during the last years.

Concerning the more theoretical parts of the work, the contacts with Ketil Stølen, Ph.D. has been of utmost importance. The visits to Lehrstuhl Broy at the Technical University of Munich were extremely inspiring and valuable. The visits supported our believes that in the future formal theory and practical engineering should increase their contacts.

We have had the great pleasure to participate in two very interesting workshops concerning the theoretical foundations of verifying concurrent systems. The first workshop was organized by Graham Birtwistle in Banff in the Rocky Mountains of Canada in the fall of 1994. The workshop had the topic “Logics for concurrency. Structure versus Automata” and was chaired by Faron Moller. The Rocky Mountains environment where the participants joined the excellent lecturers hiking in the wilderness and swimming in the ice cold glacier lakes, was the perfect setting for learning and for making good professional contacts.

The second workshop took place in Århus, Denmark in the fall of 1996. BRICS Autumn School in Verification had the topic “Theorem Proving and Model Checking” and some of the best experts in the field gave a series of very inspiring lectures. The Danish breweries also contributed to the making of contacts.

From the practical engineering side, the contacts made through the Norwegian SISU project, where we have participated in parallel with the doctoral studies, have been the most important. The parallel work on an integrated methodology in SISU (now given the name *TIMe*) has been inspiring and established a reference methodology for our suggestions in this thesis. In this work Geir Melby, project leader of SISU, professor Rolv Bræk, scientists Birger Møller-Pedersen and Richard Sanders have been a fantastic team to work with.

We would also like to thank the SISU companies in general for providing practical experience which has been the practical foundations of my work. In particular we are grateful for the opportunity to perform a rudimentary case study on a real system of the Mn-approach, which was granted by Siemens AS, Defense Systems. The contacts with Siemens and their engineers Ole Henrik Støren, Astrid Nyeng and Svanhild Gundersen were of great importance.

Thanks to Piet Hein for his encouraging poems, his “grooks” which we have used to introduce the chapters of the dissertation [66].

Finally our family must be thanked. We thank our three children who each in their own way made it clear that doctoral studies are not the most important mission in life, and the wife who has tried to keep the family running while the doctoral student was sitting behind his desk.

Oslo, March 1997

Øystein Haugen

1 Introduction

Acknowledgments

2 The Basic Mn-procedure

Problems

Problems worthy
of attack
prove their worth
by hitting back

2. The Basic Mn-procedure

In this chapter we present the Mn-procedure. The Mn-procedure is a procedure which aims to determine whether a given system is reducible to a simpler process. In this chapter we start by analyzing SDL systems which have several restrictions and as such can be characterized as basic systems.

The restrictions that we put on the SDL systems analyzed in this chapter are:

1. The external input sequence is finite.
2. The system consists of one process only.
3. The system contains one external input channel, one internal channel, and one external output channel.
4. The process is deterministic, meaning that given a basic state and a signal only one transition is possible. The transition contains no decisions leading to different nextstates.
5. There are no data variables in the process.
6. There is no save (no explicit permutation of signals).
7. There are no timers.

The restrictions and their relaxation will be discussed in Section 3. (p. 83). The restrictions make the presentation of the technique simpler and as we shall see, the relaxation of the restrictions still do not jeopardize the main results of the technique. On the contrary, relaxing the restrictions makes the technique even more applicable in industrial contexts.

2.1 Basic concepts

2.1.1 Notation

2.1.1.1 Basic notation

f(x)	means that function *f* is applied to argument *x*.
$f : D \rightarrow R$	is the signature of function *f* with domain *D* and range *R*.
ft(x)	means the first element of the sequence *x*.
rt(x)	means the rest of *x* after the first element has been removed
tuples	are denoted (x,y,z,...) or <x,y,z,...> or [x,y,z,...]. There is no semantic difference between the notations for tuples. Sometimes we use semicolon as separator, and sometimes comma. The difference is again only to distinguish between different kinds of lists.
$\wp A$	denotes the powerset of *A*.
A^*	denotes the set of finite sequences of elements of A.
A^∞	denotes the set of infinite sequences of elements of A.
A^ω	denotes the set of finite and infinite sequences of elements of A.
β_A	denotes a mathematical variable of type A* . In general we use greek letters to denote sequences of signals which is not known. The beta symbol is just one example of a greek letter.
ø	denotes the empty sequence
$\underline{x}$	denotes the system state where x is the basic state and the internal queues are empty.
$v \approx v'$	means for two sequences v and v' that either v is a prefix of v' or vice versa or they are equal. We say that v and v' are prefix related.
primes	Primes are used to distinguish symbols which are similar, but different
suffixes	Suffixes are used to distinguish similar items

2.1.1.2 Process definitions

We will give our process definitions in SDL-92 with some proprietary extensions summarized in Section 9. (p. 283).

2.1.2 Basic model

The concern of this thesis is the analysis of SDL-like systems. SDL systems consist of *components* which communicate via *channels* (or signalroutes) through discrete tokens of information, *signals*.

A typical example of the basic model is shown in Figure 16 (p. 43).

In Figure 16 (p. 43) the channels are named c1, c2, c3, c4, c5. For each channel there are sets of legal signals. They are named e1,e2, i1, i2, e3. The channels are divided into three sets: the external input channels, the internal channels and the external output channels. The *external input channels* receive input signals from the environment and deliver them into the system under analysis. Conversely the *external output channels* deliver signals from the interior of the system to the environment. The *internal channels*

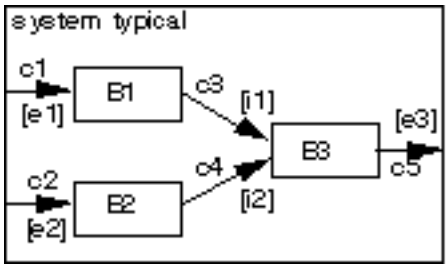


Figure 16: Typical structure of an SDL system

deliver signals from the interior of the system to the interior of the system. Observed from the outside of the system, the internal channels are invisible and their signals are not observable.

The communication is asynchronous which means that the output (sending) and the input (consumption) of a signal is not coincident in time. The exact duration of operations is not usually within our area of concern.

Each component may itself have an architecture like a system with channels and subcomponents. Such components are called *blocks*.

A component may instead of being a block be a *process* which is an atomic processing unit in the form of a *finite state machine*. One of our goals of the analysis is to be able to view a block as a process. An SDL process has an input port which is a signal buffer. There is only one input port for each SDL process even when there are more than one input channels. The single input port represents the fact that only one signal can be consumed at the time. In our basic model we shall assume that every input channel has a buffer and that the selection of which signal to consume happens just before the transition takes place. The decision of which channel to consume from is non-deterministic and fair, meaning that no signal should stay forever in the buffer without being consumed. There is a question whether this fairness is actually reflected in the formal definition of SDL[79; 68], but in practice this fairness is wanted and sometimes needed.

Whenever two channels merge into one, there is implicitly a *fair merge component* which makes sure that the merge is also fair.

Every *transition* is considered *atomic*. A transition is the actions of a process following the consumption of a signal until another basic state is reached. That a transition is atomic means that the transition will not stop due to some interruption. The process internal state of variable values cannot change due to other actions than those in the transition.

In a system or a block, the sub-components execute in true parallel, but semantically this will not be different from considering that only one transition executes at the time (*interleaving semantics*). Since the processes have different clocks and are independent there is no way to observe whether individual transitions execute in sequence or in parallel. Consider our typical example in Figure 16 (p. 43). If B1 and B2 execute in parallel, we still cannot know within the framework of SDL the relative durations of their transitions. Therefore we cannot know which signal i1 or i2 will be sent first. Therefore for the sake of analyzing B3 we must consider both the situation where we get i1 on c3 before i2 on c4, and the opposite. In fact we must consider all merges of signals of c3 with signals of c4.

Our basic model assumes that transitions eligible for execution are the ones that consume the first signal on one of the input channels.

That it is necessary to have a model with multiple input channels when a transition is to be considered atomic, can easily be seen from the following scenario. Let B1 have a transition which produces two output signals: i1,i1. B2 executes a transition producing one i2. If there was one input port of B3 in our basic model, the possible sequences of that input port would be either i1,i1,i2 or i2,i1,i1 since the transitions by assumption were atomic. If there are two input ports (input buffers) of B3, it may also consume the signals in this order: i1,i2,i1. This corresponds well with a model where the transitions are not atomic seen from the system level such that the signals may arrive in any order to the single input port. We conclude that our basic model of multiple input ports and atomic transitions corresponds well with the SDL model of a single input port and merging of the individual inputs from the different sources.

2.1.3 Basic definitions

Here we shall present the basic concepts in a more formal framework. We define a concept of *process* which is slightly more general than the one presented informally in Section 2.1.2 (p. 42). The SDL process corresponds closely to our concept *CFSM* (Communicating Finite State Machine).

2.1.3.1 A process and related concepts

A *process* is a tuple $\langle S;C;Z;T\rangle$ where

- S a finite set of *basic states*
- C the *alphabet* which represents possible values of all the channels of the process. Each individual value is of a *tuple* where each entry corresponds to a channel. Each entry may be a sequence of *symbols*. Each symbol may in principle be a sequence of *signals*.
- Z a set of *initial complete states* from where to begin the execution ($Z \subseteq K$)
- T the *transition table*. $T: S \times A \rightarrow K$ where the S designates the basic state and the A the input. The result is a complete state.

The auxiliary sets used above are defined by:

- K the set of *complete states* $K = S \times C$.
- A the *input alphabet*. $A \subset C$. The input alphabet consists of the elements of C which have exactly one symbol on one input channel and nothing on the others. Normally we denote an element of A by the name of the signal.

These capital letters will refer to the concepts of a process throughout the thesis, i.e. whenever we talk generally of a process P, K means the set of its complete states and A its input alphabet. If confusion may arise we subscript the notation by the process identifier e.g. K_P

2.1.3.2 A Communicating Finite State Machine

A *communicating finite state machine (CFSM)* is a process (cf. Section 2.1.3.1 (p. 44)) where

C the alphabet $C = E^* \times I^* \times O^*$ where:

- E: the external input alphabet, a finite set of signals input to the system from its environment;
- I: the internal alphabet, a finite set of signals for a channel within the system itself
- O: the external output alphabet, a finite set of signals for transmitting signals out from the system to the environment.

Z the set of initial complete states is based on one initial basic state z
$Z = \{(z;\theta_E;\emptyset;\emptyset) \in K \mid (\theta_E \in E^*)\}$ which we denote $\underline{z}$.

The reader should notice the notation used to denote a complete state because this general notation is used throughout this thesis. A complete state is a tuple of a basic state, external input channels, internal channels and external output channels. Each of these parts are separated by a semicolon. In the general case which is treated in Section 3.2 (p. 87), each of the sections may consists of a set of elements. They will then be separated by comma. Notice that we make no special syntactic distinction of the alphabet element within the complete state. The alphabet element is the tail of the complete state where the first basic state element is removed.

A the input alphabet is $\{(e, i, o) \in C \mid e \in E \wedge i = \emptyset \vee i \in I \wedge e = \emptyset\}$ which means that the CFSM reacts to either one external or one internal signal.

2.1.3.3 A transition system

The execution of a process $\langle S;C;Z;T \rangle$ can be interpreted relative to the transition system $\langle K;A;\longrightarrow;Z \rangle$ where K is the set of (complete) states, A the input signal alphabet, $\longrightarrow$ the transition relation derived from T, and Z the set of initial (complete) states.

The transition table T of the process is used to transform the complete states. Each complete state can be seen as the root of an execution tree. An execution of a transition means to consume the first signal of one of the non-empty input channels and to perform the transition indicated in the transition table. The result will be a new complete state.

Assume that the process is a CFSM. If T is the name of the transition table, T_i denotes the complete state reached when consuming the first internal signal and T_e denotes the complete state reached when consuming the first external input signal. More formally in Figure 17 (p. 46):

The plus symbol (“+”) in Figure 17 (p. 46) refers to a simple concatenation operator where the signal sequences are concatenated. The state is not affected. The trivial definition is not given here.

The labelled transition relation is a simple application of T_e and T_i.

$$T_e : K \rightarrow K$$
$$T_e(s;e;i;o) \equiv (rt(e);i;o) + T(s;ft(e);ø;ø)$$
where $e \neq ø$

$$T_i : K \rightarrow K$$
$$T_i(s;e;i;o) \equiv (e;rt(i);o) + T(s;ø;ft(i);ø)$$
where $i \neq ø$

Figure 17: Transition function

$$\xrightarrow{c} : K \times A \times K \rightarrow Bool$$

$$(s_1;e_1;i_1;o_1) \xrightarrow{c} (s_2;e_2;i_2;o_2)$$

when $(T_e(s_1;e_1;i_1;o_1) = (s_2;e_2;i_2;o_2)) \wedge (c = ft(e_1))$
or $(T_i(s_1;e_1;i_1;o_1) = (s_2;e_2;i_2;o_2)) \wedge (c = ft(i_1))$

Figure 18: Labelled transition relation

The reader should notice that in Figure 18 (p. 46) there are two right arrows which serve two different purposes. One purpose is to serve as a symbol for the labelled transition relation, and the second purpose is to separate the domain from the range in the function signature.

$$\longrightarrow : K \times K \rightarrow Bool$$
$$k_1 \longrightarrow k_2 \equiv \left(\exists a \in A \bullet k_1 \xrightarrow{a} k_2 \right)$$

Figure 19: Unlabeled transition relation

It is obvious how these definitions in Figure 18 (p. 46) and Figure 19 (p. 46) generalize to processes from CFSMs. The formulation is slightly more intricate and adds little to the understanding.

2.1.3.4 Execution graph

The *execution graph* is the execution tree of x denoted by G(x). The execution graph is a set valued function which is based on the unlabeled transition relation as defined in Figure 20 (p. 47). The nodes of G(x) is denoted by H(x).

2.1.3.5 Leaves

The *leaves* of the execution tree are the states which have no input. The set of leaves of the tree with root x is denoted by L(x).

$$G : K \rightarrow \{(u, v) \in K \times K \mid u \longrightarrow v\}$$
$$H : K \rightarrow K$$

For any $x \in K$, H(x) and G(x) are the least sets such that:
$x \in H(x)$ **and**
$(y \in H(x) \wedge y \longrightarrow k) \Rightarrow ((y, k) \in G(x) \wedge k \in H(x))$

Figure 20: Execution graph of x, G(x) with nodes H(x)

$$L : K \rightarrow \wp K$$

$$L(x) \equiv \{k \in H(x) \mid G(x) = \varnothing\}$$

Figure 21: Leaves of x

If the set of leaves contains only one element, the execution is *deterministic* from the given state. If the execution from all complete states is deterministic, the process is deterministic.

2.1.3.6 A stable state

A stable state is a complete state where the internal queues are empty.

The concept of stable states differ from “leaves” since leaves are stable states with no external input signals left, either.

Stabilization is the execution of internal signals only such that a stable state is reached.

2.2 Reducibility

The idea behind establishing reducibility is obviously to eliminate aspects of the system under analysis that is irrelevant for the kind of analysis which we currently are doing. In our thesis we concentrate on the properties of processes as signal consuming and signal emitting entities.

From this point of view, to reduce a process would mean to eliminate all internal signalling such that only the external signalling relations are present. It is our hope that such a reduction will yield a new process description which has some interesting features:

1. *Improved overview* capabilities through the removal of internal signals. Understanding the system may become easier. The reduction may also reveal aspects of the system which was more hidden in the complete description.
2. Simpler descriptions such that *other methods may be applicable* to larger systems. Often other techniques such as reachability analysis and formal verification cannot handle large systems. If parts of the systems are reduced, larger systems can be manageable.
3. The *behavior of the reduced process is identical* to the original one when seen from the outside.

Furthermore, the procedure to establish reducibility may have as a spin-off effect that complexities of the original process are revealed. The problems of the establishing of reducibility may even prove to serve as a complexity measure (see Section 5.2.2 (p. 193)).

2.2.1 What is reducibility?

Kwong showed already in [94] that with his definition of reducibility which corresponds closely to ours, a number of correctness criteria are preserved over reductions. For our purposes the preservation of different variants of deadlock-freedom and homing are of special interest. Homing means that there is a set of complete states which can be reached from any other complete state. Homing can express that the system may always return to an idling situation.

If subsystems can be reduced, e.g. to a single finite state machine, the analysis by standard reachability techniques can be feasible for larger enclosing systems. We give in this chapter a reduction algorithm that yields a reduced system in terms of Kwong [94].

*2.2.1.1 **Defining** Reducibility*

A process P1 is reducible to a process P2 if the following criteria hold:

P1=<S1;C1;Z1;T1> *reduces to* P2=<S2;C2;Z2;T2> **iff**

$S2 \subseteq S1$

$C2 = (E^* \times \varnothing \times O^*) \subset C1$

where E and O are the sets of external input and output of P1

$Z2 = Z1$ (which means that the set of initial states are equal)

$\forall(z \in Z1), \forall(\theta \in E^*) \bullet (L_{P1}(z;\theta;\varnothing;\varnothing) = L_{P2}(z;\theta;\varnothing;\varnothing))$

Figure 22: Reducible process

This definition of reducibility (Figure 22 (p. 48)) corresponds closely to that of Kwong. See Section 2.5.3 (p. 75) for the detailed correspondence between Kwong's definition and ours.

We notice that the definition says nothing about the input alphabets and the transition tables. We also notice that a reduced process is *observationally equivalent* [102; 103] to the process from which it is reduced.

2.2.2 The reduction algorithm

Our reduction algorithm is based on the process being reduced is *progressive* and *confluent*. These two concept will be defined in Section 2.3.1 (p. 50) and Section 2.4.1 (p. 51), but here we just want to present the simple reduction algorithm. The proof of the correctness of the algorithm follows after the presentations of progress and confluence in Section 2.5.1 (p. 73).

1. Let P1=<S1;C1;Z1;T1> be the original process and P2=<S2;C2;Z2;T2> the reduced one. The set of external input signals of C1 is E and the set of external output signals of C1 is O. We will build P2 from executing P1.
2. Initialize S2:=Z2:=Z1 and Q:=Ø. Q will hold the states of S covered so far.
3. Find $p \in S2, p \notin Q$
4. For all $e \in E$ do

   4.1 Find $L_{P1}(p;e;ø;ø)$[1] since P is progressive. That it is progressive means here that we are certain to find the one leaf node just by stabilization (see Section 2.1.3.6 (p. 47)).

   4.2 Let $T2(p;e;ø;ø) := L_{P1}(p;e;ø;ø)$[2]

   4.3 Include q in S2 **where** $(q;ø;ø;o) \in L_{P1}(p;e;ø;ø)$

   4.4 Next e
5. Include p in Q.
6. Repeat from point 3 until such a p cannot be found (i.e. S2=Q).

2.2.3 The example process D

Throughout our presentation of our Mn-approach in this chapter we shall use an example process D shown in Figure 23 (p. 49) and Figure 24 (p. 50).

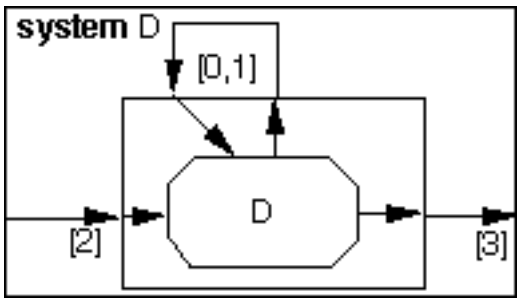


Figure 23: Structure of example process D

Process D is a CFSM with one external input signal {2}, two internal signals {0,1} and one external output signal {3}. It is not immediately obvious that process D is reducible. But given that it is reducible it is simple to calculate by the algorithm in Section 2.2.2 (p. 48) that the reduced process is process D'. Both the original process D and the reduction D' are given in Figure 24 (p. 50).

1. When the CFSM is deterministic there is only one element in $L_P(p;e;ø;ø)$.
2. The type of L is generally a set of complete states, but in our simplest case the set of leaves contains exactly one element and therefore the type of L can be considered K.

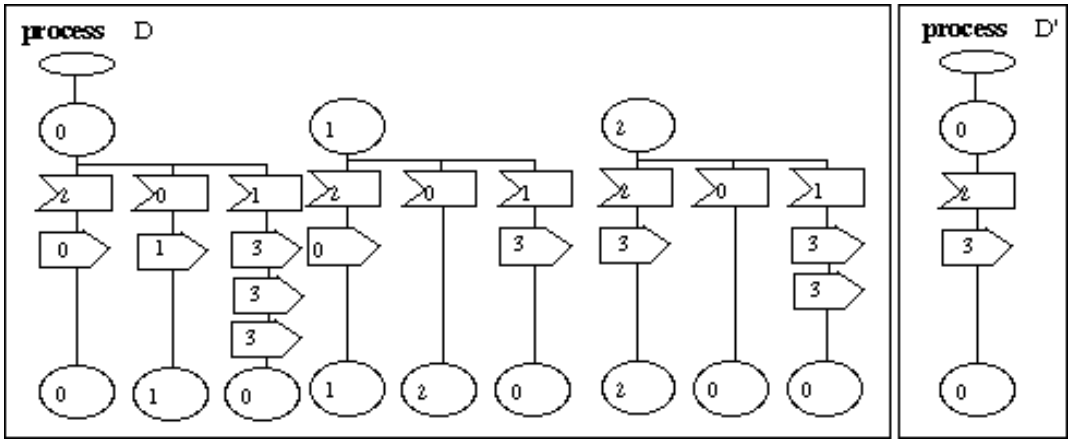


Figure 24: A reducible process

2.3 Progress

2.3.1 What is progress?

At this point in the thesis we define progress mainly as termination of the execution of a finite input. We may also express this by saying that for any external input signal, eventually it will be consumed and all internal derivatives of its execution (recursively) will be consumed until only external output is the result of the original (external) input.

More precisely we define that progress means that the execution graph of any complete state with finite input stream should be finite and cycle free.

Let P be a process <S;C;Z;T>
$Pr(P) \equiv (\forall(x \in K) \bullet G(x)$ is finite and cycle free)

Figure 25: Progress

We notice that our definition in Figure 25 (p. 50) says nothing about situations where the external input stream is infinite since we have restricted ourselves only to complete states where the external input is finite. The generalization to infinite external input streams will be covered in Section 3.1 (p. 84).

Progress is a desirable property by itself in most systems, but here progress is also a pre-requisite for the Mn-procedure for determining confluence to be presented in Section 2.4 (p. 51), and for the reduction algorithm presented in Section 2.2 (p. 47).

2.3.2 Progress of the example process D

In order to prove reducibility we must show that the process D in Figure 24 (p. 50) has progress i.e. that the stabilization of any instable state is a finite path.

This is simple in this case as process D adheres to the signal ordering principle introduced in Section 2.6.4.1 (p. 80). All transitions that consume signal “2”, produce either “0”,”1” or “3”. All transitions consuming “0” produce “1” or “3”, and finally all transitions consuming “1” produce only “3”. Thus we have a signal order “2”->“0”->“1”->“3” and all stabilizations will terminate.

2.4 Confluence

2.4.1 What is confluence?

Confluence is that race conditions between internal and external signals cannot affect the final result. A *race condition* is when the signals of several channels race to be consumed first. If we can reach a situation where the process can choose between consuming an internal signal or consuming an external signal, and the choice is significant for the final outcome, then the process is not confluent.

2.4.1.1 A confluent state

A *confluent state* is a complete state x of an CFSM where the predicate F(x) given in Figure 26 (p. 51) holds.

$$F : K \rightarrow Bool$$
$$F(s;e;i;o) \equiv (e \neq \emptyset \wedge i \neq \emptyset \Rightarrow (L(T_e(s;e;i;o)) = L(T_i(s;e;i;o))))$$

Figure 26: Confluent state

The final result of all the different feasible execution branches are the same. It follows from Figure 26 (p. 51) that a complete state where some input queue is empty is confluent.

A simple consequence is that the set of leaves of the root node itself is also equal to the set of leaves of any of its subtrees since they are all equal.

$$F(x) \Rightarrow L(T_e(x)) = L(T_i(x)) = L(x)$$

Figure 27: Consequence: leaves of root node

A set of states is confluent if all the states in the set are confluent.

Confluence in our terms corresponds to Church-Rosser property [122; 92] of the associated transition system representing the execution of the process.

2.4.1.2 A confluent CFSM

A *confluent* CFSM is a CFSM where all reachable complete states are confluent. In more formal terms we may describe a confluent process by Figure 28 (p. 52):

$$F(\langle S;C;Z;T\rangle) \equiv \forall z \in Z, \forall x \in H(z) \bullet F(x)$$

Figure 28: Confluent CFSM

The set of reachable complete states $H(z)$ cannot be determined in general [48] such that we will normally concentrate on assessing confluence of the whole set of complete states K. We define this as *absolute confluence* of the CFSM.

$$F(\langle S;C;Z;T\rangle) \equiv \forall k \in K, \forall x \in H(k) \bullet F(x)$$

Figure 29: Absolute confluence

We have used the leaf function L in our definitions of confluence. In this chapter we concentrate on situations where there is no explicit non-determinism. This means that the only factor to make the set of leaves contain more than one element is the race condition between external and internal channels. Thus if the process is confluent, it is also deterministic meaning that the set of leaves contains only one element. In Section 3.5 (p. 97) we shall introduce explicit non-determinism and that is the reason why our definitions of confluence is more general than the restricted case covered in this chapter.

2.4.2 Determining confluence

We shall now present an approach which may determine confluence. We call it the *Mn* approach. The approach is a procedure which is such that if it concluded confluence, this is definitely the case. Conversely if the Mn-procedure concludes “non-confluence”, a tentative non-confluent state is suggested as a counterexample of confluence, but the procedure gives no support for establishing whether the proposed non-confluent state is actually reachable. We have then established that the process is not absolutely confluent, but it may still be confluent.

As we shall see in Section 2.4.7 (p. 69), there is also a chance that the Mn-procedure may not terminate even when the system under analysis is progressive and confluent.

Thus the procedure has two major, potential drawbacks:

1. The Mn-procedure may not terminate Pragmatic improvements of the procedure to make it terminate is discussed in Section 2.6.5 (p. 81)
2. The Mn-procedure may find non-confluence patterns which cannot be reached.

The Mn-procedure aims to eliminate the possibility of a non-confluence pattern. A non-confluence patterns is a situation where in a state S there are an internal and an external signal which may execute, and their execution order makes a difference in the final result. In other words we try to find a complete state in K, say x such that $F(x)$ does not hold.

2.4.3 The non-confluence pattern examined

Here we show that it is sufficient to find only quite restricted non-confluence patterns. It suffices to find a *minimal non-confluence pattern* where:

1. There are only confluent states in their subtrees. This means that the non-confluence shows up exactly at the root state.

2. The sequence of external signals is only one element long.

We shall approach this result gradually. Firstly we define the *least non-confluence pattern* formally in Figure 30 (p. 53):

$$N(y) \equiv \neg F(y) \wedge (\forall r \in H(y) \setminus y \bullet F(r))$$

Figure 30: Least non-confluence

We want to show that if we have progress and a non-confluent state, this implies the existence of a least non-confluence pattern in the graph of the non-confluent state.

2.4.3.1 Proof: progress and non-confluence implies a least non-confluence pattern

We want to prove that progress and non-confluence is sufficient for the existence of a least non-confluence pattern. Formally we want to show the statement in Figure 31 (p. 53)

$$\neg F(x) \wedge (G(x) \text{ is finite and cyclefree}) \Rightarrow \exists y \in H(x) \bullet N(y)$$

Figure 31: Existence of a least non-confluence pattern

This is quite simple to see. Assume that we have the complete graph $G(x)$. Consider $T_i(x)$ and $T_e(x)$. If they are not both confluent, pick one which has a non-confluent state and repeat the procedure from that state. Since we assume that $G(x)$ is finite and cyclefree somewhere down the tree we will reach a leaf and leaves are always confluent since their internal queue is empty. Thus we will find a non-confluent root where all subtrees have only confluent states.

2.4.3.2 Necessity of progress for the restricted non-confluence patterns

Example process K in Figure 32 (p. 53) shows that an internal livelock will make an execution tree where there is a branch which consists of only non-confluent states infinitely. It is impossible to find a non-confluent state with only confluent states in its graph.

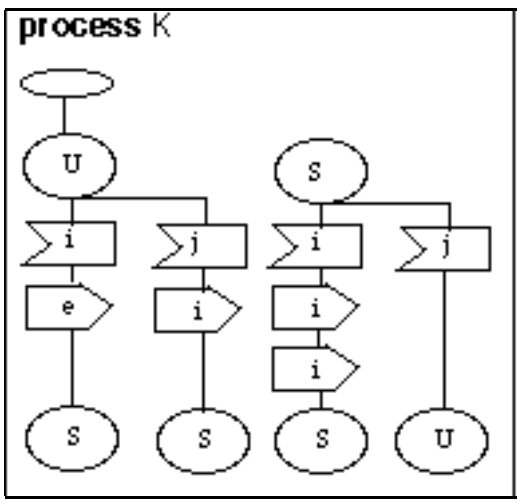


Figure 32: Necessity of progress for least confluence pattern

The process K has external alphabet {j} and internal alphabet {i}.

We show the beginning of an execution graph in Figure 33 (p. 54).

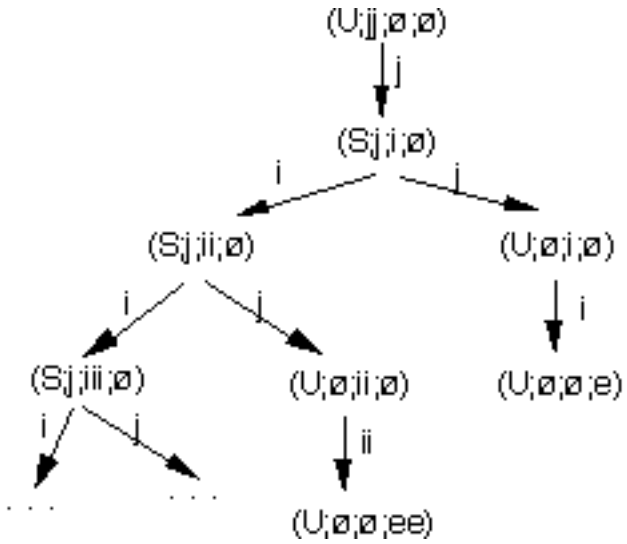


Figure 33: Selected execution of process K

We observe easily that there is a livelock since the process reproduces (duplicates) the internal signal i and there is not necessarily progress. We find no non-confluent state with only confluent states below, and consequently no least non-confluence pattern.

2.4.3.3 The symmetric non-confluence pattern

Now we assume N(y). Thus we have a complete state y where both the subtrees contain only confluent states. This means that: $L(T_e(T_i(y))) = L(T_i(y))$ and $L(T_i(T_e(y))) = L(T_e(y))$ according to consequence of Figure 27 (p. 51). Since we have that N(y) (defined in Figure 30 (p. 53)), we have that $L(T_e(y)) \neq L(T_i(y))$ which we may transform by substitution to the formula given in Figure 34 (p. 54)

$$N(y) \Rightarrow (L(T_e(T_i(y))) \neq L(T_i(T_e(y))))$$

Figure 34: Symmetric non-confluence

2.4.3.4 The minimal non-confluence pattern

We shall now show that we need only consider non-confluence patterns where the sequence of external signals is of length one only. Such special symmetric non-confluence patterns we shall call *minimal*.

The reason for that is that the external input channel will not be appended during the execution. To see that a minimal confluence pattern is sufficient, assume the converse that we shall have to need more than one signal on the external input channel. The situation is presented in Figure 35 (p. 55).

Observe the resulting stabilized states on the bottom of Figure 35 (p. 55). If these two states are exactly equal, the root state has to be confluent (contrary to the assumption).

This follows from the assumption that the first level subroots are confluent by the following:

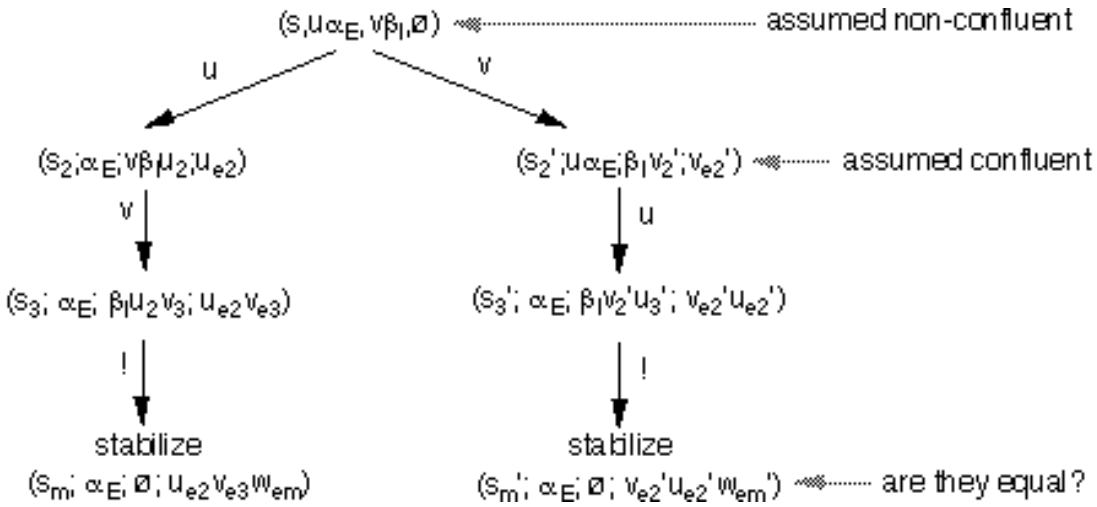


Figure 35: Minimal non-confluence pattern

$L(s_2;\alpha_E;v\beta_I u_2;u_{e2}) = L(s_m;\alpha_E;ø;u_{e2}v_{e3}w_{em})$ since the subtree is confluent

$= L(s_m';\alpha_E;ø;v_{e2}'u_{e2}'w_{em}')$ since the set of leaf states are assumed equal

$= L(s_2';u\alpha_E;\beta_I v_2';v_{e2}')$ since the subtree is confluent.

If the subtrees have equal sets of leaves for any pair of signals, the root must be confluent.

Since our assumption was that the root is *not* confluent, the two states are *not* equal which means that no value of α_E should make them equal.This will also have to include the empty sequence ø. Thus the non-confluence will show up also for α_E = ø which leaves the original external input sequence to only the single signal u.

This proves that the non-confluence pattern needs only a single external input signal. In other words if there is a symmetric non-confluence pattern, there will also be a minimal non-confluence pattern.

2.4.3.5 Summary

We assume that a CFSM is progressive.

If a CFSM has no minimal non-confluence patterns in its set of complete states, there is no symmetric non-confluence pattern either.

If there is no symmetric non-confluence pattern, there is no least non-confluence pattern.

If there is no least non-confluence pattern, there is no non-confluence pattern.

If there is no non-confluence pattern, the CFSM has only confluent complete states and it is therefore confluent.

The Mn procedure will aim to explore whether there are any minimal non-confluence patterns.

2.4.4 M0 – the first generation

The Mn-procedure is a procedure which consists of executing a tree of "machines". The name "Mn" comes from "Machine on level n". The root machine is M0 and it is defined in Figure 36 (p. 56) as a transition system. The execution of the machine starts from the initial states and continues along the transitions. The transition systems are infinite, but the Mn-procedure is designed such that only a (small) portion of the transition system is visited.

The Mn-procedure has a simple goal. By trying out all potential minimal non-confluence patterns as defined in Figure 35 (p. 55), we shall eliminate them incrementally and finally conclude that the CFSM under analysis is absolutely confluent as defined in Figure 29 (p. 52).

If we find non-confluence, this is normally a good indicator of errors or complexities in the CFSM design.

2.4.4.1 Formal definition of M_0

We define the root machine M0 in Figure 36 (p. 56).

$M_0 = \langle K_0;A_0;\xrightarrow[M0]{};Z_0\rangle$ is a transition system based on CFSM <S;C;Z;T> and the derived transition system $\langle K;A;\longrightarrow;Z\rangle$.

$K_0 = K \times K$ the set of M0-nodes, pairs of complete states

$A_0 = \{((\text{ø}, a),(\text{ø}, a))|a \in I\}$ pairs of equal internal signals

The empty sequences of the A_0-elements refer to the external input.

$$Z_0 = \left\{(x,x') \in K_0 \middle| \left(p \in S, u \in E, v \in I \wedge (p;u;v;\text{ø}) \xrightarrow{u} \xrightarrow{v} x \wedge (p;u;v;\text{ø}) \xrightarrow{v} \xrightarrow{u} x'\right)\right\}$$

which represents the potential minimal non-confluence patterns.

$$(x,x') \xrightarrow[M0]{} (y, y') \Leftarrow \exists(((\text{ø}, i),(\text{ø}, i')) \in A_0) \bullet [i + x] \xrightarrow{i} y \wedge [i' + x'] \xrightarrow{i'} y'$$

where the plus operator again is the concatenation operator used in Figure 17 (p. 46).

Figure 36: M_0 definition

The execution of M_0 represents the execution in the CFSM of all possible minimal non-confluence patterns $(p;u;v\beta;\text{ø})$ where $\beta \in I^*$.

The Mn-procedure will calculate the Z_0 states and evaluate them. Thereafter, depending on the evaluation, further execution of M_0 might take place. Whenever a new node is generated it will be evaluated. See Section 2.4.6 (p. 65) for termination criteria of M_0.

2.4.4.2 Evaluation of the states of M0

Every node in M_0 is a pair of complete states of the CFSM. In general we may depict the node as $((s;ø;i;o),(s';ø;i';o')) \in K_0$. The two elements of the pair are compared. The evaluation criteria should be applied in the order they are presented below. Implicitly, then, the negation of all earlier criteria can be assumed.

1. *Confluent branch.* $(s = s') \wedge (i = i') \wedge (o = o')$
   Since A_0 is a *parallel* alphabet (its two components are equal), it is clear that if the two elements of a node of M_0 are equal, all further execution of M_0 from that node will produce states which have pairs of equal elements. Stabilization of any M0-node with equal elements will have to reach a stable node with equal elements since the internal signals consumed during the stabilization are also in equal pairs.

2. *Non-confluence 1.* $\neg(o \approx o')$
   If the external output signals of one element of the M_0 node is not a prefix of the output signals of the other element (or vice versa), we know that no further execution of M_0 may turn the elements equal since further executions only *append* signals to the external output sequences.

3. *Non-confluence 2.* $(s = s') \wedge (i = i') \wedge (o \approx o') \wedge (o \neq o')$
   We know that stabilization will reveal non-confluence if the state elements are equal and the internal queues are equal, but the external output is prefix related, but not equal. This is because the stabilization will produce equal appendices on both elements and the outputs cannot become equal.

4. *Stabilization.*
   As noted in Section 2.4.4.1 (p. 56) the execution of M_0 represents possible contents of the internal signal queue of the potential minimal non-confluence pattern $(p;u;v\beta;ø)$. We may also assume that the remaining tail of β is empty and perform the concrete stabilization. We evaluate the leaves. If the node elements of the stabilized node are not equal, we conclude non-confluence otherwise we continue to check the next evaluation criterion.

5. *Sequence permuted 1.* $(o \approx o') \wedge \neg(i \approx i')$
   If the external queues are prefix related, but the internal signal queues are not prefix related, we can neither conclude confluence nor non-confluence as the situation may change during further execution and stabilization. We may conclude, however, that continued execution of M_0 will not produce nodes which are confluent since further execution cannot change the non-prefix situation as further execution only appends to the queues.

6. *Sequence permuted 2.* $(o \approx o') \wedge (s = s') \wedge (i \approx i') \wedge (i \neq i')$
   If the situation is that the state elements are equal and the internal queues are prefix related, but the internal queues are not equal, we may also conclude that further exe-

cution of M_0 cannot produce nodes with equal elements since the subsequent appendices of the queues will always be the same since the state elements are equal and the alphabet A_0 parallel.

7. *State different*. $(o \approx o') \wedge (s \neq s') \wedge (i \approx i')$
   The remaining situations based on states, internal and external queues are that state elements are unequal but the queues are prefix related. Continued execution of M_0 would have a chance to produce nodes with equal elements. However we cannot be sure whether execution of M_0 persists forever only producing such state different situations.

The evaluation determines the continued execution within M_0 or on higher generations.

1. *Confluent branch.* We do not execute in M_0 further along this branch as no non-confluence can be found by further execution here. We are still in business for determining confluence.
2. *Non-confluence.* We have found non-confluence. The minimal non-confluence pattern can be derived from the execution path in M_0. It is possible that the found non-confluent complete state is non-reachable, but this is beyond our Mn-procedure. Unless supplementary techniques exclude the found non-confluence pattern, the Mn-procedure will cease with conclusion that the CFSM is non-confluent.
3. *Stabilization*. Similarly stabilization may find non-confluence and the Mn-procedure terminates with non-confluence verdict. If the stabilization shows equal leaf state sets, we are still in business and continue the evaluation of the node.
4. *Sequence permuted*. We cannot conclude at this point. We know, however, that we shall not benefit from continued M_0 execution[1], and therefore we need to find a more complex solution. That solution is to produce another transition system on a higher generation level. This is covered in Section 2.4.5 (p. 59).
5. *State different*. Here we tentatively continue the M_0 execution along this branch hoping that continued execution will make the elements confluent.

2.4.4.3 M0 of the example process D

To determine confluence we will now perform Z_0-evaluation based on definition of D in Figure 24 (p. 50).

Table 2: Process D, initial set of M_0, Z_0

T#	State	Ext. signal	Int. signal	Z_0 element	Category
	{0,1,2}	{2}	{0,1}		
1	0	2	0	(1;ø;01;ø) (1;ø;10;ø)	seq. perm. 1
2	0	2	1	(0;ø;0;333) (0;ø;0;333)	confluent

1. We may perform further M0 execution and conclude confluence by “external stuttering”, see Section 2.4.5.2 (p. 61)

Table 2: Process D, initial set of M_0, Z_0

T#	State	Ext. signal	Int. signal	Z_0 element	Category
3	1	2	0	(2;ø;0;ø) (2;ø;ø;3)	non-confluent by stabilization
3-S	stabilize	-	-	(0;ø;ø;ø) (2;ø;ø;3)	non-confluent!
4	1	2	1	(0;ø;0;3) (0;ø;0;3)	confluent
5	2	2	0	(0;ø;ø;3) (0;ø;0;ø)	seq. perm. 2
6	2	2	1	(0;ø;ø;333) (0;ø;0;33)	seq. perm. 2 (similar to 5)

We see that there is a non-confluent node, and in principle we should conclude the whole Mn-procedure by the conclusion that the process D is not confluent. We shall later, however, show that the non-confluent state is not reachable. Thus a search for absolute confluence concludes non-confluent, but a search for plain confluence will show confluence.

There are no incidents of state different nodes and therefore we shall have to leave M0 and change generations, which will be presented in Section 2.4.5 (p. 59).

2.4.5 Mn — changing generations

The motivation for changing generation is that executing internal signals in M_0 sometimes results in new internal signals. Even though the internal sequences of the comparable complete states of the Mn node are different, this does not necessarily mean that the initial state is a non-confluence pattern. We must see what happens when these signals in due turn are executed.

2.4.5.1 Formal definition of Mn

We give the formal definition of the Mn transition system in Figure 37 (p. 60).

Some explanation may be needed. The idea behind the Mn transition system is that earlier generations of the Mn-procedure have not been able to provide a decision concerning a potential non-confluence pattern (s;e;iβ;ø) where the contents of β is not known. The executions in M0 amounts to trying out finite prefix sequences of β and we may have to conclude that continued M0 execution cannot reach a conclusion. By changing generation we create a transition system which also generates the results of situations where the internal output from M0 (in general M_{n-1}) is also consumed.

Assume that (s;e;iβ;ø) is a non-confluence pattern. M0 executes a prefix of β, but decides to change generation before the non-confluence has been detected. There will be a tail of β left (which we still may denote β without loss of generality). Changing generation means to resume the execution of the non-confluence pattern in the CFSM after β has been executed. Z_n refers to all possible states where the CFSM could possibly end up after β and the internal signals (here: (c,c')) produced in the former generation (say M0)

$M_n[((s,c),(s',c'))] = \langle K_n;A_n;\xrightarrow[Mn]{};Z_n\rangle$ is a transition system relative to $((s,c),(s',c')) \in K_{n-1}$ in M_{n-1} and based on CFSM $\langle S;C;Z;T\rangle$ and the derived transition system $\langle K;A;\longrightarrow;Z\rangle$.

$K_n = \overline{K_n} \times \overline{K_n}$ the set of Mn-nodes, pairs of generalized complete states where

$\overline{C_n} = E^* \times I^* \times O^* \times \overbrace{O^* \times \ldots \times O^*}^{n}$ the type of generalized CFSM alphabet

$\overline{K_n} = S \times \overline{C_n}$ the type of generalized complete state

$C_n = \overline{C_n} \times \overline{C_n}$ the type of the generalized Mn alphabet

$$A_n = \left\{(x,x') \in C_{n-1} \middle| \exists (p,p'), (t,t') \in R_{n-1}(s, s') \bullet ((p,\phi),(p',\phi)) \xrightarrow[M_{n-1}]{} ((t,x),(t',x'))\right\}$$

where ϕ is the (element) with only empty sequences and R_{n-1} is a set valued function which returns all reachable basic states (i.e. of type $S \times S$) of M_{n-1} from the operand basic state. R_{n-1} is decidable by executing M_{n-1} and pruning when reaching a basic state already visited. This will terminate since there is a finite set of basic states.

$$Z_n = \left\{((t,d),(t',d')) \in K_n \middle| \left((p,p') \in R_{n-1}(s, s') \wedge ((p,c),(p',c')) \xrightarrow{(c, c')} ((t,d),(t',d'))\right)\right\}$$

where the labelled transition refers to a generalization of the labelled transition relation of the original transition system shown in Figure 38 (p. 61).

$$(x,x') \xrightarrow[M_n]{} (y,y') \Leftarrow \exists (i, i') \in A_n \bullet ([i + x],[i' + x']) \xrightarrow{(i, i')} (y,y')$$

Figure 37: Definition of Mn

has been executed. The execution of β has produced a sequence of input symbols to M_n which must be a sequence of output symbols from the former generation. A_n gives the alphabet of output symbols of the former generation.

The complete states on higher generations need a slight generalization. We need to keep the external output from the different generations separate. This is because the external output from a late generation actually in time appears after external output from earlier generations. When we execute Mn we actually simulate executions on all generations before the n-th simultaneously.

The labelled transitions in Figure 37 (p. 60) refer to generalized transitions based on the transition table of the CFSM itself as shown in Figure 38 (p. 61).

$\xrightarrow{(c, c')} : K_n \times A_n \times K_n$

where each of the elements of the pair is given by the following simpler labelled transition:

$$(s;ø;iu;o_0v_0, \ldots, o_{n-1}v_{n-1}, v_n) \xrightarrow{(ø;i;o_0, \ldots, o_{n-1})} (t;ø;uj;v_0o_0, \ldots, v_{n-1}o_{n-1}, v_nw_n)$$

$$\Leftarrow (T(s;ø;i;ø) = (t;j;w_n))$$

Figure 38: Generalized labelled transition

In Figure 38 (p. 61) we assume that elements of the alphabet $\overline{C_n}$ can be seen alternatively as a tuple of sequences (according to the definition of the type in Figure 37 (p. 60)) or as a series of tuples where each element is in $\overline{C_n}$ and the piecewise concatenation of the series makes up the tuple of sequences.

We notice that the external output elements of the symbol are just removed from the head of the sequence and appended to the tail.

2.4.5.2 Evaluating nodes of M_n

The Mn-procedure on this generation is very similar to the one on M_0 as the nodes are evaluated as they are generated through the execution. The evaluation then determines the further execution.

The evaluation of M_n nodes follows very much the same lines as with M_0 nodes.

Let $h = ((s;ø;i;e_0, \ldots, e_n), (t;ø;j;f_0, \ldots, f_n))$ be the node in $M_n[q]$ to be evaluated where (i,j) is the sequence(s) of internal signals, (e_k,f_k) represent the k-th generation of external output.

Confluent branch

We recall that a confluent branch is a branch where we are certain that the Mn execution graph from the node under analysis will only consist of nodes where the complete states in the pair are equal when they are stabilized. In the general situation it is not as simple as in M_0 since a node where the two complete states are absolutely identical is *not* sufficient. If the A_n alphabet has a pair of symbols where the symbols are not identical, continued execution may bring the node with the identical complete states to a node where the two complete states are *not* identical. From this we can conclude that also the alphabet plays a role in the confluence conclusion. We shall give a series of sufficient criteria for the confluence conclusion for a node h. In Section 2.4.6.2 (p. 67) we give a more detailed walkthrough of why the criteria are sufficient.

Everything equal

The simplest and most obvious set of requirements is only a slight generalization of the M_0 case.

1. The basic states of the node are equal;

2. The internal queues are equal;
3. Each generation of the output queues are equal;
4. The input alphabet has only symbols that are pairs of equal sequences of signals.

That this leads to confluence should be obvious. The two parts of the Mn execution are absolutely identical in all respect which affect the Mn execution.

Generation glue

Let us first relax the third criterion, and this will also affect the fourth requirement. In fact we are not interested in absolute identity before the stable states. The final stabilized states are compared not generation by generation of the output sequence, but the concatenations of the output queues. The reason why this may be significant is that there may be situations where internal signals on one generation are "compensated" and transformed to external output on another generation. Relative to the process execution graph G this means that the necessary external outputs are produced at different depths in the two alternative execution branches. There is a generation change in between these two execution depths. We cannot merely compare concatenated output sequences as can be seen from our explanation of why the Mn procedure works in Section 2.4.6 (p. 65). The reason is that further execution inserts outputs into the concatenated sequences.

After some consideration about how this insertions of more output signals can be neutralized such that the stabilizations will become equal in the whole Mn execution graph below the node under analysis, we have reached the following criterion for the output sequences and the associated alphabet.

If h can be written as
$h = ((s;ø;i;\beta_0\alpha_0, \ldots, \beta_{n-1}\alpha_{n-1}, \beta_n), (s;ø;i;\beta_0, \alpha_0\beta_1, \ldots, \alpha_{n-1}\beta_n))$ and A_n is *parallel* relative to h, we conclude that h is confluent and this branch of M_n may be terminated with success.

That A_n is *parallel relative to h* means that all its elements follow the form:
$((ø;j;\gamma_0\alpha_0, \ldots, \gamma_{n-1}\alpha_{n-1}), (ø;j;\alpha_0\gamma_0, \alpha_1\gamma_1, \ldots, \alpha_{n-1}\gamma_{n-1}))$ where the α-s are the same as those in h. They represent common *glue* signals between the generations.

The point, of course, is to make sure that subsequent executions from h will not destroy the property of confluence which means that the two elements of the pair will be equal if the external output is concatenated.

We notice that this somewhat complicated property is weaker than requiring that the external output shall be piecewise equal. We also notice that M_0 conforms trivially to this requirement.

Equal output

We notice with the generation glue above that the requirements are put on the input alphabet of the Mn execution. But it is possible to relax the requirement of the alphabet even further. Since the node under evaluation will also be checked for stabilization, we are here more interested in the possible further executions. It is true that the internal components of the alphabet symbol do not have to come in equal pairs. What we actually must require is that their output is equal. Thus we should rather check that the alphabet

A_{n+1} (relative to a possible generation change at the node under evaluation), is parallel. We have to check this criterion for all basic states in the reachability set of R_n of the node under evaluation.

The difference in checking parallelism of the output alphabet rather than the input alphabet is significant when the reachability set R_n is (clearly) smaller than the set of all basic states and in this set the difference in the input alphabet is not significant with respect to. the output.

External stuttering

The clue to conclude confluence of a node is that further execution will not bring in new possibilities for non-confluence. Our last sufficiency criterion relaxes the two first requirements that the states and the internal queues of the node under evaluation must be equal. The *external stuttering* criterion is based on "deja vu", something which has been encountered before comes up again. Assume that the node under evaluation is equal to another node encountered (and evaluated) before, then it is obvious that a new evaluation will result in the same confluence verdict. Our external stuttering criterion is based on this.

1. Assume that the node under evaluation is similar to a node which has been analyzed before as confluent. To be "*similar*" means that the node is modified by equal sequences of the external output. Since external output plays no role in stabilization, it is obvious that this modified node must also be confluent.
2. Assume then that there is a node which is similar to another node higher up in the current execution tree. That node has not been concluded as confluent yet as a continued execution is being performed. If stabilization of the current node under evaluation is acceptable, the branch can be concluded confluent.

The reason for this conclusion is again that the external output plays no role in stabilization. Therefore any problems of stabilization for nodes following the node under evaluation will have a structurally equivalent counterpart from the similar node. Thus any non-confluence in the execution graph of the node now evaluated is sure to turn up in another branch (which also is shorter) in the current Mn execution tree.

Non-confluence

Let $h = ((s;ø;i;e_0, \ldots, e_n), (t;ø;j;f_0, \ldots, f_n))$. If $\exists i \bullet \neg(e_i \approx f_i)$ which means that there is a generation component of the external output which is not prefix related, we conclude that non-confluence is possible and we terminate the whole procedure.

In fact we must suppress any "glue" of the alphabet presented earlier in this section. This means that glue from the alphabet must be removed from each generation element of the external output before the prefix relation is checked.

Stabilization

We stabilize h and check that $(s = t \wedge e_0 e_1 \ldots e_n \equiv f_0 f_1 \ldots f_n)$ where the external outputs have been concatenated across generations. If the formula does not hold, we conclude non-confluence, and the procedure should terminate.

This criterion corresponds closely to the one in M_0.

Sequence permutation

Let $h = ((s;ø;i;e_0, \ldots, e_n), (t;ø;j;f_0, \ldots, f_n))$. If $\neg(i \approx j)$, meaning that the internal sequence is not prefix related, we know that confluence cannot be reached by further M_n-execution from h. We must sooner or later change generation, which implies creating an M_{n+1}.

This is also the case if the states are equal (s=t), and the internal queue is prefix related but the two elements are not fully equal ($\neg(i = j)$).

The two cases correspond well with the two cases of sequence permutation identified for M0.

State different

If none of the above checks have triggered, we know that the state elements are different, but the internal queues are prefix related. We continue executing from h in M_n. We cannot in general be certain that the M_n execution terminates.

2.4.5.3 Mn of the example process D

From Table 2 (p. 58) we see that situation 1 is sequence permuted (type 1). We shall perform generation change.

Table 3: Generation change in state 1

#	$R_1[T_1]$ [a]	$A_1[T_1]$	Z_1 element	Category
1	{(0,0),(1,1),(2,2)}	{((ø,1,ø),(ø,1,ø)), ((ø,ø,333),(ø,ø,333)), ((ø,ø,ø),(ø,ø,ø)), ((ø,ø,33)(ø,ø,33))}		

a. T_i refers to the complete state on line i in the transition table T.

We see that the next generation alphabet is parallel. Alphabet elements where the only difference is equal appendix of external signals can be considered equivalent [1] . Furthermore the elements with only empty internal queues can be excluded as they will cause immediate external stuttering.

Table 4: Execution of M_1 from state 1

#	$R_1[T_1]$	$A_1[T_1]$	Z_1 element	Category
1	{(0,0),(1,1),(2,2)}	{((ø,1,ø),(ø,1,ø))}	base=(1;ø;01;ø) (1;ø;10;ø)	
11 [a]	(0,0)	gen. change	(0;ø;1;ø,3) (1;ø;1;ø,333)	state different
11-1		((ø,1,ø),(ø,1,ø))	(0;ø;1;ø,3333) (0;ø;1;ø,3333)	confluent
12	(1,1)	gen. change	(0;ø;ø;ø,33) (1;ø;1;ø,3)	state different

1. This is not entirely true as the appendix must adhere to the rules of confluence laid down in the evaluation of Mn-complete states in Section 2.4.5.2 (p. 61).

Table 4: Execution of M_1 from state 1

12-1[b]		((ø,1,ø),(ø,1,ø))	(0;ø;ø;ø,33333) (0;ø;1;ø,33)	seq. perm.
12-1-S		stabilization	(0;ø;ø;ø,33333) (0;ø;ø;ø,33333)	ok
12-1-1		((ø,1,ø),(ø,1,ø))	(0;ø;ø;ø,33333 333) (0;ø;1;ø,33 333)	external stuttering wrt. 12-1
12-1-1-S		stabilization	(0;ø;ø;ø,33333 333) (0;ø;ø;ø,33 333 333)	external stuttering stabiliza-tion ok
13	(2,2)		(0;ø;ø;ø,333) (1;ø;1;ø,33)	similar to 12

a. The numbering scheme here is that the states of Z_0 has the numbers 1,2,3, while states of Z_1 has numbers such that the prefix designates which zeroth generation state it was based upon.

b. The numbering scheme within a generation is such that 12-1 is a result of executing from state 12, 12-2 is another execution result from state 12. 12-1-1 is an execution result from 12-1 and so forth.

In Table 4 (p. 64) we have an example of a state different node (11) where continued execution leads to a confluent state directly (11-1). We also have an example of state different nodes (12) where the continued execution leads to 12-1 which is a sequence permuted node where external stuttering can be used to ensure confluence.

Since all of the branches following node “1” (namely “11”, ”12” and “13”) are concluded with confluence, we have shown that the whole branch “1” is confluent.

2.4.6 Why the Mn procedure works

Having presented the Mn procedure and explained how the nodes are evaluated, there may still be some who feel uncertain about whether the Mn procedure is sure to uncover any non-confluence pattern. Here we shall go through this in greater detail.

2.4.6.1 A detailed walkthrough of the Mn procedure

The idea of the detailed walkthrough is to compare the execution within the original CFSM of an assumed non-confluence pattern (s;e;iβ;ø) with the coverage of the Mn-procedure.

1 We have in Section 2.4.3 (p. 52) shown that it suffices to detect minimal non-confluence patterns.

2 Assume that (s;e;iβ;ø) is a complete state of the CFSM which is the root of a minimal non-confluence pattern. $\beta \in I^*$, $e \in E$, $i \in I$. We then have that from definition of non-confluence
$L(T_e(T_i(s;e;i\beta;ø))) \neq L(T_i(T_e(s;e;i\beta;ø)))$.
The execution graph $G(I,s,N,T;s)$ has two separate branches since the choice to execute an external input is only present at the root. After having executed e and i the corresponding states in G are

$((s_1;ø;\beta i_1;o_1), (s_1';ø;\beta i_1';o_1'))$ by symbolically executing the T-functions. We may also write this

$((ø;\beta_1;ø), (ø;\beta_1;ø)) + ((ø;\beta_2;ø), (ø;\beta_2;ø)) + \ldots$

$\ldots + ((s_1;ø;i_1;o_1), (s_1';ø;i_1';o_1'))$

by again seeing the tuple of sequences as a concatenated series of tuples where $\beta_i \in I$.

3 From the definition of M_0 we have that $(T_e(T_i(s;e;i;ø)), T_i(T_e(s;e;i;ø))) \in Z_0$. From the definition of M_0 we see that the execution of M_0 follows the execution of G. The internal signals β_i are executed in the order of their suffixes. We shall compare M_0 and G.

3.1 If M_0 executes the whole β sequence, the non-confluence will be revealed by the stabilization of the node reached when having executed the whole β sequence.

3.2 If M_0 has not covered the whole β sequence, this means that evaluations of an M_0 state has interrupted the M_0 execution before the whole β sequence was executed:

3.2.1 *Confluence* cannot have occurred since then no further execution or stabilization could produce non-confluence and that was our assumption in 2.

3.2.2 *Non-confluence* can have occurred which means that another non-confluence pattern has been detected. The non-confluence verdict of the Mn procedure is still correct.

3.2.3 *Stabilization* leading to non-confluent situation can also have occurred and similarly to the non-confluence evaluation, another non-confluence pattern has been found, but the Mn procedure verdict is still correct.

3.2.4 *Sequence permutation* may have occurred and generations changed. The node where the generation change occurred is in general $((s_2;ø;i_1i_2;o_1o_2), (s_2';ø;i_1'i_2';o_1'o_2'))$. The corresponding pair of states in G is $((s_2;ø;\gamma i_1i_2;o_1o_2), (s_2';ø;\gamma i_1'i_2';o_1'o_2'))$ where γ contains a sequence which is the tail of β.

3.2.4.1 Now there is a discrepancy between the execution of G and that of M_n. The idea is that M_1 should "catch up" with G by trying all points where G possibly can land. We finish the execution of γ in G and conclude that it results in state pair $((s_3;ø;i_1i_2\delta;o_1o_2\varphi), (s_3';ø;i_1'i_2'\delta';o_1'o_2'\varphi'))$.

3.2.4.2 It is no doubt that if we had continued executing M_0 we would have reached the same state pair as in 3.2.4.1. Thus we have that $(s_3, s_3') \in R_0(s_2, s_2')$ which says that the basic state of the reached state in G is reachable from the basic state of the generation change in M_0. Furthermore we know that $((ø;\delta;\varphi), (ø;\delta';\varphi'))$ has been produced by M_0 execution. Thus according to the definition of A_1 we have that $((ø;\delta;\varphi), (ø;\delta';\varphi')) \in A_1^*$.

3.2.4.3 Continued execution of G after the γ has been executed, must now start by executing the internal signals $(i_1i_2, i_1'i_2')$. This will lead to $((s_4;ø;\delta i_4;o_1o_2\varphi o_4), (s_4';ø;\delta' i_4';o_1'o_2'\varphi' o_4'))$.

3.2.4.4 From the definition of Z_1 we have that since our generation change state is

$((s_2;ø;i_1i_2;o_1o_2), (s_2';ø;i_1'i_2';o_1'o_2'))$ and since $(s_3, s_3') \in R_0(s_2, s_2')$, we must have that $((s_4;ø;i_4;o_1o_2, o_4), (s_4';ø;i_4';o_1'o_2', o_4'))$ is a node in M_1. The correspondence between G and M_1 has again clearly been established.

3.2.4.4.1 Assume that M_1 executes the whole sequence $((ø;\delta;\varphi), (ø;\delta';\varphi')) \in A_1^*$ we reach $((s_5;ø;i_4\vartheta;o_1o_2\varphi, o_5\theta), (s_5';ø;i_4'\vartheta';o_1'o_2'\varphi', o_5'\theta'))$. Notice the generalized transition relation used for the external output.
Correspondingly G will execute the internal signals (δ, δ') and reach $((s_5;ø;i_4\vartheta;o_1o_2\varphi o_5\theta), (s_5';ø;i_4'\vartheta';o_1'o_2'\varphi' o_5'\theta'))$. Stabilization and external output concatenation will show non-confluence. The obvious correspondence between G and M_1 leads to this.

3.2.4.4.2 If M_1 does not execute the whole sequence, we are back to very much the same situation as 3.2, only now we are on a higher generation. The approach repeats itself again on higher generations. Formally the proof can be done through induction on the number of generations.

3.2.5 *State different* may have occurred, but this will not terminate the execution of the β sequence.

4. Finally we must conclude that *no* non-confluence pattern can escape through our sieves of M_n generations.

2.4.6.2 A more detailed walkthrough of the confluence criteria

We presented in Section 2.4.5.2 (p. 61) four sufficient criteria for concluding confluence. The reasons why the three last criteria are actually sufficient for concluding confluence may not be entirely obvious. Therefore we walk through them in greater detail here.

Generation glue

The idea is that the production of the external output may happen in different generations in the two alternative branches of execution from the potential non-confluence pattern.

If h can be written as
$h = ((s;ø;i;\beta_0\alpha_0, \ldots, \beta_{n-1}\alpha_{n-1}, \beta_n), (s;ø;i;\beta_0, \alpha_0\beta_1, \ldots, \alpha_{n-1}\beta_n))$ and A_n is parallel relative to h, we conclude that h is confluent and this branch of M_n may be terminated with success.

That A_n is *parallel relative to h* means that all its elements follow the form: $((ø;j;\gamma_0\alpha_0, \ldots, \gamma_{n-1}\alpha_{n-1}), (ø;j;\alpha_0\gamma_0, \alpha_1\gamma_1, \ldots, \alpha_{n-1}\gamma_{n-1}))$ where the α-s are the same as those in h. They represent common "glue" signals between the generations.

Let us execute from h using some arbitrary element of A_n. According to the execution rule of the generalized transition relation given in Figure 38 (p. 61) we get the following result shown in Figure 39 (p. 67).

$$(t;ø;ik;\beta_0\alpha_0\gamma_0\alpha_0, \ldots, \beta_{n-1}\alpha_{n-1}\gamma_{n-1}\alpha_{n-1}, \beta_n\gamma_n),$$
$$(t;ø;ik;\beta_0\alpha_0\gamma_0, \alpha_0\beta_1\alpha_1\gamma_1, \ldots, \alpha_{n-1}\beta_n\gamma_n)$$

Figure 39: Executed the parallel alphabet symbol

We see that the concatenation of the output signals over generations gives equal results for the two elements of the state pair.The α-s are still the glue. It is obvious that stabilization will give identical (confluent) results.

Equal output

We noticed also that we are more interested in the result of the execution as shown in Figure 39 (p. 67) than on the input alphabet. The input alphabet and the output alphabet share the first n-1 generations. A parallel output alphabet should have equal internal component and equal external component on n-th generation. Referring to Figure 39 (p. 67) the internal component is k and the external component γ_n. What we gain is that the internal components of the input alphabet symbols need not be identical. We need to find the output alphabet when the basic states are in the reachability set of the basic state of h since this is the execution graph we want to cover.

External stuttering

Let $h = ((s;ø;i;e), (s';ø;i';e'))$. Then we assume that execution of M_0 leads to another similar state $h' = ((s;ø;i;eo), (s';ø;i';e'o))$ where the only difference on both branches is the addition of the external output signal sequence o. We have stabilized and found confluent both h and h' and all nodes in between.

Our hypothesis is that any node in the execution graph of h' will have a counterpart in the execution graph of h with shorter length from the potential non-confluence pattern which is the root of the Mn execution. This counterpart state will be analyzed for confluence. Any non-confluence in the execution graph of h' will also appear as non-confluence in the counterpart.

First we stabilize h. The stabilization results in appendices u and u' on the external output sequences. Since the difference between h and h' is only in the external output, the appendix is the same for h and h'. Since both h and h' were confluent when stabilized, the output sequences must be identical.

1. $eu = e'u'$
2. $eou = e'ou'$

From 1 we see that we may assume $e' = e\bar{u}$ without loss of generality. This leads to

1.1. $eu = e\bar{u}u'$

which by stripping off the prefix e leads to:

1.2. $u = \bar{u}u'$

Substituting 1.2 and e' into 2, we get:

2.1 $eo\bar{u}u' = e\bar{u}ou'$

and stripping off prefix e and postfix u' leads to:

2.2 $o\bar{u} = \bar{u}o$

This string equation has the solution:

3. $o = x^n$ and $\bar{u} = x^m$ for some non-negative integers n and m and shorter sequence of signals x.

Assume now that we execute from h' and reach:

4. d' = ((t;ø;ij;eov), (t';ø;i'j';e'ov'))

We want to show that the stabilization of this *must* be confluent. If we stabilize 4 we reach in general:

5. d" = ((t;ø;ø;eovw), (t';ø;i'j';e'ov'w'))

A counterpart of d' is found when the same execution which led from h' to d' is applied to h. Since the same execution is performed from the same basic states, we have:

6. d = ((t;ø;ij;ev), (t';ø;i'j';e'v'))

We may assume that d stabilized is confluent which means:

7. evw = e'v'w' since the stabilization follows the same execution of internal signals as we could find in stabilizing d' to d" (in 5).

Then we substitute for $e' = e\bar{u}$ and also $\bar{u} = x^m$ and get:

8. $evw = ex^m v'w'$ which means

8.1 $vw = x^m v'w'$

Then we return to the two sides of the node in 5. (First the left side)

9. $eovw = ex^n x^m v'w'$ from 3 and 8.1

Then we take the right hand side of 5.

10. $e'ov'w' = ex^m x^n v'w'$ from $e' = e\bar{u}$ and 3

We see from 9 and 10 that the two sides of the external output of d" must be equal and thus confluent.

We have shown that external stuttering criterion is sufficient.

2.4.7 Why the Mn-procedure may not terminate

We mentioned briefly that the Mn procedure itself has no strong termination criteria. There are two different possibilities of executing the Mn procedure eternally:

1. In some Mn, there is an infinite series of state different situations.
2. Changing of generations take place infinitely. There is no upper limit to the number of generations.

We show one example of each of the two situations.

2.4.7.1 Infinite series of state different situations

In Figure 40 (p. 70) we see extracts of a process which seems to be progressive because it follows the signal ordering criterion for all those transitions we see. Still performing M_0 from complete state (S;e;i;ø) leads to an infinite stuttering as we can see from the following. Executing the external and the internal signal leads to ((T;ø;ij;ø),(U;ø;i;ø)).

This situation is state different and continued execution of M_0 is advised. Continued execution of M_0 will give an infinite branch on executing (i,i) giving ((T;ø;ijj...j;ø),(U;ø;ij...j;ø)).

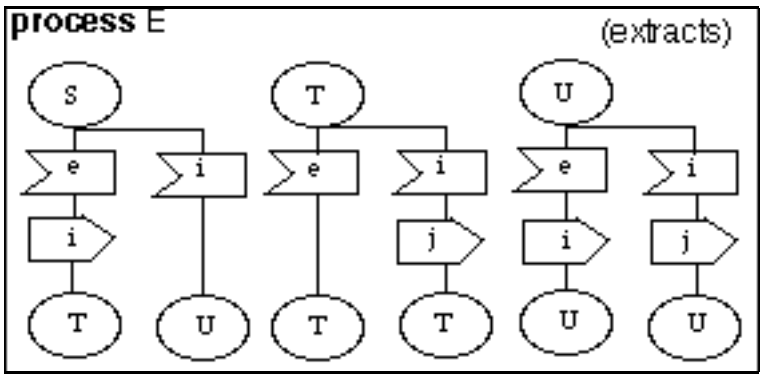


Figure 40: Process E which makes M_0 livelocked

We conclude that the Mn-procedure may not terminate even when the process under analysis is progressive.

2.4.7.2 Infinite generation changes

The execution of the Mn-procedure becomes more complex than one expects with process G of Figure 41 (p. 70) and the alphabet does not converge towards expressing only external signals since the output of a and b will always be present. Changing generations does not help as neither the state space nor the alphabet decreases. To perform the Mn-

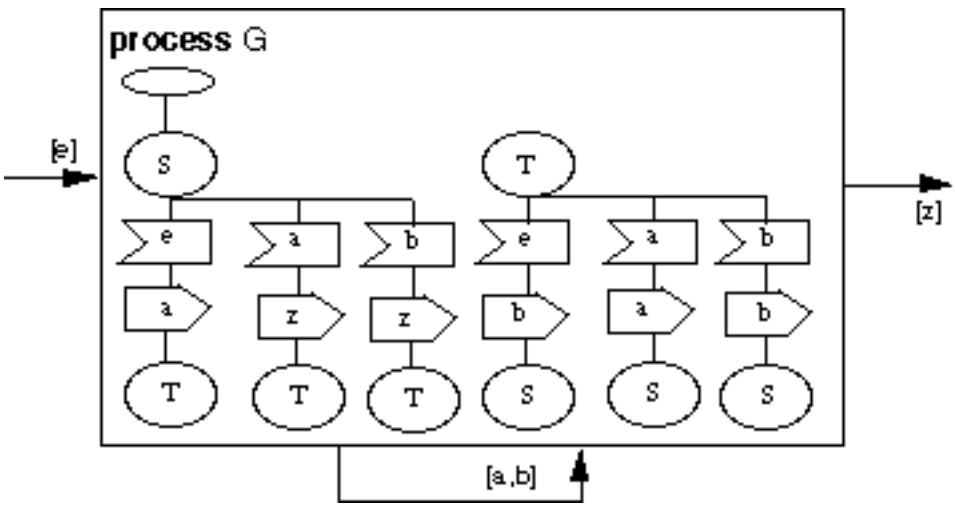


Figure 41: Process G which makes infinite number of generations

procedure on process G is left as an exercise to the reader. The clue to the unexpected complexity is that a and b are distinctly different signals, but they behave identically in the process.

We may notice that the process is not infinitely progressive (see Section 3.1.3 (p. 86)), and it does not keep to signal ordering criterion (see Section 2.6.4.1 (p. 80)). It is, however, progressive for all finite complete states and it is quite simple to see that it is

actually reducible. Each external input e produces either an a or a b. Whenever an a or a b is consumed either an external z is output or the internal signal is reproduced. Eventually all internal signals will be consumed and external z output since when the internal signals are reproduced the state changes as well. The external output z concludes the execution of an external input e and the system must reside in state T afterwards. This gives the reduction shown in Figure 42 (p. 71).

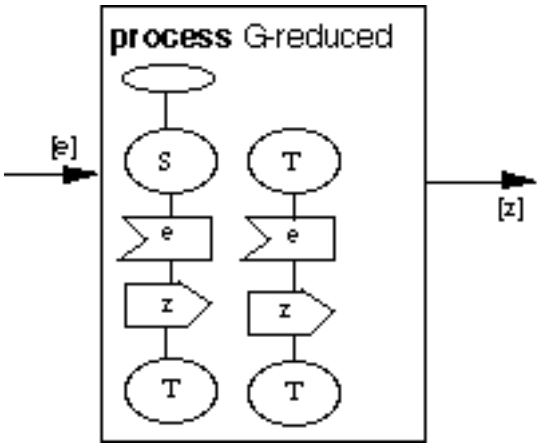


Figure 42: Process G reduced

2.4.8 Why the stabilization step is necessary

In Section 2.4.7 (p. 69) we showed that the Mn-procedure is not certain to terminate and therefore there should be a need for pragmatic modifications to the Mn-procedure to ensure termination.

In this section we look at the necessity of the stabilization step during the evaluation of nodes. One may get the impression that the stabilization step is a matter of optimization and that the same result would appear if the execution was continued within the same Mn generation. This is not the case.

2.4.8.1 Stabilization step of example process D

In our example process D where the M0 execution is shown in Table 2 (p. 58), we have state 3 revealed by stabilization to be non-confluent. If we continue execution in M0 we get the states 3-1 and 3-2 shown in Table 5 (p. 71).

Table 5: Continued execution of M0

#	Int. signal	node	category
3		(2;ø;0;ø) (2;ø;ø;3)	
3-1	(0,0)	(0;ø;0;ø) (0;ø;ø;3)	seq. perm.
3-2	(1,1)	(0;ø;0;33) (0;ø;ø;333)	seq. perm.

The nodes labelled 3-1 and 3-2 are both confluent when stabilized. Thus the non-confluent situation disappeared when the M0 execution was continued.

The node labelled 3 is sequence permuted (type 2) if we disregard the stabilization step. A generation change directly will of course reveal the problem since a series of generation changes is a general stabilization.

2.4.8.2 Stabilization step of example process J

That generation change reveals what the stabilization step reveals is not quite good enough since generation change is not really necessary for every node.

We have here an example process J in Figure 43 (p. 72) which shows that the stabilization criterion may reveal non-confluence also in cases where the node is state different (disregarding stabilization) and the evaluation thus implies continuation of the current Mn generation. We see that it is possible that the process is confluent “in the long run”, but not “close to the start”.

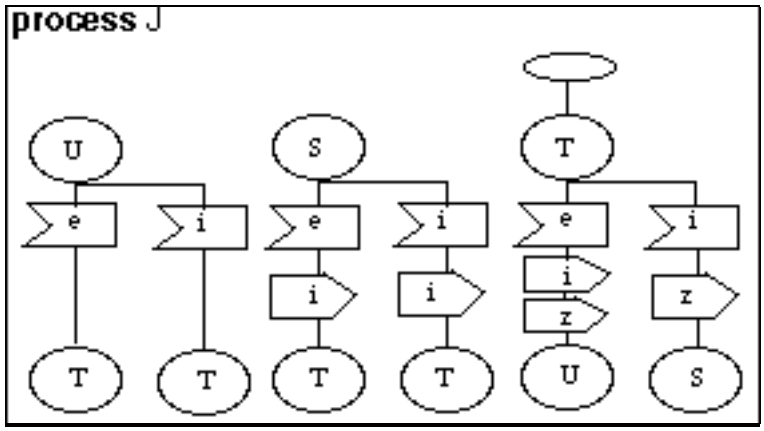


Figure 43: Stabilization of intermediate results are needed

In Figure 44 (p. 72) we display a part of the execution of M_0 of process J from (U,e,iβ,ø) to show why stabilization is necessary.

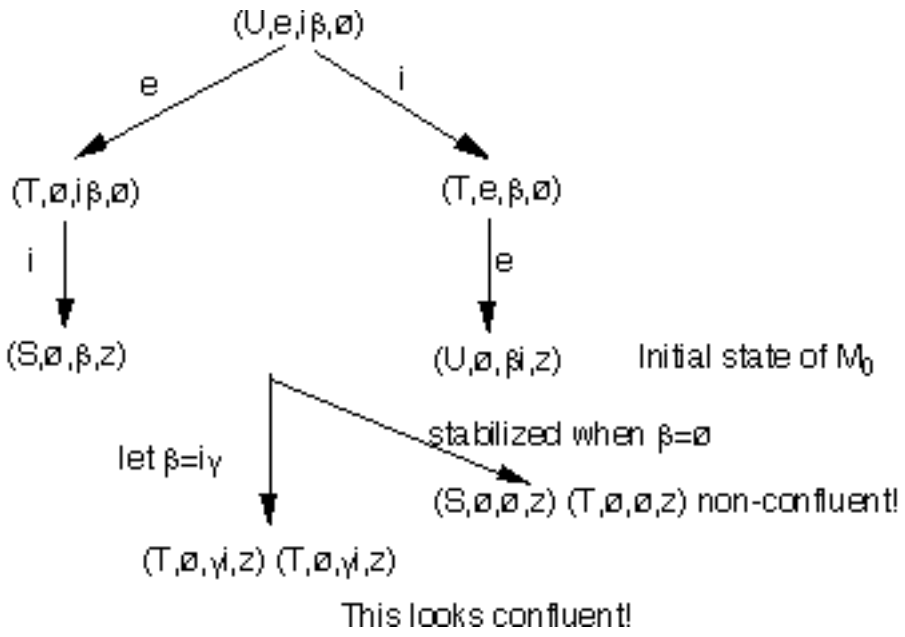


Figure 44: Part of the execution tree of M_0 of process J

2.5 Reducibility revisited

We motivated our presentation of progress and confluence by our reduction algorithm in Section 2.2.2 (p. 48). Having discussed progress and confluence, we may now return to reducibility to show that our reduction algorithm really yields a reduction as defined in Figure 22 (p. 48)

2.5.1 Why the reduction algorithm yields a reduction

The question now is whether the algorithm in Section 2.2.2 (p. 48) actually gives a reduced process according to our definition in Section 2.2.1.1 (p. 48) provided that the process is progressive and confluent.

The requirements on the sets S2, C2 and Z2 are trivially covered since Z2 is initialized to Z1 and not changed, S2 always gets elements from S1 and C2 is by definition given.

We now turn to the last criterion.

1. *To be proved*: $\forall z \in Z, \forall \theta_E \in E^* \bullet (L_{P1}(z;\theta_E;ø;ø) = L_{P2}(z;\theta_E;ø;ø))$

2. *Induction base*

   2.1 When θ_E = ø the statement of 1. is trivial since (z;ø;ø;ø) is a leaf state. This is actually sufficient as an induction base, but we also give the case where the sequence of external inputs has length 1.

   2.2 From the reduction algorithm 4.2, we get that
   $\forall p \in S2, \forall e \in E \bullet (L_{P1}(p;e;ø;ø) = T2(p;e;ø;ø))$

2.3 From definition of CFSM in Section 2.1.3.2 (p. 44) we have that $\forall p \in S2, \forall e \in E \bullet L_{P2}(p;e;ø;ø) = L_{P2}(T2(p;e;ø;ø))$ since the execution of e is the only choice from the state given.

2.4 By simple substitution of the result of step 2.2 into right hand side of step 2.3: $L_{P2}(T2(p;e;ø;ø)) = L_{P2}(L_{P1}(p;e;ø;ø))$

2.5 Since leaf states have no input signals we must have that $L_{P2}(L_{P1}(p;e;ø;ø)) = L_{P1}(p;e;ø;ø)$

2.6 By 2.2, 2.3, 2.4, 2.5 we get $\forall p \in S2, \forall e \in E \bullet L_{P1}(p;e;ø;ø) = L_{P2}(p;e;ø;ø)$ which constitute the induction base in our proof of 1.

3. *Induction step.*

3.1 Induction hypothesis:

$\forall p \in S2, \forall e_1 \ldots e_n \in E^n \bullet L_{P1}(p;e_1 \ldots e_n;ø;ø) = L_{P2}(p;e_1 \ldots e_n;ø;ø)$ for any n. We assume that the hypothesis holds for some specific n and shows that it will hold for n+1.

3.2 Let us now find $L_{P1}(z;e_0e_1 \ldots e_n;ø;ø) \quad z \in S2$

3.3 By executing the e_0 we get $L_{P1}(z;e_0e_1 \ldots e_n;ø;ø) = L_{P1}(p;e_1 \ldots e_n;i;ø)$ for some basic state p and some sequence of internal signals i.

3.4 By confluence (Figure 26 (p. 51)) we have $L_{P1}(T_i(p;e_1 \ldots e_n;i;ø)) = L_{P1}(T_e(p;e_1 \ldots e_n;i;ø))$ as long as the internal signal sequence is non-empty. By progress of P (Figure 25 (p. 50)) we know that execution of internal signals will terminate. Therefore we continue executing the internal signals and finally reach $L_{P1}(p';e_1 \ldots e_n;ø;o')$ for some basic state p' and sequence of external output signals o'.

3.5 Since earlier external output is irrelevant for the future execution, we can use the induction hypothesis in 3.1 on leading to $L_{P1}(p';e_1 \ldots e_n;ø;o')$

$L_{P1}(z;e_0e_1 \ldots e_n;ø;ø) = L_{P1}(p;e_1 \ldots e_n;i;ø)$ by 3.3

$= L_{P1}(p';e_1 \ldots e_n;ø;o')$ by 3.4

$= L_{P2}(p';e_1 \ldots e_n;ø;o')$ by induction hypothesis 3.1.

3.6 The execution in 3.4 is exactly the execution which would lead from $(z;e_0;ø;ø)$ to a stable state $(p';ø;ø;o')$ and this is exactly what the reduction algorithm does in step 4.. Therefore we have that $p' \in S2 \wedge (T2(z;e_0;ø;ø) = (p';ø;ø;o'))$.

3.7 Let us now find $L_{P2}(z;e_0e_1 \ldots e_n;ø;ø)$

3.8 We start by executing e_0 as given by 3.6 leading to $L_{P2}(z;e_0e_1 \ldots e_n;ø;ø) = L_{P2}(p';e_1 \ldots e_n;ø;o')$.

3.9 By 3.5 and 3.8 and we have proved the induction hypothesis for an arbitrary sequence of external signals which has length n+1.

4. The conclusion is that 1. holds and our reduction algorithm gives a reduction according to our own definition. QED.

2.5.2 Non-confluent, reducible process?

It is quite obvious that confluence comes in handy when a reduction is to be generated, but can there be reductions on processes that are *not* confluent?

If the process is non-confluent this means that there is a race condition which introduces non-determinism in the execution of the system. If we had a reducible, but non-confluent process this would mean that the reduction has to be able to express the non-determinism introduced by the non-confluence. The question of explicit non-determinism is handled in Section 3.5 (p. 97).

2.5.3 Mn-reduction is a Kwong-reduction

Here we prove that the Mn-reduction as defined by the reduction algorithm in Section 2.2.2 (p. 48) under the assumption that the system is confluent and progressive, is actually a Kwong-reduction[94]. Intuitively this is no big surprise since Kwong shows that Church-Rosser is preserved over Kwong-reduction, while we base our Mn-reducibility on a Church-Rosser criterion, confluence.

Kwong specifies four criteria for when a transition system is a reduction of another transition system. We shall go through these four criteria.

We showed in Section 2.5.1 (p. 73) that the reduction algorithm yields a reduction as defined in Figure 22 (p. 48) where P1=<S1;C1;Z1;T1> reduces to P2=<S2;C2;Z2;T2>. We shall show here that for their corresponding transition systems
$\langle K1;A1;\longrightarrow_1;Z1\rangle$ is reduced to $\langle K2;A2;\longrightarrow_2;Z2\rangle$

2.5.3.1 Kwong criterion (1)

$(K2 \subseteq K1) \wedge (Z2 = Z1)$

From the definition of complete states in Section 2.1.3.1 (p. 44) and the assumption of Mn-reducibility defined by Figure 22 (p. 48), the Kwong criterion follows immediately. The set of basic states of the reduction is a subset of the set of basic states of the original and the alphabet likewise. These two components make up the complete state. Furthermore the definition of Mn-reducibility also defines that the set of initial states should be equal which is exactly what the criterion says.

2.5.3.2 Kwong criterion (2)

$$\forall(z_1 \in Z1)\forall(k_1 \in K1)\text{if } z_1 \xrightarrow[1]{*} k_1 \text{ then}$$

$$\exists(k_2 \in K2)\left(k_1 \xrightarrow[1]{*} k_2 \wedge z_1 \xrightarrow[2]{*} k_2\right)$$

This criterion is at the heart of what reducibility amounts to also in Mn-terms. If there is a path from an initial state in P1 to some complete state k_1, there should be a path in the reduction P2 from the same initial state to some state k_2 which is reachable from k_1 in P1.

According to the definition of reducibility in Figure 22 (p. 48) we know that from every initial state the sets of leaves are the same for the original P1 and the reduction P2. Any complete state k_1 in P1 reachable from an initial state z_1 is on a path to a leaf state (see Figure 21 (p. 47)) due to progress defined in Figure 25 (p. 50). That very leaf state is also reachable from the initial state z_1 in P2 since the set of leaf states are equal according to definition of reducibility. The leaf state thus reached is the state k_2.

2.5.3.3 Kwong criterion (3)

$$\forall q, r \in R_2(Z2) \text{if } q \longrightarrow_2 r \text{ then } q \xrightarrow{+}_1 r$$

If q and r are in the reachable states of the reduction and if r is reachable in one step from q in the reduction P2, then r should be reachable in a finite, positive number of steps in the original P1.

Any reachable state q in P2 is a stable state, one transition in P2 corresponds directly to a series of execution steps in P1 following the reduction algorithm. The algorithm executes the external signal (which is also executed by P2) and then stabilizes. The resulting stable state is by definition the state of P2.

2.5.3.4 Kwong criterion (4)

$$\forall q, r \in R_2(Z2) \text{ if } q \xrightarrow{*}_1 r \text{ then } q \xrightarrow{*}_2 r$$

If q and r are reachable states of P2 and r is reachable from q in P1, then r should also be reachable from q in P2.

Both q and r must be stable states since they are reachable in P2, but r need not in general be a leaf state (i.e. with no external input). If r has external input, then q must have the very same external input as the tail of its own external input since the external inputs are only consumed and never produced. The external input of r is of no significance for the path between q and r, therefore we may without loss of generality eliminate the external input of r from both r and q such that r is a leaf state.

From the proof in Section 2.5.1 (p. 73) we have the following general result:

$\forall p \in S2, \forall e_1 \ldots e_n \in E^n \bullet L_{P1}(p;e_1 \ldots e_n;ø;ø) = L_{P2}(p;e_1 \ldots e_n;ø;ø)$. In terms of the transition systems this means that the leaf sets of any complete state of P2 is equal to the corresponding leaf set of P1.

What the Kwong criterion (4) says is that when r is in the leaf set of q relative to P1, then r is in the leaf set of q in P2 as well. This is exactly what we concluded above.

This Kwong criterion has a stronger version which assumes that the leaf state has been reached in P1 with at least one step then the leaf state should need at least one step in P2 as well. In our Mn-reduced systems this holds always since P1 must execute an external signal to leave q and the same signal is executable by P2.

We conclude that Mn-reductions fulfill all Kwong criteria and therefore an Mn-reduction is also a Kwong-reduction. QED.

2.6 Basic pragmatics

As we have shown through our example process D which will be summarized in Section 2.6.3 (p. 79), the Mn-procedure is not certain to succeed. There are three points encountered so far where pragmatics and auxiliary techniques should supplement the Mn-procedure to make the Mn-approach more usable:

1. Eliminate the unreachable nodes.
2. Make the Mn-procedure terminate each generation.
3. Make the Mn-procedure terminate the series of generations.

2.6.1 Unreachable nodes

Later we shall introduce different kinds of remedies:

1. introduce a **save** construct to force the sequencing of the inputs,
2. define erroneous transition as a special escape criterion,
3. prove that the state is unreachable from the initial state.

In order to be able to prove our example process D confluent and reducible we show here how backwards execution can be applied to prove that the state is unreachable from the initial state.

2.6.1.1 Unreachability of the example process D

The state 3 of Table 2 (p. 58) is non-confluent by stabilization as shown in Figure 45 (p. 77).

$$((2;\emptyset;0;\emptyset),(2;\emptyset;\emptyset;3)) \xrightarrow{((\emptyset;0;\emptyset),(\emptyset;\emptyset;3))} ((0;\emptyset;\emptyset;\emptyset),(2;\emptyset;\emptyset;3))$$

Figure 45: Stabilizing state 3

At this stage we will try to prove confluence on the original process D by performing backward execution from the problematic (simple) complete state $(1;2\theta;0;\psi)$ to show that it is not reachable from the initial state $(0;\gamma;\emptyset;\emptyset)$.

Since the external input is not produced by the process itself, the last transition must have produced the internal signal “0” (or nothing) and ended in state “1”. The only possibility is the transition consuming the external signal “2” in state “1”. We thus have the following state as the next to last state before the problematic one: $(1;22\theta;\emptyset;\psi)$. From

this state backwards there is no road! We conclude that (1;2θ;0;ψ) cannot be reached from initial state and that the state must really be (1;2θ;0φ;ψ) where φ is non-empty and the state is simply sequence permuted rather than non-confluent.

We have now considered all states that were evaluated to non-confluence, and we have a profile for the process D which consists of 4 sequence permuted situations (of which two are similar) and 2 confluent situations.

We continue with the branch "3" where we quickly realize that the set of reachable states from "3" and the next generation alphabet is the same as with branch "1" shown in Table 4 (p. 64).

Table 6: Execution of M_1 from state 3

#	$R_1[T_3]$	$A_1[T_3]$	Z_1 element	Category
3	{(0,0),(1,1),(2,2)}	{((ø,1,ø),(ø,1,ø))}	base=(2;ø;φ0;ø) (2;ø; φ;3)	
31	(0,0)	gen. change	(1;ø; φ'1;ø,ø) (0;ø; φ';3,ø)	similar to 12
32	(1,1)	gen. change	(2;ø, φ';ø,ø) (1;ø; φ';3,ø) (Section 2.6.2.1 (p. 78))	non-confluent?
33	(2,2)	gen. change	(0;ø; φ';ø,ø) (2;ø; φ';3,ø) (Section 2.6.2.1 (p. 78))	non-confluent?

2.6.2 General invariants

Sometimes there are invariants which form the base of the specification. These should definitely be made explicit, and can as such be used favorably during the Mn-approach as auxiliary information about reachability.

2.6.2.1 Auxiliary information used in analyzing example process D

We observe that there are two problematic situations in Table 6 (p. 77). Regarding situation "3" we have already proved manually by backward execution that the internal queue portion φ must be non-empty.

Even though φ is non-empty the corresponding φ' need not be non-empty, but here this is the case since to reach basic states "1" and "2" from initial state "0" it is necessary to pass through the transition which produces the internal signal "1" when the signal "0" is consumed in state "0". This is the only way to leave state "0". Thus φ' must contain at least one element of A_1. Thus we conclude that situations "32" and "33" are only state different and not non-confluent (yet).

We continue execution within M1 of branches "32" and "33".

Table 7: Execution of M_1 from states 31, 32 and 33

#	$R_1[T_3]$	$A_1[T_3]$	Z_1 element	Category
3	{(0,0),(1,1),(2,2)}	{((ø,1,ø),(ø,1,ø))}	base=(2;ø;φ0;ø) (2;ø; φ;3)	

Table 7: Execution of M_1 from states 31, 32 and 33

31	(0,0)	gen. change	(1;ø; φ'1;ø,ø) (0;ø; φ';3,ø)	similar to 12
32	(1,1)	gen. change	(2;ø, φ';ø,ø) (1;ø; φ';3,ø)	state different
32-1		((ø,1,ø),(ø,1,ø))	(0;ø; φ";ø,33) (0;ø; φ";3,3)	confluent
33	(2,2)	gen. change	(0;ø; φ';ø,ø) (2;ø; φ';3,ø)	state different
33-1		((ø,1,ø),(ø,1,ø))	(0;ø; φ";ø,333) (0;ø; φ";3,33)	confluent

2.6.3 Concluding the analysis of example process D

2.6.3.1 The final states

Now finally we must consider situation "5". As the evaluation is sequence permuted, we change generation and again find the same set of reachable basic states and alphabet.

Table 8: Generation change of M1 from state 5

#	$R_1[T_5]$	$A_1[T_5]$	Z_1 element	Category
5	{(0,0),(1,1), (2,2)}	{((ø,1,ø),(ø,1,ø))}	base=(0;ø;ø;3) (0;ø;0;ø)	

We recognize that the base of situation "5" is similar to the base of situation "3" when seen from M_1 since the set of reachable basic states is the same and the rest of the base is also the same. The argument of the non-emptiness of φ' holds equally well. Thus the whole new generation of situation "5" is similar to that of situation "3" since both reachability set and alphabet are the same.

Since situation "6" is similar to situation "5", we may conclude that process D is confluent!

Finally the conclusion is that process D is reducible. The reduction gives the process D' shown in the Figure 24 (p. 50).

2.6.3.2 Lessons learned

Then, what can be learned from this example? We have taken an intricate process with feedback (internal signals) and shown that it is reducible. The proof could not be performed automatically, but required two manual interventions. Firstly we needed to prove that the non-confluent state (1,2θ,0,ψ) could not be reached. Secondly we needed to prove that the sequence φ' in M_1-state (2;ø, φ';ø,ø) (1;ø; φ';3,ø) had to be non-empty.

The manual proofs were simple in this case, but the necessity of manual intervention is always worrying as it is more time consuming and less maintainable than purely automatic proofs.

The question then is how the process could have been improved in order to obtain a more easily proved reducibility without losing the major aspects of the process. In this example it is hard to know what the major aspects are, but the confluence can be more easily established if we saved the external signal “2” in states “1” and “2”.

2.6.4 Basic pragmatics of determining progress

In this thesis we shall not go in great detail into the subject of progress. A lot of work has been done in this area to prove termination for different classes of programs. Since systems of CFSMs have the power of a Turing machine [13], it is also the case that termination (and thus progress) cannot in general be determined. Related to this result is also the result that the reachability of a specified complete state cannot in general be determined [48].

Still we expect that common systems are such that termination can be determined with a moderate effort. We have identified three classes of mechanisms that are used to ensure progress in the class of systems that we are dealing with.

1. The *signal ordering criterion*.
2. Progress by *fairness*.
3. Progress may also be assured through *timers*.

We shall go through these mechanisms one by one.

2.6.4.1 The signal ordering criterion

If there is a strict partial order on the signals defined by the transition table of the process such that every transition produces signals of lower value than it consumes, then every execution path will terminate.

Our example process D in Section 2.2.3 (p. 49) is a CFSM where there is an order among the signals: 2 > 0 > 1 > 3. Thus we may conclude that any execution path from any complete state of process D will terminate.

As we may consider any process as a rewrite system (see Section 1.6.2.3 (p. 33)) it is reasonable also to look for assistance in the search for termination of rewrite systems [37].

Very often the signal ordering criterion covers most situations in a process, but there are a few potential loops. Then it suffices to apply other approaches to these loops.

2.6.4.2 Progress by fairness

Fairness is a way to specify that certain choices cannot be made infinitely many times in succession. A fair die must show a 6 sooner or later.

We shall introduce assumptions of fairness in a number of places. We have already assumed fairness whenever there is an internal input ready to be consumed. It shall not be ignored forever as defined in Section 2.1.2 (p. 42). This means that there cannot be an infinite number of external signals consumed before the ready internal input signal is consumed. This ensures the progress of all specific input signals, but it does not eliminate the possibility of loops.

We shall in Section 3.5 (p. 97) introduce explicit fairness in non-deterministic decisions, and this may be used to express progress even when unbounded non-determinism is present. A typical example is a loop which may iterate any number of times, but not infinitely.

2.6.4.3 Progress through timers

In real systems one cannot always rely on the continuous correctness of all parts of the system. That the system shall not remain in deadlock or livelock situations is secured by guarding timers which exit from situations where the wanted return has not arrived.

Guarding timers may also break an eternal loop on internal signals, but it will then introduce a dash of non-determinism which we shall have to cope with. More about this when we handle timers in Section 3.7 (p. 119).

2.6.5 The termination of the Mn-procedure

The Mn-procedure may not terminate even when the system under analysis is progressive as pointed out in Section 2.4.7 (p. 69).

Will the Mn-procedure terminate or may we change generations forever?

The Mn-procedure applied to example process G shown in Figure 41 (p. 70) does not terminate the changing of generations.

We shall argue in Section 5.2.2 (p. 193) that in practice it is not necessary to consider systems with more than very few generations.

The proof of the algorithm is not dependent on when generations are changed. Heuristics may be utilized to find the most probable good places. Backtracking and further M_{n-1} execution before generation change is another approach.

In practice we will not expect the executions to be very long, neither within each machine (M_i) nor with different generation of machines. We suggest the following simple improvements which will make the Mn procedure terminate.

1. Max depth
2. Basic State Cycle detected

Max depth

The simplest remedy is to set a limit to the depth of executions within one generation and on the number of generations. A simple suggestion would be 5 levels within one generation and max 3 generations. In practice the limits could be even smaller. We shall argue more about this in Section 4.4.1.2 (p. 164). The Mn-procedure technique must be supplemented with other techniques if there seems to be a need for long executions.

Basic State Cycle detected

If the execution within one generation has reached the same basic state pair as before, this means that there is a cycle which is possible to repeat. If the cycle has left only external signals in equal proportions between the elements of the situation, we may conclude confluence on this branch due to external stuttering (Section 2.4.5.2 (p. 61)).

If the cycle has left internal (as well as external signals), we should change generation because further execution of this generation can never reach confluence since the loop may just be repeated.

2.7 Concluding the Basic Mn-procedure

In this chapter we have presented the basic Mn-procedure. Our starting point is a very basic SDL system consisting of only one process which had one external input and one external output channel. It had an internal channel to communicate asynchronously with itself, too.

We presented an idea of reducibility which was based on two requirements, progress and confluence. In this thesis we concentrate mostly on confluence, but progress is a prerequisite for our procedure to determine absolute confluence – the Mn-procedure.

We show that it is sufficient to prove absence of minimal non-confluence patterns to prove absolute confluence. A minimal non-confluence pattern has only one external signal. The Mn-procedure compares two branches of execution, firstly to execute the (only) external signal first and then some internal signal, secondly to execute the internal signal first and then the external signal. The resulting pairs of complete states constitute the initial set of nodes of the transition system M0. The continuation of M0 is defined through the execution of internal signals.

Unfortunately we find that it is not sufficient to pursue only the M0 transition system and we define a generation change giving rise to transition systems on higher levels called M1, M2 etc. and in general Mn.

We apply the Mn-procedure to an example, the process D which is reducible, but where the reducibility is not simple to spot ad hoc from the definition. We find that the Mn-procedure is not fully sufficient to prove the reducibility. We must use auxiliary, ad hoc techniques to prove that some of the encountered complete states are actually unreachable. This highlights the fact that the Mn-procedure determines absolute confluence, while we are usually interested in (plain) confluence where only the *reachable* complete states need to be confluent.

The Mn-procedure may not terminate and at the end of the chapter we indicate a few simple pragmatically inspired remedies to ensure that the Mn-procedure does terminate for all interesting cases.

3 General Mn-procedure

A Maxim for Vikings

Here is a fact
that should help you to fight
a bit longer:

Things that don't act-
ually kill you outright
make you stronger.

3. General Mn-procedure

Having explained the principles behind the Mn-algorithm we want to see how we can generalize the approach such that the restrictions imposed in Section 2. (p. 41) can be relaxed. The restrictions were:

1. The external input sequence is finite.
2. The system consists of one process only.
3. The system contains one external input channel, one internal channel, and one external output channel.
4. The process is deterministic, meaning that given a basic state and a signal only one transition is possible. The transition contains no decisions leading to different nextstates.
5. There are no data variables in the process.
6. There is no save (no explicit permutation of signals).
7. There are no timers.

We shall see how the Mn-procedure must be modified to accommodate for relaxation of each of these restrictions. We conclude that the Mn-procedure is well suited for systems that are close to real SDL systems in the respect that the seven restrictions can be relaxed without abandoning the general approach of the Mn-procedure.

3.1 Infinite external input sequence

In Section 2.3 (p. 50) we defined that the external input sequences of the complete states were finite. Considering the fact that modern systems should be designed to execute infinitely, we will review our initial restriction in these respects and examine how the restriction of finitude can be relaxed.

Still even though modern systems may be designed to last forever, most practitioners will settle for less than eternity. For all practical purposes it suffices that a certain property holds for all finite external input sequences. Still studying infinite input sequences may provide us with more insight into the behavior of systems.

3.1.1 What challenges do infinite input sequences pose?

We defined in Figure 25 (p. 50) that the external input sequence should be finite. This is not necessarily a reasonable assumption in our modern world. There are many systems which should be designed to run forever (even though this is not a realistic ambition). A telephone switch should be made to receive and connect telephone calls infinitely.

That we restricted ourselves to systems with finite external input in Section 2. (p. 41) does not necessarily have to be interpreted as a synonym for the system having to terminate. We only need to assume that the system every once in a while “cools down” so much that the processing triggered by the consumed external signals can be assumed to have taken place before the next external signal is admitted. This still leaves an interesting class of systems.

Conversely we can argue that systems which do not have such stable situations infinitely many times during the processing of an infinite external input, are inherently instable. This could mean that we cannot ascertain that the size of the internal queues will stay below a given limit. Since our model of communication is strictly asynchronous, we cannot in general assert anything about limits to the signal queues since we do not reason about the speeds of the transition consumption and the frequency of external input signals.

Finally there is a class of systems, which we may call *time-dependent*, where the signals are purely dependent on time and not of some random user. Such systems include managing sensors which give a measurement at certain time intervals. This kind of system may or may not turn into stable situations. It is reasonable that such a sensor management system consists of a pipeline of processes such that there are internal signals present at any specific point in time.

We may conclusively classify the systems in these three classes:

1. *Stable systems* that every once in a while are in stable states;
2. *Unstable systems* that haphazardly have internal signals at any point in time;
3. *Time-dependent systems* that systematically have internal signals always.

Example of a stable system may be local electronic locks which demand to return to an idle state before the next person can enter. Telephone switches can be seen as an example of unstable systems as we can be pretty sure that for a reasonable size switch there

is no safe time where it can be brought to a halt without the loss of internal communication. Finally sensory systems like ABS brakes and antispin may serve as example of a time-dependent system.

We covered stable systems in Section 2. (p. 41) and we shall now see how we could extend our scope to unstable and time-dependent systems as well.

3.1.2 Reducibility of unstable and time-dependent systems

Assume that we have a system which we have proved reducible according to Figure 22 (p. 48). We then have an original system and a reduction. What is the relationship between this original system and its reduction if the external input is not finite, but infinite?

To compare the two versions of the system, we assume that they are executed in parallel synchronized by every consumption of an external input.

Firstly we realize that at any synchronization point the external output from the original must be a prefix of the output from the reduction. This is clear because we could just stop the execution at this point and have a finite input. For finite input, confluence is certain and the reduction has finished its execution while the original may still have some internal signals left to execute, but that execution cannot change the output already output.

Our next concern is whether we are always certain that the original will “catch up” with the reduction. By “catching up” we mean that the original will always reach an output which the reduction produced up to some earlier point. We shall see that this is *not* the case.

A counter example is given in Process G shown in Figure 41 (p. 70). If G is a time-dependent process such that for every consumption of an internal signal in state T, the next signal to be consumed in state S is an external input. If we also then make sure to consume an internal signal in state T again, the signal consumed will be the same signal as produced before and the signal is reproduced again. The queue of internal signals will keep growing.

$$(S;e_1e_2e_3\ldots;\emptyset;\emptyset)\xrightarrow{e}(T;e_2e_3\ldots;a_1;\emptyset)\xrightarrow{a}(S;e_2e_3\ldots;a_1;\emptyset)$$

$$\xrightarrow{e}(T;e_3\ldots;a_1a_2;\emptyset)\xrightarrow{a}(S;e_3\ldots;a_2a_1;\emptyset)\xrightarrow{e}\ldots$$

Figure 46: Infinite consumption of internal signal

We have for the execution graph shown in Figure 46 (p. 85) that internal signals are always handled in state T while external ones come in at state S. This means that the internal signals never will manage to produce the external output which only takes place in state S.

Consequently we have a reduction (Figure 42 (p. 71)) which produces external output as a sequence of z’s, while the original system G produces no external output at all!

3.1.3 Infinite Progress

Informally we mean by progress that the processing of the external inputs should not halt and that the processing of one external signal should eventually terminate. In everyday life a process may be compared with a bureaucracy. Whenever we send a letter (i.e. external signal) to a bureaucracy we want it to respond eventually and not that the bureaucracy should get tied up in its own red tape (i.e. internal signals). Since no single letter may activate the bureaucracy for ever, any specific letter will eventually be handled as long as the bureaucracy cannot stay idle while letters are pending.

More precisely we can formulate this by marking all internal signals with the external signal which originally produced it. No execution branch should have infinitely many states with internal signals marked by some particular external signal.

We summarize our definition of infinite progress in Figure 47 (p. 86).

Given a process P=<S;C;Z;T>

1. Assume that every individual external input has a unique identification.
2. In the initial complete states of the transition system, Z, annotate the external input signals with their identification.
3. For all transitions , annotate the resulting complete state by annotating all signals produced by the transition with the annotation of the consumed signal.

P is *infinitely progressive* **iff**

For all individual signals e and for all execution paths q in P there is only a finite number of complete states in q with signals annotated with the identification of e.

Figure 47: Infinite progress

Infinite progress is implicitly based on our basic model fairness assumption that every signal received will eventually be consumed (see Section 2.1.2 (p. 42)). Thus an internal signal which has been produced cannot be overtaken by external input signals for ever.

We see that the modified definition of progress in Figure 47 (p. 86) coincides with our original definition of progress (Figure 25 (p. 50)) in the cases for finite external input sequences. Thus infinite progress implies finite progress.

Our example process G studied in Section 3.1.2 (p. 85) was not infinitely progressive which can be seen from the counterexample in Figure 46 (p. 85) and this is what made the reduction behave significantly differently from the original. Infinite progress ensures that the original is faithful to the reduction even for infinite external input.

The signal ordering criterion defined in Section 2.6.4.1 (p. 80) ensures infinite progress.

3.1.4 Concluding infinite complete states

In this section we have shown that to prove reducibility for all finite input results in a reduction which may not always be fully faithful to the original in unstable infinite systems.

We gave an example where the reduction outputs, while the original runs the risk of never producing any external output.

By strengthening the requirement for progress in the infinite case such that the effects of any individual stimulus is completed within a finite number of transitions, we get that reductions are sufficiently faithful to the original.

3.2 Multiple channels

In this section we shall discuss the generalization of the Mn-procedure and of reducibility provided there are more channels. This turns out to pose few problems. In fact the generalization may in some cases be an advantage.

We have in our simple model already two different input channels, one external and one internal one. Correspondingly we have two different output channels, one external one and one internal one. The latter is the same as the internal input channel.

Our basic model is that if there are signals available on both channels, it is an arbitrary choice whether the first signal of the internal channel or the first signal of the external channel, should be consumed first. Readers acquainted with SDL will know that SDL processes have only one input port (i.e. input sequence). This means, however, simply that the arbitration takes place at the entry of the input port. If we consider the input port an integral part of the process there is no difference between our multiple input sequence model and the SDL model as discussed in Section 2.1.2 (p. 42). Similar to the traditional SDL model we have that the process cannot explicitly specify from which channel it wants its input.

The challenge of multiple channels is that the degree of freedom increases. There are more choices during an execution since there are more input channels. There is more flexibility on output because there are more output channels. It could be conceivable that the results of our restricted model could not be transferred to the more general case. It turns out, however, that this is not the case. The results of the restricted analysis is mostly transferable to the more general context.

3.2.1 More external input channels

Having more external input channels actually adds nothing to our model since we have assumed that external signals can occur at any time. Our aim is to show that the choice between an external and some internal signal is insignificant wrt. the final result of the computation. We remember that we showed that the Mn-procedure need only check for the existence of a minimal non-confluence pattern which had only one external signal (Section 2.4.3.5 (p. 55)). Which channel this external signal comes from is not significant. In fact it is equivalent to considering all external signals on one channel.

However, what is external to one subsystem, may be internal to an enclosing system. On that level the distinction between the very same channels is significant as we shall cover in more detail in Section 3.3 (p. 90).

3.2.2 More internal input channels

The idea behind confluence is that all execution branches from a given complete state will result in the same set of leaf (stable) states. A generalized non-confluence pattern is a complete state from which not all execution branches lead to the same set of leaf states, but where each of the first level subtrees of the complete state are confluent. This is illustrated in Figure 48 (p. 88).

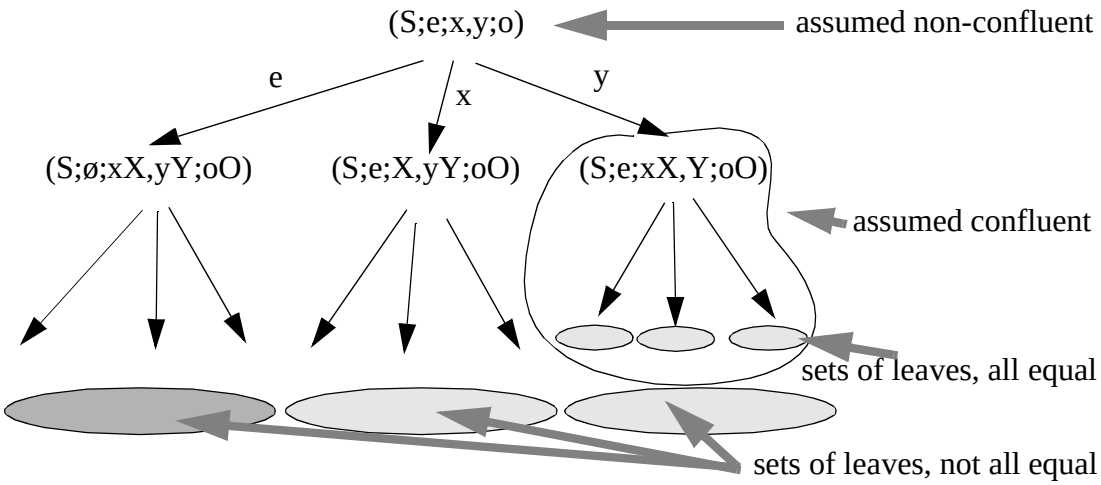


Figure 48: Generalized non-confluence pattern

The clue is again that the first level subtrees of the non-confluent state are assumed confluent. This means that the subtrees themselves can be executed in any order. This means that stabilization may consume the internal signals in whatever order suits the execution. This means that the input alphabet of the M0-transition systems is the union of the internal input signals of all input channel.

It is necessary, however, to consider all pairs of internal signals from different internal channels as starting points for the search for non-confluence pattern as well as all pairs of one external and one internal signal. It is quite possible that the non-confluence pattern occur due to the conflicting initiatives of two internal channels.

An example is shown in Figure 49 (p. 89).

We have two internal channels, one external input and one external output. We notice that the consumption of external input results in internal buffering and the subsequent consumption of the internal signals results in external output. The two different external inputs results in loading two different internal buffers (channels). This means that the four potential non-confluence patterns involving one external input and an internal input will all be confluent trivially since the output of consuming external input and output from consuming internal input are placed on different channels. The potential non-confluence pattern of one internal i and another internal j, which are on different internal channels, leads to a non-confluence since the result is either external output of uv or of vu. The non-confluent state is definitely reachable as the external inputs ef or fe will produce it.

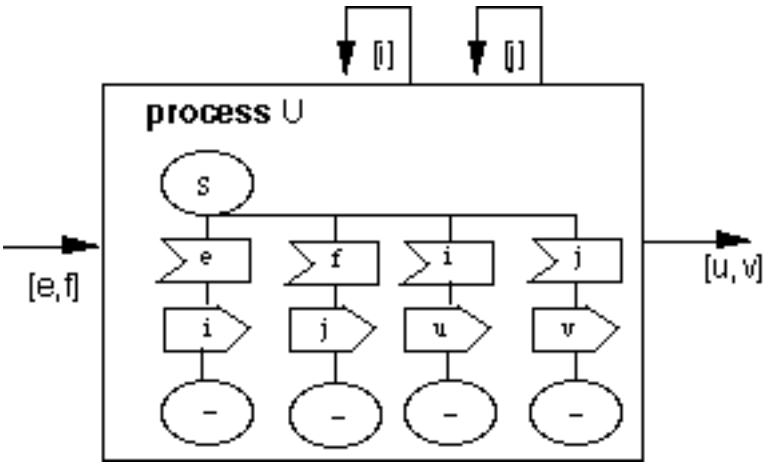


Figure 49: Process U with internal non-confluence pattern

Conclusively multiple internal channels result in having to check also the potential non-confluence patterns resulting from conflicts between two internal signals on different channels in addition to the potential conflicts between an external and an internal signal. This means that the number of potential non-confluence patterns will increase with the number of separate internal input channels.

3.2.3 Multiple output channels

Multiple output channels make less conflicting situations than only one output channel. In Process U of Figure 49 (p. 89) we have that the problem appears when the consumption of i and j both are output to the same external output channel. If we had had two external output channels as well, one for u and one for v, the problem would have been non-existing and the process confluent since sequencing between channels is insignificant for the final state.

3.2.4 Concluding multiple channels

Having more channels than the restricted amount assumed in Section 2. (p. 41) imposes almost no complications for the use of the Mn-procedure. The complete state must be extended to include one element per channel (buffer), but this is trivial.

Multiple internal input channels mean that it is necessary to check all potential non-confluence patterns involving two internal channels as well as the patterns involving an external and an internal channel. The M0-alphabet, though, is not affected as it is the union of all internal signals.

The signals on internal channels can be executed in any order as long as the order on each channel is preserved, since the generalized minimal non-confluence pattern has all subtrees confluent. This means that the generalized reduction algorithm can also execute the internal signal (which is first on a channel) that suits the algorithm best.

Multiple output channels makes it easier to conclude confluence since more output channels mean greater freedom and independence of pieces of the output.

3.3 Multiple processes

Up until now we have considered processes which have been described by one SDL process graph. This is not a very realistic system. Internal channels will normally not appear from an SDL process to itself, internal channels are normally between components of a larger system.

Here we show that a system of multiple processes can be seen as one process, but that a non-confluence pattern can only occur within one component.

3.3.1 Definitions

We need a few more definitions to be able to talk more effectively about multiple communicating processes. Some of these concepts have already been informally introduced in Section 2.1.2 (p. 42).

3.3.1.1 System

A *system* is a set of *components* (Section 3.3.1.2 (p. 90)), which communicate asynchronously via channels. A system in our terms corresponds closely to an SDL system.

3.3.1.2 Component

A *component* is either a *block* (Section 3.3.1.3 (p. 90)) or a *process* (Section 2.1.3.1 (p. 44)).

3.3.1.3 Block

A *block* is a *system* (Section 3.3.1.1 (p. 90)) on a lower nesting level. Sometimes we also refer to blocks as *subsystems*.

3.3.2 The basic model and the combined CFSM

We assume that we consider an SDL block as our unit of observation. It contains a set of processes and a set of channels between these processes. Each internal channel should have one process on either side. Each external channel should have a process on one side and the environment on the other.

Thus our model implies that where SDL allows merging channels, we have to separate the channels all the way to the receiving processes.

We want to transform the block such that it becomes a process. Thus we reach the following definition of the block as a CFSM <S;C;Z;T>:

1. The set S of basic states is the Cartesian product of the sets of basic states for each process.
2. The alphabet C is the cartesian product of all signal sets of all channels. This can easily be divided in an external input set, and internal set, and an external output set of channels.
3. The initial set Z of complete states is derived from the tuple of initial basic states of the processes and the sequences of external input.

4. The input alphabet A contains the elements of C which have only one non-empty input channel, and only one element on that channel.
5. The set of complete states K is simply defined as before K=S x C*.
6. The transition table T is derived from the individual transition tables of the component processes. We assume that a signal also contains information from which channel it comes such that we may distinguish between signals of the same type on different channels.

We may then in principle calculate the combined CFSM explicitly and work from there. It is more practical to consider each component separately as far as that works because the combined process meets state explosion very quickly.

3.3.3 Interleaving semantics

Let us first convince ourselves that the combined CFSM behaves exactly as the block it is derived from. While the block may have several transitions execute in parallel, the combined CFSM can only have one transition at one point in time. Does this make a difference? The only difference we accept as a real difference is if there is a way the block can reach a complete state which the combined CFSM cannot reach or vice versa. To determine that two actions on different processes are actually concurrent, is in practice impossible if the processes do not share a common clock. We can only observe that two transitions have taken place approximately at the same time when their output is merged in a way that makes the output different from the situation where one transition executes before the other.

With our basic model (Section 2.1.2 (p. 42)) the output from two different processes are merged only when there are two channels going into a merging process one from each of the concurrent ones. This means that the output from the concurrent processes are placed on different channels and thus considered to be independent relative to the merging process. Thus all possible interleavings of the signals will be considered also in the combined CFSM.

We conclude that the combined CFSM corresponds to the original block with respect to the complete states.

3.3.4 Piecewise execution of the Mn-procedure

Assume that we have constructed the combined CFSM and we start performing our Mn-procedure. Let us assume that we have been able to establish progress of the CFSM and that we are determining confluence.

Any potential non-confluence pattern involves two signals on two independent channels. If these two signals are handled in two distinct component processes of the block, we know by definition of our basic model that the output from handling the two different signals cannot occur on the same channel. Thus confluence is assured directly for this kind of potential non-confluence patterns.

The simple conclusion is that a non-confluence pattern can only occur within one individual component process meaning that both the involved signals must be handled by the same component.

This simplifies the search for non-confluence patterns considerably since most of the theoretical non-confluence patterns of the combined CFSM can be eliminated at the outset since the signals are handled in different processes.

Projection

Our “piecewise” execution of the Mn-procedure is actually a projection of the Mn-procedure applied to the combined process. We notice which potential non-confluence pairs cannot ever produce non-confluence since they involve more than one component. Said differently the M0-procedure for a combined CFSM consists basically of taking the M0-procedure for each component process. Higher generations may similarly be projected. The M1-procedures involve the components which receive output from the process which triggered a generation change in its M0-procedure. Often the M1-procedures can be performed for one process at a time, but there are also (theoretical) cases where the set of processes receiving the M1 input, interact mutually and then the interacting set should be included all at the same time when performing the M1-procedure. Similarly an M2-procedure may in principle be even more involved.

Stabilization

The pure confluence search can be performed very much piecewise, but stabilization which is an important part of Mn-procedure often involves more than one process. Since the piecewise execution of the Mn-procedure is a projection of the Mn-procedure used on the combined process, it should be clear that stabilization is not only a linear execution. If we analyze non-confluence of a component process U which outputs to a component process V, stabilization of a state pair in U will involve executing V, and we must make a sequence of assumptions concerning the state of V. Thus a stabilization is in principle an execution tree.

Stabilization should, however, be performed breadth first since whenever the two elements of the node pair are equal, we may halt the stabilization. Furthermore we take advantage of the fact that the stabilization can be performed in any order wrt. which internal channel to choose signal from. Thus we execute all signals input to one process before going on to the next.

Example: UV

We shall give an example of piecewise execution of the Mn-procedure in Figure 50 (p. 92).

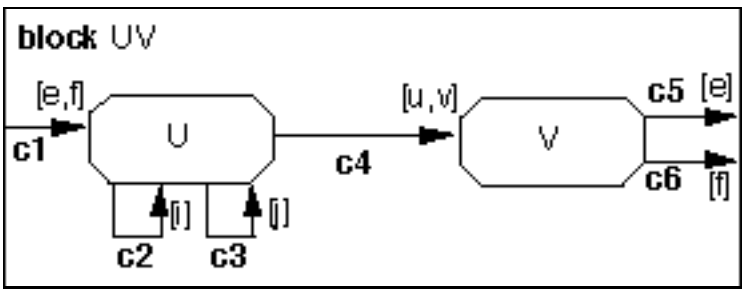


Figure 50: Block UV

We connect the process U shown in Figure 49 (p. 89) with another process V shown in Figure 51 (p. 93). We shall show that block UV is confluent by applying Mn-procedure piecewise.

Process V takes the internal signals u and v and produces external output e and f on separate channels.

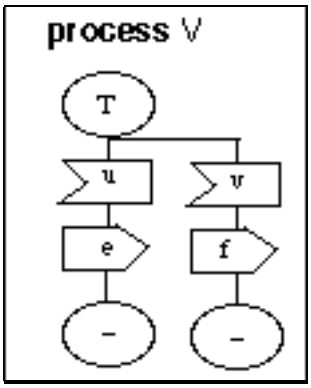


Figure 51: Process V

We apply Mn-procedure piecewise to procedure U. The M0-procedure is quite similar to what we did in Section 3.2.2 (p. 88), but when the difference in production of sequences onto c4 is encountered this is not a direct sign of non-confluence, but merely a sequence permutation which may be resolved when changing generations since c4 is an internal channel of block UV. The M1-procedure originating from potential non-confluence pattern (S;ø;i,j;ø) of process U, activates V. The output alphabet of M0 of U is {(u,u),(v,v)}. This is the input alphabet to M1. It is parallel. The state from which the generation change should take place is ((T;uv;ø,ø),(T;vu;ø,ø)) relative to V. Stabilization of this node must take place in V and leads directly to a confluent result.

The generation change leads to initial set of M1 equal to {((T;ø;e,f),(T;ø,e;f))} relative to process V (which is equal to the stabilization results).This is directly confluent.

To apply Mn-procedure to process V is trivial since V has only one input channel and no conflicts may arise. Thus block UV is confluent. It is progressive due to signal ordering criterion and therefore block UV is also reducible.

3.3.5 Progress

Progress in a system of processes is equal to progress within a process as we introduced it in Section 2.3 (p. 50). Normally there are some feedback loops which prevent the simple signal ordering criterion from holding. What is normally needed is to consider each of these cases individually.

Even though progress is not a major part of this thesis, we shall spend some time discussing three ways to ensure progress. We have already presented the *signal order criterion* in Section 2.6.4.1 (p. 80). In Section 3.5 (p. 97) we show how fairness in nondeterminism can ensure progress and in Section 3.7 (p. 119) we show how timers can prevent an infinite loop.

3.3.6 Concluding multiple processes

Since a system of multiple component processes can be seen as one process, the Mn-approach can also be used for systems.

Progress must be determined globally in the system by examining possible feedback loops.

Confluence may be determined piecewise meaning that non-confluence patterns are always in one component only. M0-procedure can be done one component at a time while higher generations may be more involved. Still the work needed in a piecewise approach is normally much less than transforming the system to a process and performing a total Mn-procedure from there.

3.4 Save

Saving signals to a more appropriate state is the only way SDL can express permutation of incoming signals. **Save** can be seen as suspending some transitions such that confluence more easily can be established. On the other hand the existence of saved internal signals makes it necessary to modify our notion of stable state.

The designer may also want to establish that the saved internal signals will disappear eventually. This problem modifies the concept of progress. To establish the disappearance of saved internal signals is the same problem as the reachability problem which is not determinable in the general case [48].

SDL has the property that all signals eligible to the process are legally consumed in any state of the process. This is not necessarily the case with other notations where finite state machines are described by graphs where not every possible combination of state and input signal is defined. Of course a major problem when implementing systems is that signals show up in states where there was no intention that they should arrive. What is the interpretation of this if there are no default transitions? To cope with undesired signal receptions, SDL has the option to save the signal for later. One may interpret this to mean that all the saved signals of a process are put in a separate save-queue parallel to the input port. When the process changes state the whole save-queue is inserted in front of the rest of the input port. Alternatively one may interpret a **save** as just ignoring the signal and taking the next. The saved signals are kept in the queue, but ignored when in a state which saves them. The latter interpretation suits our basic model best since we assume a queue for each channel.

In a state where there is a **save** construct, confluence is more easily established since there is no need to look for non-confluence patterns involving any saved signals since they cannot be consumed before the state has been left.

3.4.1 Stable states revisited

In Section 2.1.3.6 (p. 47) we defined a *stable state* as a complete state where all the internal input channels were empty. As we introduce saved signals, we run the risk of reaching complete states where there are only saved signals in the internal queues. Is this a stable state? At least it is certain that no more stabilization is possible as no more internal signals can be immediately consumed (since they are saved in this state). On the other hand the internal queues are not empty.

We decide to define two specializations of stable states. A *totally stable state* is a stable state where the internal queues are empty. This corresponds directly to the former definition. A *semi-stable state* is a state where no more stabilization can be performed, but where the internal channels are still not empty.

During confluence calculations, however, we may find states which are not totally stable as end results of stabilizations. When we introduce non-deterministic saves in Section 3.5.4 (p. 106) that we must be even more lenient with the stability of states.

3.4.2 Save and Progress

It is a question whether progress should mean that all stable states must be totally stable. Otherwise there are internal signals in the system which must originate from some external input and that input has not been fully handled.

At this point we should remind the reader that progress plays two slightly different role in our approach. Firstly it is a goal in itself that a system has progress which is synonymous with the system getting things done. Secondly we need progress for our Mn-procedure to function properly. The second aspect of progress is only to ensure that the execution graph of any complete state is finite.

When analyzing progress as an end in itself, we want to examine whether all saved signals will eventually disappear. We define *strong progress* to mean that all stable states are totally stable.

Progress is closely related to termination and to reachability both of which are in general not decidable. Our task may still be manageable for each individual practical case. Due to the theoretical complexity of the establishment of progress, we must accept that a full proof will require use of advanced proof techniques. That is not the topic of this thesis.

For the sake of the Mn-procedure it suffices to establish *weak progress*, which we shall define to mean that the execution graph of any complete state is finite. This definition is satisfied if stabilization ends in semi-stable states.

Weak progress suffices for the Mn-procedure to work since every stabilizing execution is finite and therefore the assumption that all complete states in the execution graph of a non-confluent state are confluent can be kept (Section 2.4.3.2 (p. 53)).

Weak progress is significantly simpler to establish than strong progress. We only have to ascertain that feedback loops do not run forever.

3.4.3 Save and confluence

We introduced **save** for the explicit purpose of controlling the order of signal consumption, which means that confluence should be more easily established.

Nevertheless we need to clarify a few points concerning the comparing of the two elements of the Mn-nodes. How different may two semi-stable states be before they are considered non-confluent?

It is perceivable that two non-identical semi-stable states are such that whatever signal consumed, either the signal is saved or both states are resolved into totally stable states which are identical. Should such a state pair indicate non-confluence?

The simplest approach, which we will adopt, is that two semi-stable states are considered confluent only if they are identical (with the possible modification of “glue” as presented in Section 2.4.5.2 (p. 61)).

During M0-execution, the only problematic situation is a potential non-confluence pattern where both signals are consumed from the starting basic state, but (at least) a **save** is invoked in the follow up state. This leads to at least one of the elements of the pair to have a **save**. All other situations are trivially confluent when **save** is involved.

3.4.4 Save and reducibility

If we have established weak progress, we know that the Mn-procedure can be applied to the system. If the Mn-procedure returns confluence, the system is reducible and the reduction algorithm can be applied.

If the reduction contains no basic states which correspond to a semi-stable state of the original system, we may also conclude that the system is strongly progressive. The reason for this is that any stable state reachable in the original system is also reachable in the reduction.

We explain this in more detail. Assume that there is a reachable stable state Q of the original system which is not in the reduction. Since Q is reachable, there is some complete state W of the set of initial states Z which has an execution path leading to the stable state Q. This initial state W is also an initial state in the reduction. We execute the reduction from W and because it is progressive, we reach some set of stable states L. The execution in the reduced process corresponds to an execution tree of the original system as a consequence of the reduction algorithm. According to the assumption Q is not a member of L. Then W must be a non-confluent state of the original system since there is one path leading to Q and other paths leading to L which does not include Q. But our assumption was that the system was reducible, and thus confluent. The conclusion must be that our assumption that Q is unreachable in the reduction cannot hold. All reachable stable states of the original are also reachable in the reduction.

Our Alternating Bit Protocol example Section 3.5.3.1 (p. 100) is shown to be reducible since it is weakly progressive and confluent, and the reduction Figure 61 (p. 105) shows that it is strongly progressive and that there cannot be any deadlock in the waiting states since the reduction includes no stable states with the waiting states as components.

3.4.5 Concluding Save

Introducing the SDL **save** construct makes it easier to obtain confluence, but we have to modify our notion of stable states.

The property of progress becomes slightly more involved as we distinguish between strong progress and weak progress. Strong progress means that only stable states with no internal signals (totally stable) should exist in the execution graph, while weak progress only requires that stable states should be semi-stable where also saved internal signals are present.

We show, however, that weak progress is sufficient for determining reducibility. Furthermore we show that strong progress can be deduced from reducibility and the fact that no basic states of the reduction originates from semi-stable states of the original. The Alternating Bit Protocol example shows this.

3.5 Non-determinism

The requirements for reduction are that the race conditions between external and internal signals (and between internal signals) should not be significant. Explicit non-determinism can be handled by a generalized Mn-procedure by talking about sets of states rather than single ones.

SDL has defined two different kinds of non-determinism, the anyvalue expression in decisions and the spontaneous transitions. We shall cover them both and then we shall introduce a couple of new mechanisms which extend SDL, but add expressiveness to our description of reduced processes.

3.5.1 Anyvalue expressions in decisions

Decisions with pure anyvalue expressions are the simplest form of non-determinism. The only change is that a transition does not necessarily result in one specific state. It may result in any of a set of states. This does not prevent the process from being reducible! Already in our definition of reducibility given in Section 2.2.1.1 (p. 48), we foresaw this and defined that the final set of leaves should be independent of the race conditions between channels with signals to be consumed.

Our Mn-procedure must be elaborated such that every complete state in our simple procedure becomes a set of states in the more elaborated one. Still our proofs and other extensions to the simplest model are not dependent on there being only one complete state as the result of every transition and the results may be generalized accordingly.

We give a simple example just to show how the execution of the Mn-procedure looks when generalized to sets of states.

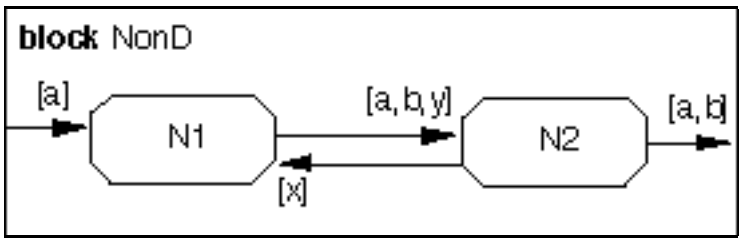


Figure 52: Block NonD, a reducible block with non-determinism

The block NonD (Figure 52 (p. 97)) does not do much sensible computing. It consumes an a signal in N1 which then produces either an a or a b, if it produces a b an x reply will come back from N2, but N1 will only respond by sending a y. In N2 the y and a signals will result in outputting either a or b. If the reader is confused by this informal specification, please take a look at Figure 53 (p. 98).

We show the computations of the Mn-procedure through a graph in Figure 54 (p. 98).

There is only one possible non-confluence pattern of N1 and no patterns for N2 since N2 has only one input channel. The initial set Z_0 of M_0 shows a pair of sets of states which we see is sequence permuted and we decide to change generation. The initial state of M_1 is obviously confluent since the input alphabet A_1 is parallel and the pair of state sets has equal elements.

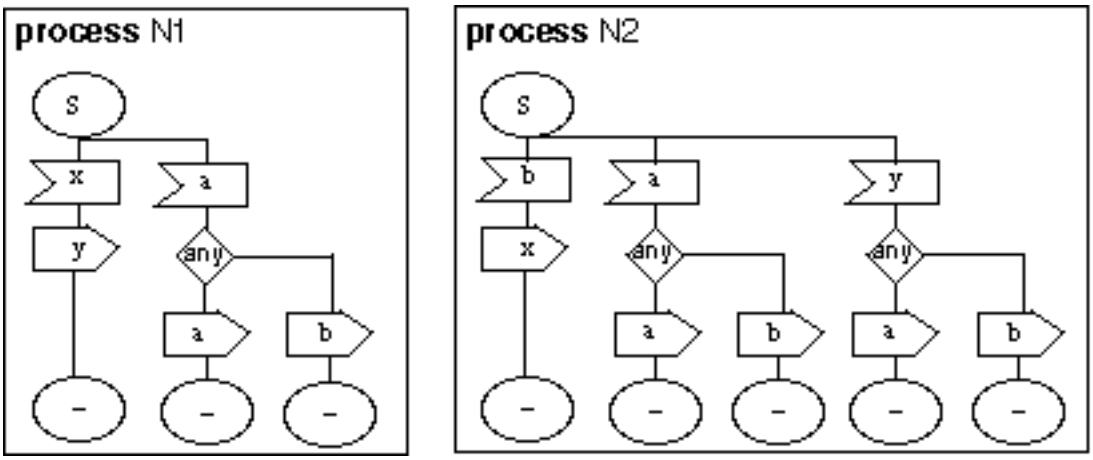


Figure 53: The processes of NonD

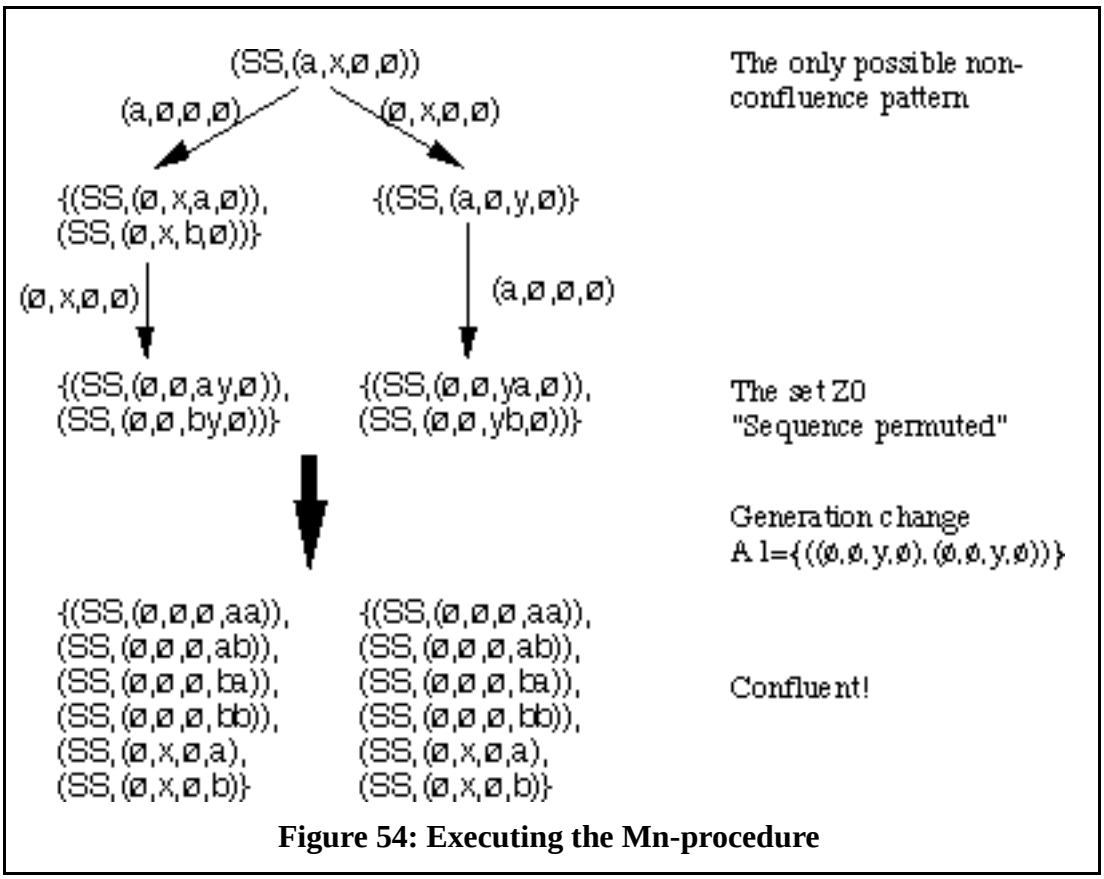


Figure 54: Executing the Mn-procedure

The evaluation of the nodes of the Mn transition systems becomes slightly more intricate. Stabilization may certainly determine non-confluence as easily as with the deterministic case. Conversely confluence is not so hard to spot either. The sets must be equal and the input alphabet parallel. The distinction between sequence permuted and state different may be slightly less obvious. We have also seen that the exact point on which to perform generation change can be moved as a result of heuristics and experience also for the simple case. When it becomes clear that further execution on this generation cannot succeed in establishing confluence, we change generation.

Changing generation is also somewhat more involved. Our example in Figure 54 (p. 98) is too simple to show the complication. The input alphabet of the new generation is the output alphabet of the former. The output alphabet is dependent upon the basic states on which the input has been applied. When there are several states as when non-determinism is involved, there will be several (possibly different) output sequences as well. We need to record one output element for each complete state in the set. Keep in mind that these elements are different only if the basic states of the complete states are different. This is the reason why the complication does not show up in Figure 54 (p. 98). It is important to keep the association between the states and the signals as shown graphically in Figure 55 (p. 99). Each element of the output alphabet is associated with at least

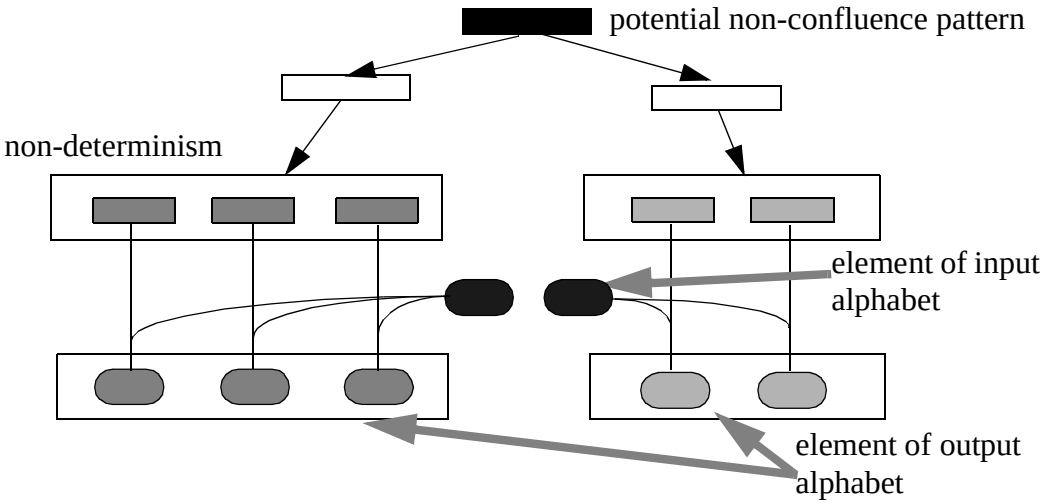


Figure 55: Symbol alphabets and non-determinism

one element of the input alphabet. Futhermore for every individual state of the non-determinism set of states, there is a symbol component within the output alphabet. Following a generation change the alphabet is applied as a pair of tuples to the pair of tuples representing the Mn-node.

3.5.2 Spontaneous transitions

Spontaneous transitions in SDL are transitions where the input symbol contains **none**. The transition may execute without consuming any signals at any time when the process is in a state where a **none**-transition is specified. We have two quite different approaches to this, either the spontaneous transition is considered triggered by an external event, or it is considered internal.

3.5.2.1 Spontaneous transition as externally invoked

We define that **none** is a signal type which comes on a special channel external to all possible enclosures. It is consequently considered exactly as any other external signal. In states where there are no spontaneous transitions this is considered to be equivalent to executing a default transition for **none**. A default transition means that the signal is merely consumed and no state change takes place.

Spontaneous transitions do not add much to the question of progress since **none**-signals cannot be produced by any process.

Concerning confluence, any internal signal must be insensitive to whether a possible **none**-signal arrives before or after its consumption. This is exactly the same as with normal external signals.

3.5.2.2 Spontaneous transition as internally invoked

Alternatively to the interpretation of spontaneous transitions as external signals is the interpretation which considers the spontaneity as internal. During an execution the process may either halt in the state with spontaneous transition, or it may continue along the spontaneous transition. This becomes exactly similar to interpreting a spontaneous transition as a non-deterministic decision (Section 3.5.1 (p. 97)) at the end of all transitions leading to a state with a **none** input. The non-deterministic decision decides between an empty branch and a branch corresponding to the body of the spontaneous transition.

It is possible that there is a cycle of (non-trivial) spontaneous transitions. The interpretation of spontaneous transitions as internally invoked will then yield an infinite set of states. For an automatic application of the Mn-approach, infinite sets are not very practical. Theoretically the infinite sets may not pose any extra problems.

3.5.2.3 Concluding spontaneous transition

Both interpretations affect our notion of stability in states. A state which looks stable, but which has a **none** input is not absolutely stable after all.

Both interpretations allow reducibility provided their respective confluence criteria have been met. The reduction algorithm in the two different cases will give two different reduction results since the reduction algorithm applies the execution interpretations. A reduction under the external interpretation of spontaneous transitions will include spontaneous transitions, while a reduction under the internal interpretation may appear without spontaneous transitions, but with non-deterministic decisions.

3.5.3 Fair Anyvalue-expressions

While the standard SDL decision does not assume any kind of fairness, we find that many times fairness is what you would like to have in order to use non-determinism to terminate a loop in the execution. *Fairness* is defined as a restriction on some infinite behavior according to eventual occurrence of some events [49].

A perfectly valid implementation of the anyvalue expression in SDL is to pick one value from the start and stick to that value every time the anyvalue expression is executed for that sort[1]. Thus the anyvalue expression does not ensure random drawing when it is applied.

3.5.3.1 Alternating Bit Protocol

The *Alternating Bit Protocol* example shows how fairness constructs can be used to ensure progress. The example was introduced by Bartlett et al. in [3]. It has later been used as a simple, but illustrative example of a provable protocol for unreliable communication. We shall use it to show that non-deterministic decisions with fairness can be

1. “Sort” in SDL means “datatype” in programming languages.

used to resolve infinite loops to achieve *unbounded non-determinism*. That the non-determinism is *unbounded* means that it is not possible in advance to determine an upper bound for the number of iterations of the loop.

The example is a protocol of full-duplex transmission over half-duplex links. In 1969 it was important to point out that only one control bit is needed to ensure reliable communication assuming that all errors in the transmission are detected. Our aim is to show that by a small extension of SDL, we can show that the protocol is progressive. We may also show that the protocol is confluent (under some reasonable conditions) and thus the protocol is reducible. The reduction shows trivially the correctness of the protocol.

The structure of the protocol is given in Figure 56 (p. 101).

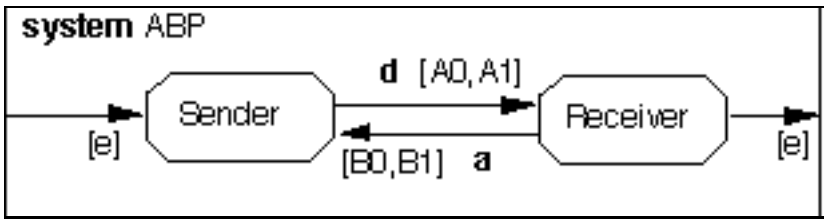


Figure 56: Alternating Bit Protocol structure

We have actually abstracted away the real contents of the message and show only the control information. The signals between the Sender and the Receiver represent the value of the control bit. A0 and A1 are values 0 and 1 of the bit on the message from the Sender to the Receiver, while B0 and B1 are values 0 and 1 of the bit on the acknowledgment from the Receiver to the Sender.

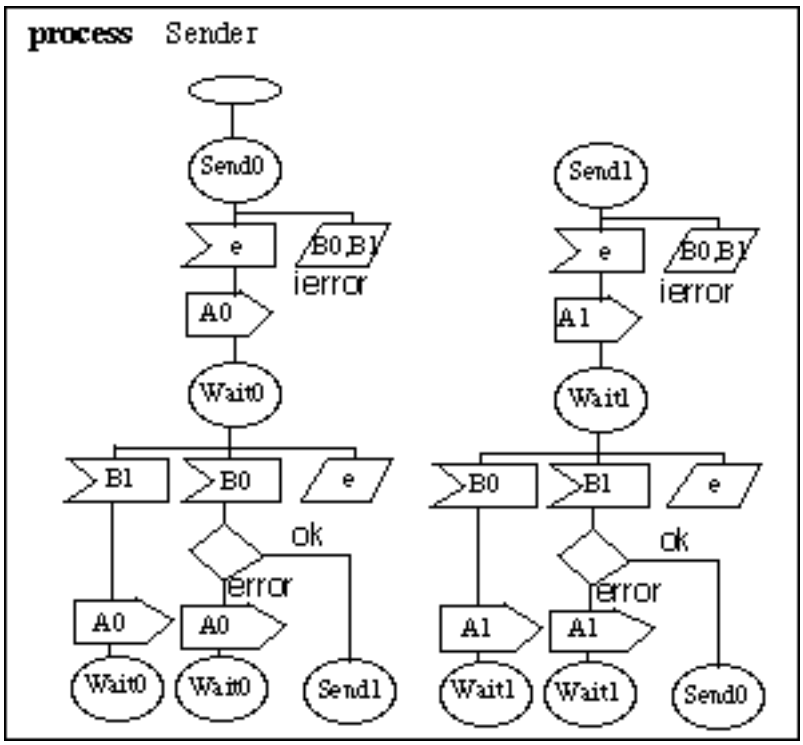


Figure 57: Sender of Alternating Bit Protocol

The messages are never lost, but they may be corrupted. If a message is corrupted, this will be discovered.

The Sender (Figure 57 (p. 101)) will test for the correctness of the returned acknowledgment. Whenever it concludes that it is wrong, it will ask for a repetition of the acknowledgment.

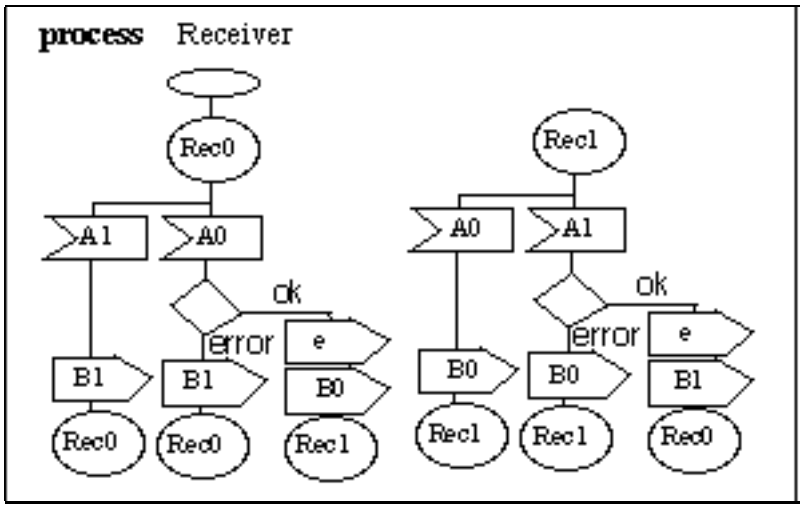


Figure 58: Receiver of Alternating Bit Protocol

The Receiver (Figure 58 (p. 102)) will test for the correctness of the message sent. Whenever it concludes that it is wrong, it will ask for a repetition of the message.

The big issue here is the progress. We can quite easily conclude confluence since the saves eliminate the conflicts which may occur.

In order for the protocol to terminate, we need that both the message from the Sender to the Receiver is checked ok and the acknowledgment from the Receiver to the Sender is checked ok. If things go very wrong, it is even possible that the acknowledgment from the Receiver which is meant to ask for a retransmission of the message is not received by the Sender properly, but the response by the Sender is still adequate as the only sensible thing to do is to retransmit.

Let us look at the possible eternal loops of the protocol.

1. The original message is not received properly. This gives the infinite signal sequence $e \rightarrow A0 \rightarrow B1 \rightarrow A0 \rightarrow B1 \rightarrow \ldots$ which may be terminated only by the check finally giving ok.
2. The acknowledgment of a well received message is corrupted. This gives the infinite signal sequence $e \rightarrow A0 \rightarrow e, B0 \rightarrow A0 \rightarrow B0 \rightarrow \ldots$ which may be terminated only be the check finally giving ok.

There are two additional cases that are symmetric to the above two where the starting state of the Sender is Send1 instead of Send0.

The first loop is only dependent upon the Receiver checking ok sooner or later, while the second loop is dependent only upon the Sender checking ok sooner or later. The SDL anyvalue in the decision cannot ensure that this happens. We need a construct that ensures that whenever the decision is visited infinitely many times one (or more) spec-

ified alternatives will appear infinitely many times. We need a *fair decision*. Often for pure specification purposes we are not so interested in the sort of the decision or the values of the alternatives.

3.5.3.2 Fair decision

We shall specify a construct which is a fair decision which has arbitrary sort and only specifies that some alternatives will be infinitely-often chosen in an infinite sequence of choices.

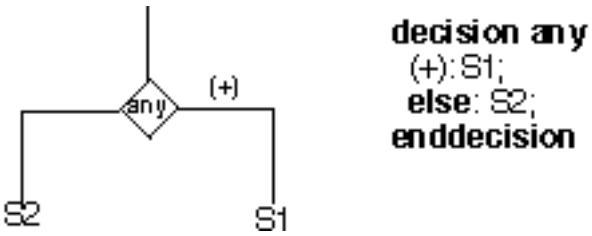


Figure 59: Fair Decision

A fair decision specifies some alternatives denoted by "(+)" which are certain to occur infinitely many times when the decision is encountered infinitely many times. The other alternatives may or may not occur. We have in principle no knowledge of their possibility or probability.

Defined in this way our *fair decision* construct is a construct for *weak fairness* [49 p37]. The (+)-specified alternatives are the *helpful directions* which are continuously enabled when the decision is encountered. Our construct is defined such that the helpful directions will never be postponed infinitely.

For those more imperatively inclined, the following transformation of the construct of Figure 59 (p. 103) gives a definition inspired by the method used by Apt et al. in [1] as referred in [49].

In the definition given in Figure 60 (p. 103), S1 is the helpful direction, which in our notation would be denoted by (+) and S2 is a normal branch, optionally denoted by (0).

```
dcl z1 Integer:=any(Natural);
dcl z2 Natural:=any(Natural);
...
z2:=any(Natural);
decision (z1<=z2)
  (true):  S1; z1:=any(Natural);
  (false): S2: z1:=z1-1;
enddecision;
```

Figure 60: Imperative definition of fairness

The Integer variable z1 designate the priority of alternative S1 while z2 designate the priority of alternative S2. Whenever the helpful direction S1 has been chosen the initial setup is repeated and we are back to square one. Whenever the non-helpful direction S2 has been chosen, the priority of the helpful directions (here: S1) is decreased (improved). Whatever z2 becomes (greater than 0), sooner or later z1 will become less than or equal to z2 and the helpful direction will be chosen.

3.5.3.3 Extremely fair decision

Are we completely happy with our definition of a fair decision? It turns out that most programmers would like to be "even more fair" than our weakly fair definition in Section 3.5.3.2 (p. 103). Most practitioners will have an implementation-oriented attitude towards a fair decision. They will be thinking about how the decision should be implemented, and they would like it to be implemented as *locally* as possible. A local implementation means that the outcome of the decision should not depend on other parts of the description than the construct itself. Neither should the global state be considered nor should the process scheduling matter.

The practitioner will think about implementing a random selection between the alternatives where the helpful directions are assigned constant positive probabilities and the other alternatives possibly a zero probability. Such an approach is called *probabilistic*. From probability theory we get that the probability of being postponed infinitely when having had an infinite number of constant positive chances is zero. Thus we have achieved our fairness for the helpful directions.

It turns out, however, that a probabilistic approach is not equivalent to the weak fairness approach sketched in Section 3.5.3.2 (p. 103). An example will clarify this. Assume that the Receiver receives messages from multiple Senders. Since the messages always carry their origin (SDL predefined value SENDER), the Receiver should only take care to send the acknowledgment back to SENDER. In this many-to-one situation, however, the weak fairness construct for the decisions in Receiver is not sufficient. To see this we may consider the specific sequence of signals which is such that every time the Receiver decision is OK, it is the process A which may terminate its loop. Process B never seems to win an OK from the Receiver. This situation is legal from the point of view of the Receiver decision because it actually returns OK infinitely many times, but process B never gets the benefit of it. From the definition of weak fairness of the decision, this is legal. From a probabilistic point of view, this situation cannot occur. Since process B has infinitely many independent decisions taken in the Receiver, the helpful directions with positive probability must turn up!

The clue is that our "starvation" of B was legal according to weak fairness since the selection of the events was made according to some global state while the weak fairness concentrated on the decision alternatives and not the global state. We want a fairness concept that is independent of which global state there is. Francez defines *extreme fairness* in [49]. *Weakly extreme fairness* means that for any global state in a finite set Γ, helpful directions shall occur infinitely-often in an infinite sequence of decision invocations as long as they are continuously enabled. In our case the global state set Γ would cover different Senders approaching the Receiver.

For practitioners the probabilistic interpretation is most often the best since it models the sense of stepwise execution best. Even with the extremely fair interpretation we can apply the transformation shown in Figure 60 (p. 103) only demanding that the anyvalue expression is really a random drawing from a distribution where all Natural numbers have the chance to be chosen. That the priority of the helpful direction is improved every time the other direction is chosen, is actually of no importance. The clue is that the helpful direction always has a non-decreasing probability to be chosen.

We may also express the notion of extreme fairness in terms of the sequence of alternatives chosen by the decision. For *any* infinite subsequence of the sequence of alternatives, there is an infinite number of helpful directions.

3.5.3.4 Wrapping up the Alternating Bit Protocol

Having defined our notion of a fair decision, we return to the Alternating Bit Protocol which has four such fair decisions where the helpful directions are those which are labelled OK.

The question of weak progress of the Alternating Bit Protocol is a question of the termination of all the feedback loops, which should now have been covered, even though we have not performed a formal inference of the termination of all possible loops.

Assuming now that we have made reasonable both weak progress and confluence of the Alternating Bit Protocol, we may reduce the block described in Figure 56 (p. 101) to an SDL process shown in Figure 61 (p. 105).

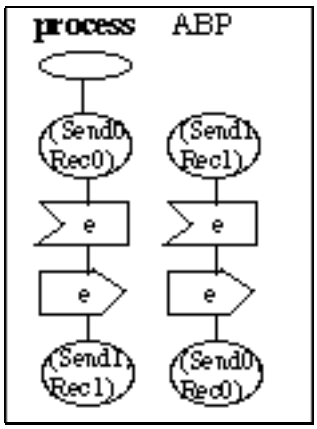


Figure 61: Alternating Bit Protocol Reduced

The process ABP can easily be reduced even more to only one transition consuming e and outputting the very same signal e. Even a practitioner (or should we say: even a formalist?) may be convinced that the protocol is correct.

Strong progress is also the question of avoiding that saved signals are never being handled. The progress problem of the Alternating Bit Protocol is whether we are certain to receive B0 or B1 in Wait0 and Wait1 such that the Sender will proceed. We have already shown in Section 3.4.4 (p. 96) that reduction of the Alternating Bit Protocol implies strong progress since the problematic situations do not turn up as stable basic states of the reduction.

The same result of strong progress can be reached through the following argument. Every entry to a Wait state is directly preceded by output of an A0 or A1 to the Receiver. Every transition of the Receiver will output a B0 or B1 back to the Sender. Thus we are certain that when the Sender is in a Wait state there is something coming from the Receiver sooner or later.

We have been able to divide the problem of the correctness of the protocol to questions of progress and confluence.

3.5.4 Spontaneous save

We have introduced constructs for the explicit specification of non-determinism. In Section 3.5.1 (p. 97) we presented the plain SDL non-determinism in decision and in Section 3.5.2 (p. 99) we covered spontaneous transitions which are also a part of the SDL standard. Finally in Section 3.5.3 (p. 100) we presented a new construct which introduced explicit fairness into decisions.

In our basic execution model (Section 2.1.2 (p. 42)) we have also assumed another form of fairness, namely the fairness between the channels of a process. A signal shall eventually be consumed no matter what channel it is on. In standard SDL this is normally also considered the case only obstructed by priority signals which may always precede the normal signals. In SDL all signals of a process will enter into an *input port* which is basically a FIFO queue which is fair[1].

Confluence means that the choice between internal channels and between external channels and internal channels are irrelevant wrt. the final result of the subsystem. What then if the non-determinism between channels *is significant* for the final result, and this is totally acceptable? Are such systems not reducible?

3.5.4.1 Explicitizing race conditions

We could need a construct to express that a certain race condition is actually “acceptable”. Furthermore the construct must make it possible to talk about progress and confluence and preserve the desired non-determinism. In that way we may specify reductions of systems which also include race conditions.

One problem is that the number of legitimate outcomes is usually infinite. If we want to express the non-determinism of the race condition between an external and an internal channel, we must express that the internal signals may come in between any number of external signals. This is an infinite set of alternatives. We need a shorter notation for this set.

Let us look even closer at the problem. We want to accept the permutations caused by a race condition as acceptable. In order to be able to utilize our concepts of confluence and reduction, we must be able to express that a system (say Sk as shown in Figure 66 (p. 110)) *containing* a component FM with an acceptable race condition is confluent. Furthermore we must be able to apply our reduction algorithm to the system Sk and obtain a reduced process.

The reduction algorithm can choose to execute any internal signal present in the intermediate instable complete state. We realize then that the execution path of the reduction algorithm may never actually encounter a complete state where there are signals present on several channels which should be (fairly) merged. We need a mechanism which takes care of the race condition even when there is no race condition! We need a mechanism which describes a potential merge situation.

We should emphasize that we have no intention to make a construct which changes the SDL semantics. What we want to find is a way to describe a race condition which preserves the fact that the outcome of the race condition is significant for the final result.

1. Some doubt has been raised whether the formal definition of SDL [79] actually defines this fairness, but a common interpretation is that there is a fairness between channels in SDL.

Let us imagine ourselves as a signal on a channel in a race condition. If we suffer from some random delay, or we are disabled for some random period, we fear that signals from some other channel will win the race. But we cannot know. The other channels may also be hampered on their way to the finish, or there may be no other competing signals around at this point in time. On the other hand, there may be other signals on a number of other channels such that there are several signals which suddenly pass us before the finish. The clue is for how long we are being disabled or delayed relative to the other competing signals.

Taking the position of one signal is exactly what the common SDL execution does. The different signals have no knowledge of other signals. When a signal is first in the input port, the state of the process is all that matters. We have to find extensions to SDL which are operations on one individual signal in harmony with SDL in general.

It seems that we need a mechanism that disables the consumption from a channel temporarily, and then some mechanism that enables the signals of the specific channel again.

To simulate disabling of a signal we resort to the only SDL mechanism which delays signals which are first in the input port – the **save** construct (see Section 3.4 (p. 94)). A **save** means that the tentatively consumed signal will not be consumed anyway and rather saved for later until the process is in a state where this signal type can be consumed. **Save** by itself does not introduce non-determinism, but it introduces permutation. Our disabling mechanism will be a kind of **save** which not only saves the first signal of a channel, but all signals of a channel. We call this special save construct a *spontaneous save*.

To simulate enabling, we turn to spontaneous transition covered in Section 3.5.2 (p. 99). A spontaneous transition introduces non-determinism such that the time in which a transition executes is not known. This corresponds well with an unknown delay of the signal. Our modified spontaneous transition concept will be dependent upon there being some signal which has been saved by our spontaneous save construct sketched above. This special spontaneous transition is called *spontaneous consumption.*

Relative to an enclosing system (say Sk) the spontaneously saved signals are internal, but the consumption will appear as a spontaneous transition on top level (i.e. Sk-level) as described in Section 3.5.2.1 (p. 99).

Spontaneous save and spontaneous consumption is not intended for the SDL specifier. The extensions are used in the Mn-procedure and in the reduction. The SDL specifier gets a way to annotate that a race condition is acceptable and there will be an automatic transformation to spontaneous save and spontaneous consumption.

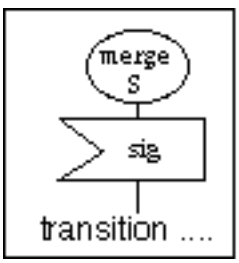


Figure 62: Merge state

In Figure 62 (p. 107) we introduce notation for an acceptable race condition, a **merge** state. Normally if there is one merge state in a process all the other states of the process reachable from the merge state should also be merge states. This is because reachable states must also be able to consume spontaneously saved signals. Notice also by definition that a merge state cannot be the base of a non-confluence pattern since we accept all possible mergers of signals.

In Figure 63 (p. 108) we define the notations for spontaneous save and spontaneous consumption. We notice that a spontaneous save may have a part of a transition before it

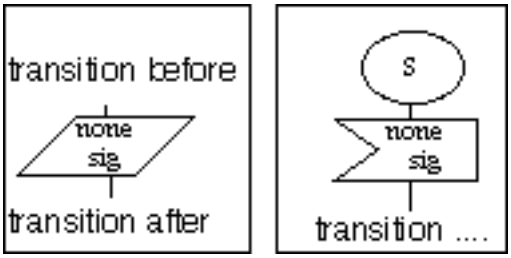


Figure 63: Spontaneous save and spontaneous consumption

and a part of a transition behind it. This is because the signal (here: sig) of the spontaneous save is normally an internal signal and the transition of a reduction may have the spontaneous save as an intermediate action. The transition of the reduction may go on consuming other internal signals of other component processes. Likewise the spontaneous save may succeed a series of internal transitions. Remember also that all spontaneous consumptions similar to spontaneous transitions (Section 3.5.2.1 (p. 99)) are lifted to the top level of the reduction.

The transformation from the merge state of Figure 62 (p. 107) to the new mechanisms is given in Figure 64 (p. 108).

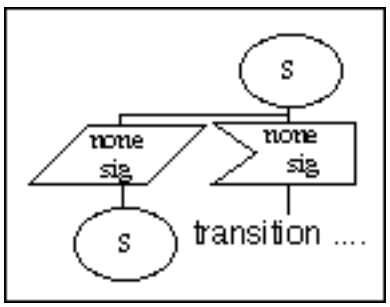


Figure 64: Transformation of merge state

The merge mechanism is extremely fair in the sense that any spontaneously saved signal will eventually be consumed. The spontaneous consumption requires that a spontaneous save has occurred and that such an object is first in the channel.

In Figure 65 (p. 109) we give the definition of the merge mechanism. We show complete states where the component process containing the merge state has only two channels. We show merely those parts of the complete state which are relevant to the execution of the component process containing the merge state. We assume the concatenation operator "+" for concatenation of signal sequences of the complete state, and the transition table $T : S \times A \rightarrow K$.

1. *Spontaneous save*: $(\text{merge } S;i\varphi, \theta;\Upsilon) \xrightarrow{i} (\text{merge } S;i_{saved}\varphi_{saved}, \theta;\Upsilon)$

2. *Spontaneous consumption*:

   $(\text{merge } S;i_{saved}\varphi_{saved}, \theta;\Upsilon) \xrightarrow{\text{none i}} ((\varphi_{saved}, \theta;\Upsilon) + T(S, i))$

Figure 65: The merge mechanism

1. *Spontaneous save*. An internal signal i is first in a channel in the merge state S. Then the signal i is spontaneously saved and so are all the other signals on that channel. This will lead eventually to a semi-stable state (Section 3.4.1 (p. 94)). A semi-stable state may accept signals from other channels. The reason for saving all signals of the channel is that we do not want to have signal overtaking on one channel. This is different from normal **save**.
2. *Spontaneous consumption*. From a semi-stable state spontaneously saved signals may be spontaneously consumed. This means that at any point in time non-deterministically either nothing happens, or one spontaneously saved signal is consumed. The spontaneous saved signal must always be the first on some channel. Spontaneous consumption is extremely fair such that no spontaneously saved signal will be delayed for ever. Said in a probabilistic way, there is a positive probability at each decision event that a spontaneously saved signal will be consumed when it is first on its channel.

What we have defined is no new semantics, but merely a notation which makes it possible to describe acceptable race conditions in reductions. By introducing the spontaneous save, we have made a complete state which before was considered instable, into a semi-stable state. This means that the reduction algorithm halts its execution of internal signals. The consumption of the internal signal which was spontaneously saved is made external as a spontaneous consumption. This boils down to lifting internal transitions up to a global level. If all component processes are full of merge states, then all transitions are lifted to be global and no real reduction has taken place. We would not gain anything from performing the reduction and that is exactly what one would expect.

Since our new mechanism depends solely on signals of one channel (at the time), there is no problem with the reduction algorithm. Thus we have been able to describe merge situations without actually executing complete states with signals on more than one channel.

3.5.4.2 Fair Merge and the Brock-Ackerman anomaly

The Brock-Ackerman anomaly was described in [14] in 1981. The point of the example was to show that history-relations do not have the expressiveness sometimes required for real time asynchronous systems. A history relation is a relation which describes the process as a relation between external inputs and external outputs.

Central to the problem with the Brock-Ackerman anomaly is the concept of *compositionality*. By compositionality we mean that the analysis of a system can be built from the analysis of its structural components. What the Brock-Ackerman example shows is

that when a component is described by history relations, this description is not always sufficient to use in the analysis of some enclosing entity. We may also express this by saying that history relations cannot quite capture the full semantics of a system of communicating finite state machines.

The Brock-Ackerman example contains non-determinism deep inside the system. To reach the necessary expressiveness it is necessary to find some means to express the inner non-determinism on the global or external level. Broy and Stølen [19] present two ways to lift the internal non-determinism out as an external stimulus. One way is to include time ticks in the signal stream. The other way is to define a “prophecy” which describes the possibilities of the inner non-determinism as a parameter to the (global) stream processing function. The stream processing function then becomes a set of functions. In our version below we shall use spontaneous consumptions as the externally defined non-determinism which expresses the inner non-determinism.

We shall go through the example described by our SDL-like notation and show that our reduction strategy is expressive enough to capture what history relations do not capture. Central to the problems of the example is the non-determinism introduced by the race condition in the Fair Merge component which is internal to the system.

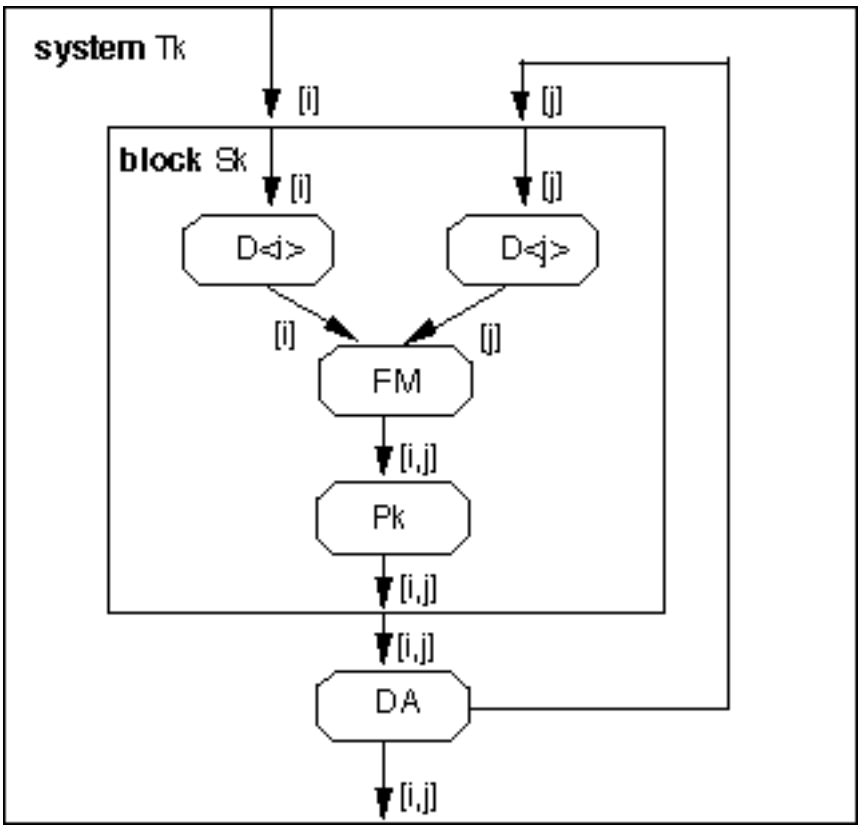


Figure 66: The Brock-Ackerman example system

The Brock-Ackerman example includes two variants of a system, systems T1 and T2. They both have the same structure as given in Figure 66 (p. 110) where k is either 1 or 2. The overall idea for the systems Tk is as follows. The processes D duplicates the incoming signal. FM merges fairly the two channels into one output channel which is in turn input into Pk which forwards the two first signals and then terminates. DA forwards

the incoming signal one by one onto the environment as well as generating a j signal onto the internal feedback channel every time an i is consumed. In Brock-Ackerman's version of the example the DA is represented by the signal being fed back with a positive increase in the value of the parameter. Our version shows the point equally well without introducing data into the CFSMs.

The difference between T1 and T2 lies in P. P1 outputs one signal when it consumes the corresponding one, while P2 buffers the first signal and outputs both when the second arrives.

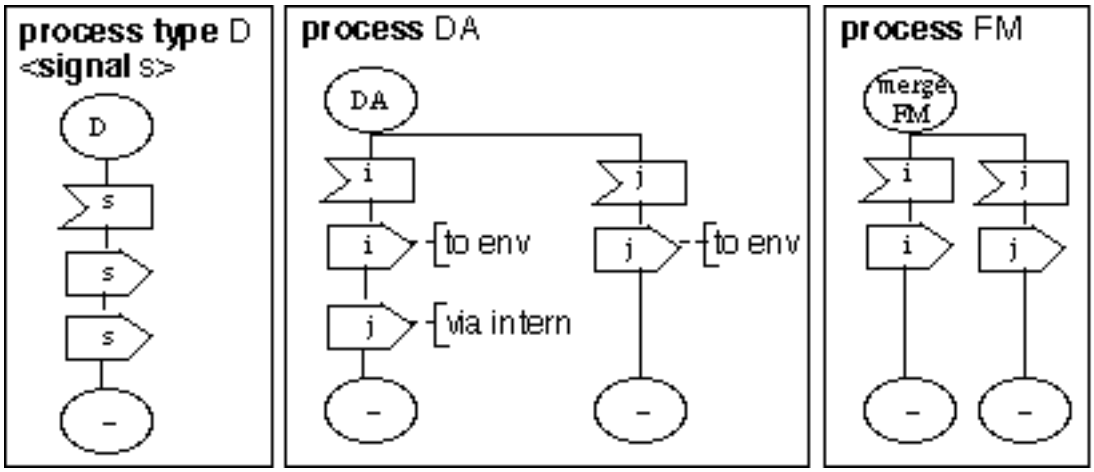


Figure 67: Processes D, DA and FM

As shown in Figure 67 (p. 111), the processes D, DA and FM are quite trivial. The clue is the difference between P1 and P2 shown in Figure 68 (p. 111), and the way the system is connected shown in Figure 66 (p. 110).

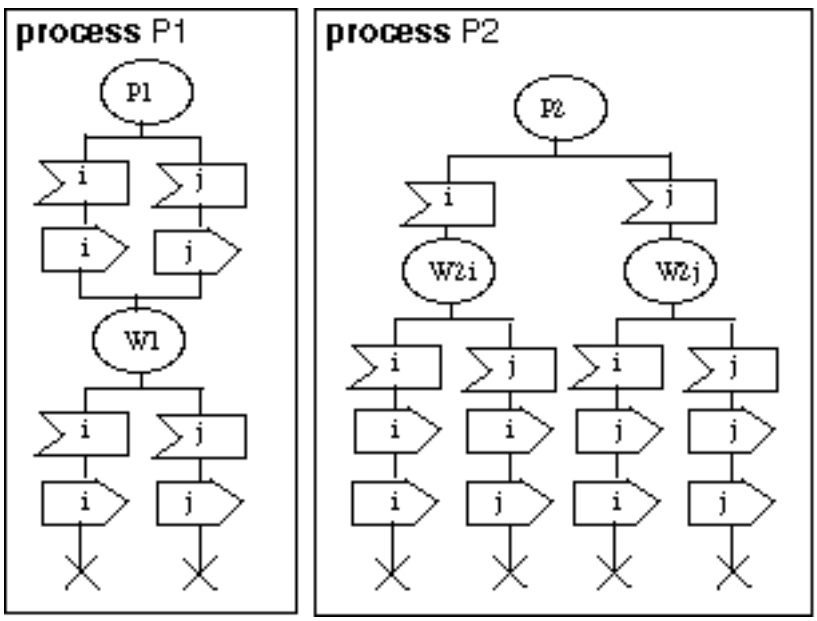


Figure 68: Processes P1 and P2

Brock and Ackerman show in their paper [14] that the history relations of S1 and S2 are identical and equal to the definition in Figure 69 (p. 112), but that S1 and S2 actually behave differently when appearing in the contexts T1 and T2. This shows that history relations are not compositional in the sense that we argue in Section 4.1 (p. 143) that our reductions are.

$$S_k(\text{ø}, \text{ø}) = \{\text{ø}\}$$
$$S_k(iX, \text{ø}) = \{ii\}$$
$$S_k(\text{ø}, jY) = \{jj\}$$
$$S_k(iX, jY) = \{ii, ij, ji, jj\}$$

Figure 69: History relation for Sk

Let us now see how our reduction strategy works on Tk and what results we reach.

Progress of Sk is easily established since the signal ordering criterion holds when we annotate signals with their channel. Progress of Tk is worse since there is a feedback loop of j being fed back from DA to D<j>. Progress is assured through the termination of Pk after having issued two signals. Confluence of Tk is also easily established since it is only FM which has a potential conflict and that has been resolved by the merge-mechanism. By definition of the merge-mechanism, we conclude that FM is confluent relative to all levels enclosing it. Thus Sk and Tk are all reducible.

We start by reducing S1 illustrated in Figure 70 (p. 112). The state names inside the complete states refer to the state names of P1 since all the other processes have only one state each which cannot change. Semicolons separate state, external input, internal queues and external output. The stable states are enclosed by rectangles and labelled.

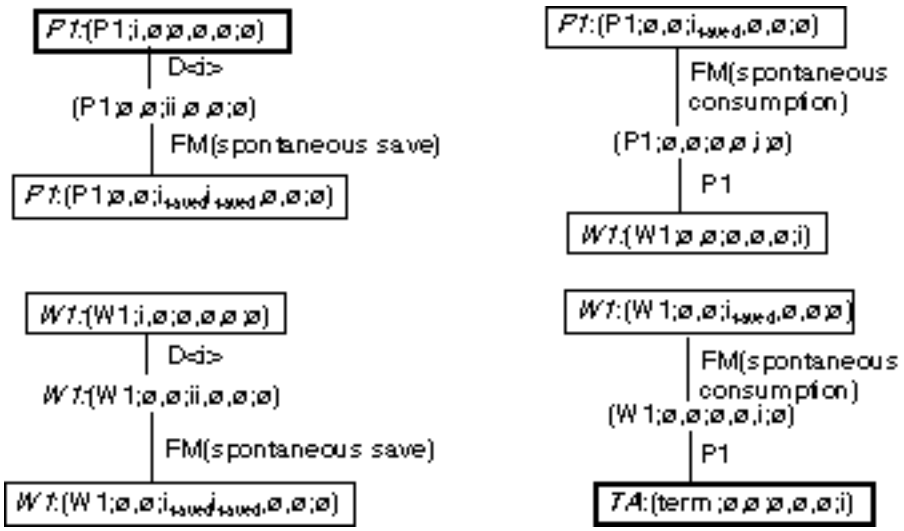


Figure 70: Stabilization during reduction of S1

We notice in Figure 70 (p. 112) that we reach some semi-stable, but not totally stable states. The stabilization of the stimulus j from the start state is symmetrical to the shown stabilization of i. The totally stable state is marked by a fat enclosing rectangle. It is in the state **term**, which in this case means that P1 has terminated. Termination means that

states with internal signals are still totally stable since no more signals will actually be processed for output. Equivalently we may assume that a terminated process always just consumes its input.

We should also notice that the system state P1 comes in two variants, one totally stable and one semi-stable where there are spontaneously saved signals. This is a consequence of the spontaneous save rule. The state W1 seems also to come in these two variants, but that is not quite true because it is a provable invariant that when the system is in W1, there is a positive odd number of spontaneously saved signals. It can also be proved that in P1 there is an even non-negative number of spontaneously saved signals. In our case these extra invariants are not really needed for the reduction as such.

The picture is symmetrical for the consumption of external input j. We have no benefit from separating as different basic states of the reduction those without and with spontaneously saved signals. The defined extreme fairness of the spontaneous consumptions takes care of the merging.

We can now show the reduced S1 process in Figure 71 (p. 113).

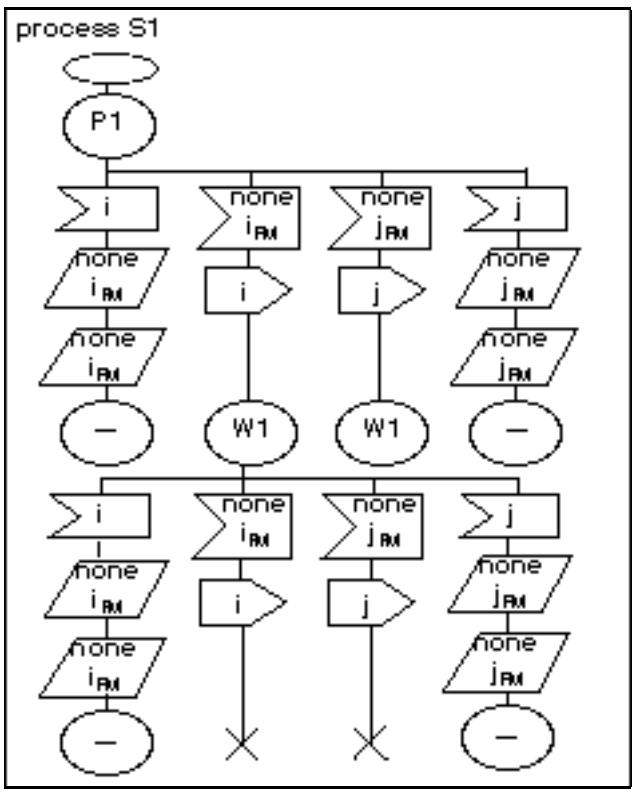


Figure 71: Reduced S1

It is uncertain whether the reduced process description of S1 is more transparent than the block description given in Figure 66 (p. 110) supplemented by the process descriptions of Figure 67 (p. 111) and Figure 68 (p. 111). It may be argued that the process description shows the real complexity of the description while the block description hides the problems. In this section the readability of the description is not the main issue. Here we concentrate on showing that our SDL notation is powerful enough to express the differ-

ence between S1 and S2 when we apply these reductions to T1 and T2 respectively. The difference reappears in T1 and T2 contrary to what happens when history relations serve as the reduced descriptions.

By reducing S2 we reach the following similar, but not equal SDL-like description in Figure 72 (p. 114).

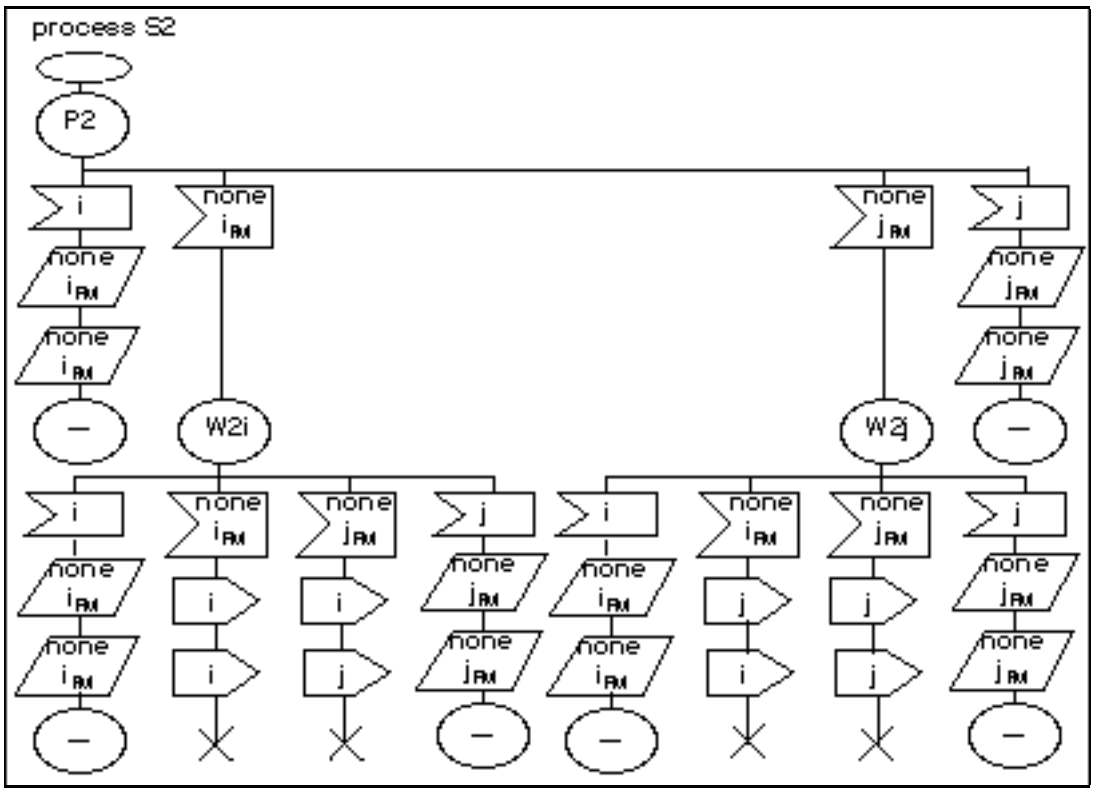


Figure 72: Reduced S2

It is not obvious that the difference in structure between S1 and S2 is actually significant wrt. any enclosing system, but it turns out to be very significant in the context of Tk.

We continue to use the very same strategy on T1 and T2. They are also reducible, and now the reduced processes turn out to be significantly simpler to read. We show also the reduction steps in Figure 73 (p. 115) to demonstrate that the reduced processes are reached stepwise according to the compositionality of our method as argued in Section 4.1 (p. 143) and not directly from the original systems.

We see from Figure 73 (p. 115) that the W1 state may have spontaneous i_{FM}-signals and/or j_{FM}-signals. This is the reason behind the two transitions for spontaneous consumption.

In Figure 74 (p. 115) we observe that T2 can never produce j_{FM}-signals, while i_{FM}-signals are at least produced when more i-signals arrive when in W2i. Actually due to the invariant mentioned earlier, there will always be another i_{FM}-signal when T2 is in W2i.

From the reductions we can make the SDL-like reduced processes shown in Figure 75 (p. 116).

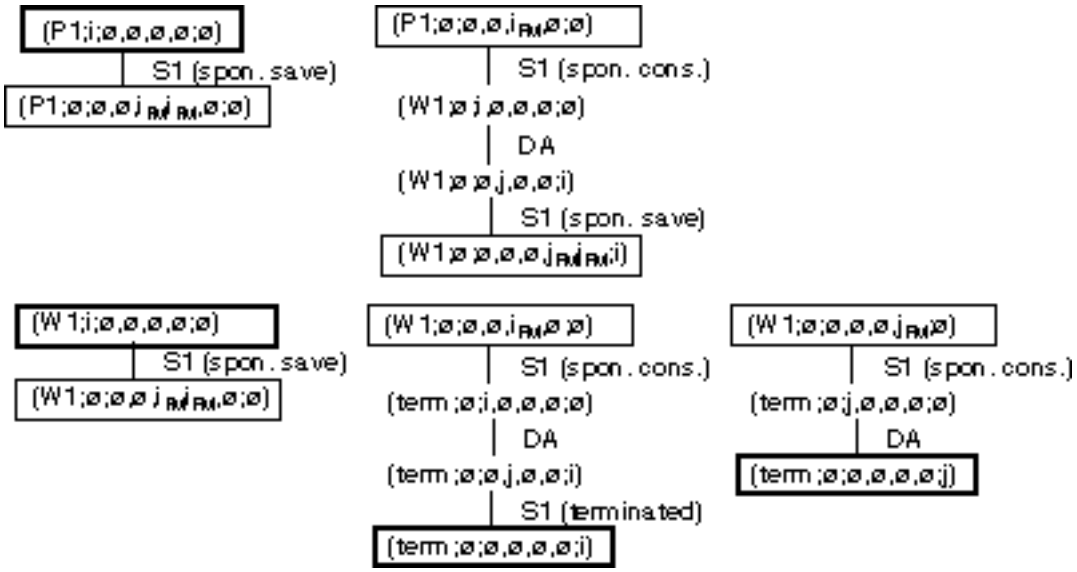


Figure 73: Reducing T1

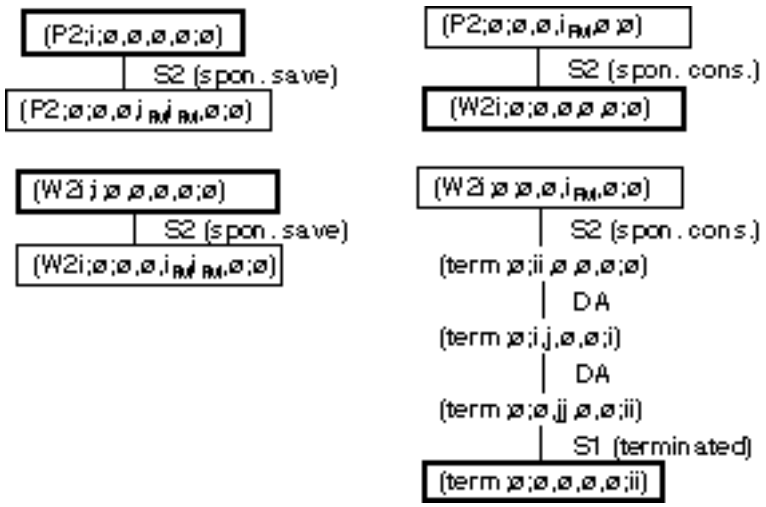


Figure 74: Reducing T2

In Figure 75 (p. 116) we see the difference between T1 and T2. We can see that T1 may produce the sequences ii and ij, while T2 can only produce the sequence ii. This is according to the Brock-Ackerman findings. The cause of the difference is of course that in T1 there is possibility for a feedback, but in T2 it is not possible that the fed back j-signal will ever reach DA since both i-signals have to come first.

We summarize:

1. Non-determinism caused by a race condition that is considered acceptable, can also be modeled in our framework such that confluence and reducibility can be established and reduction performed.

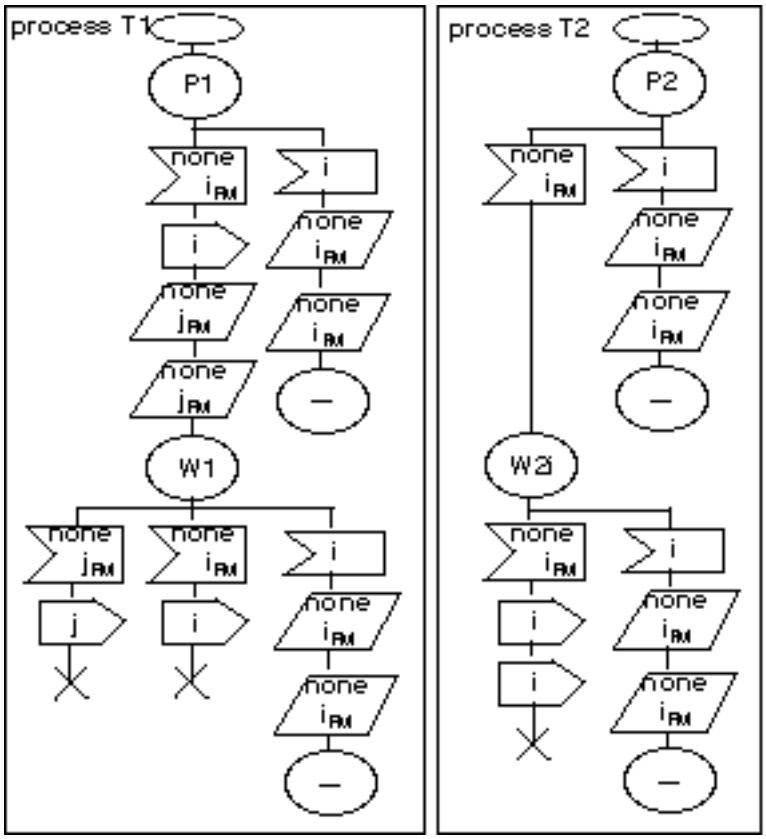


Figure 75: Reduced processes T1 and T2

2. The clue trick to describing acceptable race conditions is the spontaneous save. It makes it possible to specify the desired non-determinism without sacrificing the simplicity of the reduction algorithm. The non-determinism of the race condition is transformed into the non-determinism of the spontaneous consumption. The gain of the transformation is the possibility to use our reduction technique.
3. Our technique is expressive enough to explain the Brock-Ackerman anomaly. The inner non-determinism of the system is preserved in the reductions which then safely are used in the analysis of enclosing systems. The difference between the two variants of the Brock-Ackerman example system is easily seen from the final reductions. The inner non-determinism is also present in the final reductions.

3.5.5 Concluding non-determinism

Non-determinism can be included in our Mn-approach with some penalty in complexity. The Mn-node can no longer be a pair of plain complete states, but must be a pair of tuples of complete states where each state tuple represents the non-determinism of the execution. Likewise, the Mn alphabets must have symbols that are pairs of signal sequence tuples and not merely pairs of signal sequences. The tuples correspond closely with the tuples making up the nodes of the transition systems in the Mn-procedure.

The SDL construct of spontaneous transitions was modeled in two ways, either as an external input with its own channel, or as an internal non-deterministic decision.

We introduced fairness in decisions as a means to provide better ways to ensure progress of the system. The Alternating Bit Protocol example showed this approach and it was successfully analyzed.

Finally we introduced a way to define specific race conditions as desirable. A new construct “spontaneous save” was introduced. Our analysis of the Brock-Ackerman anomaly showed that the construct was useful and that our model was expressive enough to cover the Brock-Ackerman anomaly.

3.6 Data

The handling of data variables in the processes is definitely not the topic of this thesis. The reason for that is mainly because our interest has been the problems concerning race conditions. Furthermore we experience that systems (or subsystems) where concurrency conflicts are in focus very often have rather simple data handling. However, when data is important, they may lead to any complexity, and decidability issues may be either unsolved or negative (undecidable). There are numerous other scholars concentrating on data.

Data as such is only supported by symbolic execution in our Mn-approach. Decisions based on data give rise to alternatives *guarded* by an expression. Instead of pure system states, we introduce guarded system states.

We do not go into detail about how data expressions are simplified.

There are two major ways to treat data in the process of verifying concurrent systems of the kind that we are trying to verify, either the data are ignored or they are included.

3.6.1 Ignoring the data

To ignore the data means that we consider them irrelevant for the major problem of the system under analysis. Either we believe that the data are non-decisive for potential problems, or we mean that we cover whatever values the data have.

We start by transforming the original system under analysis such that the data are removed. The resulting system represents an *abstraction* of the original one (see also Section 4.3.3 (p. 157)). We believe that any problem of the original will also show up in the abstraction, but the abstraction is (supposedly) easier to handle.

The following transformation rules will produce a system without traces of data:

1. Remove all tasks,
2. remove all parameters to procedures, processes and signals,
3. change all decisions to non-deterministic decisions.

This abstraction is true to the original in the sense that any statement which holds for any execution path of the abstraction will also hold for any path in the original. This is because there are no paths in the original which does not have a path in the abstraction since every branch of every decision is represented and there are no changes in the signal sets.

The disadvantages with this abstraction are that progress may be more difficult to establish in the abstraction, and that errors in the original concerning data values are obviously invisible in the abstraction. Confluence may also under special circumstances be more difficult to establish, but normally it will not matter.

Failure to establish progress of the abstraction may be remedied by adding fairness to the non-deterministic decisions with helpful directions to exit the feedback loops. It is necessary, however, then to add some argument that the original system has the quality of fairness in the problematic places.

The biggest disadvantage of this strategy is that an eventual reduced process is not a reduction of the original, but of an abstraction of the original. Thus for the purpose of applying the reduction in other analysis, we have to take this fact into account.

3.6.2 Including the data

To include the data means to do calculations with them. Since our approach is based on calculations from all basic states in the finding of confluence in the Mn-procedure, including the data would mean to perform symbolic executions as opposed to normal executions where values are used. The complete states will be supplemented with the symbolic values of the variables.

To assert confluence it is necessary to verify equality between data expressions. Sometimes this is trivial, while in other cases it is beyond the reasonable scope of a tool to assert the equality. Simplification of data expressions is not a part of this thesis.

The branches of decisions are represented by symbolic boolean expressions. Sometimes they may evaluate generally to **true** or **false** regardless of the values of the variables, but more often they cannot be evaluated to a constant. This means that the continued execution must be performed under the assumption that the branch expression evaluates to **true**. This is what we shall call a *guard*. Every complete state is in principle guarded by an assumption. When the guard is not present, the default is **true**.

The RPC-Memory example in Section 6. (p. 229) shows the use of guards and simple symbolic execution.

When the executions of the reduction algorithm are brought back to the shape of an SDL diagram, the converse strategy is applied. The guards become the branches of decisions and changes of the symbolic values must be made into assignments.

3.6.3 Concluding data

We have decided to exclude a thorough discussion of data in this thesis. It is obvious that data may play significant roles regarding confluence and reducibility. Still we present only two very simple methods to cope with data.

1. Abstract the data.
2. Perform symbolic execution on the data.

The success of the first approach is dependent upon the ultimate analysis goal since the system is transformed to a system which comprises more behavior than the original.

The success of the second approach is very dependent upon the complexity of the data expressions.

We believe that real, reactive systems are such that either the data are quite simple, or it is possible to encapsulate the data complications in data operators which is then used as atomic concepts in the Mn-approach.

3.7 Timers

Timers in SDL are special signals which the process (in principle) sends to itself, and which are delayed a specified duration. Timers which have not *timed out*, can be **reset** and thereby cancelled. The timers are set by a special **set**-construct which can be considered as a special output of a delayed signal. Timers are said to be *active* if they have been set, but not reset or timed out.

Timers are used to conclude operations which have taken too much time or to make sure that certain actions do not start until they should. Timers are basically the only way SDL handles time. We notice that timers are imperative. We cannot in SDL reason about the duration of an operation or a transition.

3.7.1 Basic model of timers

Since timers are signals when they have timed out, they are merged into the same input port as the other “normal” signals. Still in our (modified) basic model we shall assume that timer signals have a channel each and thus may be executed independently of the other normal signals.

On the other hand we shall not expect our systems to be insensitive to the expiration of timers. By this we mean that we would expect that the expiration of a timer would result in a different final stable state than if it had not timed out. Thus we accept that timers introduce non-determinism of the final result. Thus confluence is not dependent on the absence of non-confluence patterns between timers and other internal signals. In this respect our handling of timers resemble our model for acceptable race conditions in Section 3.5.4 (p. 106). The setting of a timer can be compared with a spontaneous save and the expiration and following consumption of the timer is similar to a spontaneous consumption. But there are differences as well, while all signals in a merge state would turn into spontaneous saves, this is not the case with timers. The existence of a timer does not mean that all other normal signals are subject to an acceptable race condition.

We want to establish confluence between channels with normal signals, but we shall have to assume that timers may be necessary parts of such non-confluence patterns. At any state where a timer is active it may in theory trigger since we have no concepts for timed executions in our basic model so far. Therefore there is always a non-deterministic choice whether an active timer times out or not.

3.7.2 Progress

Our analysis method is oriented towards reducibility through the determination of progress and confluence. Timers often help provide progress. Since time is fair in the sense that all future points in time will eventually arrive. Therefore all active timers will eventually time out if they are not reset. This means that the expiration of a timer can be compared with a non-deterministic decision where one alternative is a helpful direction with positive probability. Loops which are guarded by an active timer will eventually terminate.

3.7.3 Confluence

We accept the non-determinism of timers. Thus non-confluence patterns may include active timers in situations with a race condition between normal channels.The existence of an active timer could be necessary to show non-confluence.

In Figure 76 (p. 120) we see a state overview diagram of a process where the existence of timers in non-confluence patterns is shown to be necessary. The legend is that the circles are basic states, the edges represent transitions where the text is “input/output”. Default transitions are not shown. In Figure 76 (p. 120), x and y are input on different channels, t is a timer and when it is on the output side of the slash it means that it has been set in the transition. u,v,w are external output on the same channel.

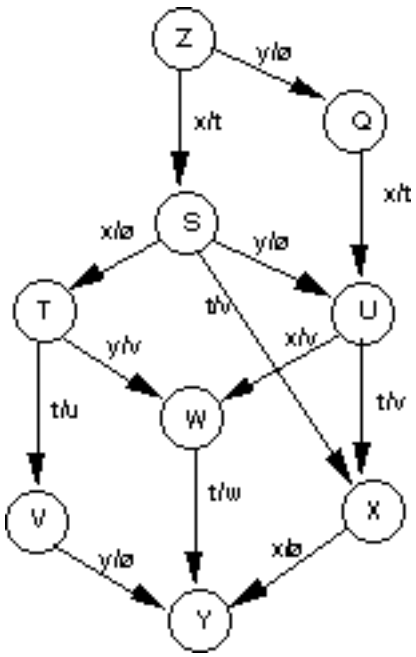


Figure 76: Non-confluence with timers

We can see that in basic state Z, the potential non-confluence pattern (Z;x,y) does not lead to a non-confluence since we have the Mn-procedure branches shown in Figure 77 (p. 121).

We notice also that we have considered the timer which is set in the transitions and also included the continuations when the timer expires.

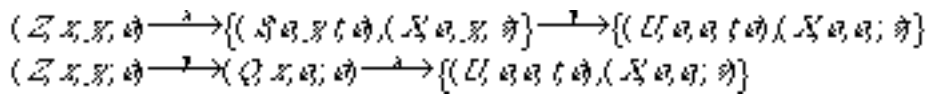


Figure 77: Timers and confluence (1)

Likewise in basic state S no non-confluence can be found for pattern (S;x,y) as can easily be seen from the commutative geometry of the transitions from S. The pattern (S;x,y;t) where t is an active timer, however, does lead to non-confluence as we can see from the execution tree in Figure 78 (p. 121).

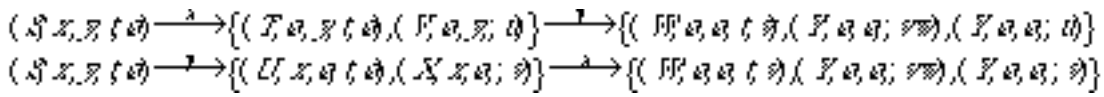


Figure 78: Timers and confluence (2)

We summarize confluence and timers by the following points:

1. Timers are considered to have an external channel of their own;
2. Active timers are considered to have the option to time out;
3. It is not sufficient to consider potential non-confluence patterns without the timers (as shown by the example in Figure 76 (p. 120)). It is not sufficient to consider the timers only in the transitions where they are set.
4. Potential non-confluence patterns in processes containing timers must also consider all situations where the timers may be active.

3.7.4 Reduction

When having considered progress and confluence it is reasonable to consider the reduction once the process has been shown to be reducible.

We shall see that we have two possible approaches to how the final reduction should appear:

1. The timers are completely eliminated.
2. The timers of the components appear as timers in the reduction.

The first approach sees the timers only as means to ensure progress and possibly some non-determinism. The second approach wants to retain the element of time also in the reduced process. So in fact our choice of strategy should be dependent on the purpose of the timers of the components.

3.7.4.1 The Alternating Bit Protocol Revisited with Timers

We shall give one example of the use of timers which also reveals a few interesting features of timers and reductions when applying timers. We take the Alternating Bit Protocol presented in Section 3.5.3.1 (p. 100) as our starting point. In our first version the assumption was that all signals will arrive to the opposite process, but the signal may have been corrupted. Any corruption will be detected. We shall now relax our assump-

tions by allowing a signal to be lost on the way. We model this by making the signals go through a process which either forwards the consumed signal or just consumes the signal. The architecture is given in Figure 79 (p. 122).

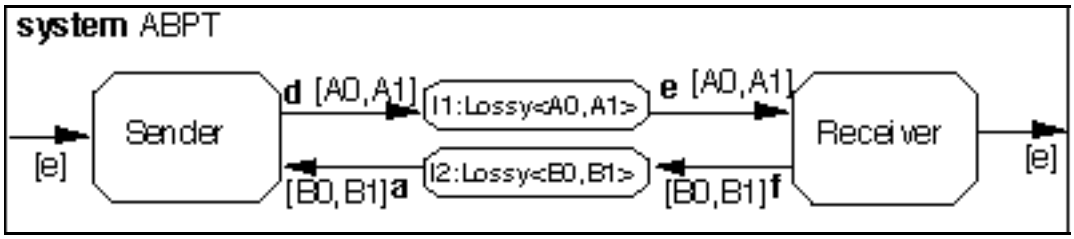


Figure 79: Alternating Bit Protocol with Timers

The definition of the lossy channel is given in Figure 80 (p. 122). We notice that the

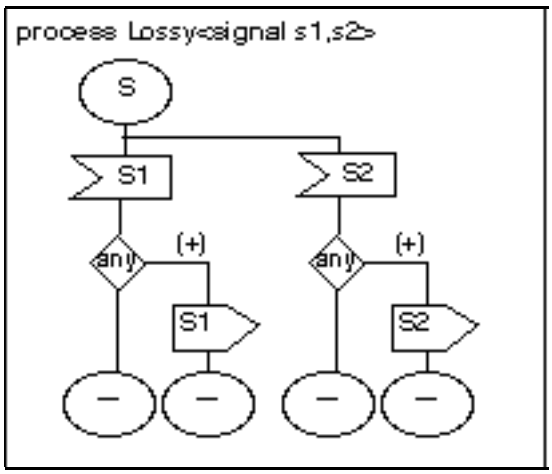


Figure 80: Modeling the lossy channel

lossy channel cannot be constantly lossy meaning that no messages come through. The alternatives to actually transfer the message have positive probability.

A timer is introduced in the Sender which now is modified and shown in Figure 81 (p. 123).

The Receiver remains as defined in Figure 58 (p. 102).

3.7.4.2 Progress and Confluence of ABPT

In Section 3.5.3 (p. 100) we argued for the progress and confluence of the original *Alternating Bit Protocol* where the signals could not get lost. The introduction of lossy communication and a timer does not alter the general argument.

To assert progress we consider the changes made by the introduction of the lossy communication. If no signals are lost, the arguments of the original version applies still.

If the A-signal is lost on its way from the Sender to the Receiver, there will be no action by the Receiver, but eventually the timer expires and another equal A-signal is re-issued. This is very similar to what happens if either the A-signal or the B-acknowledgment is

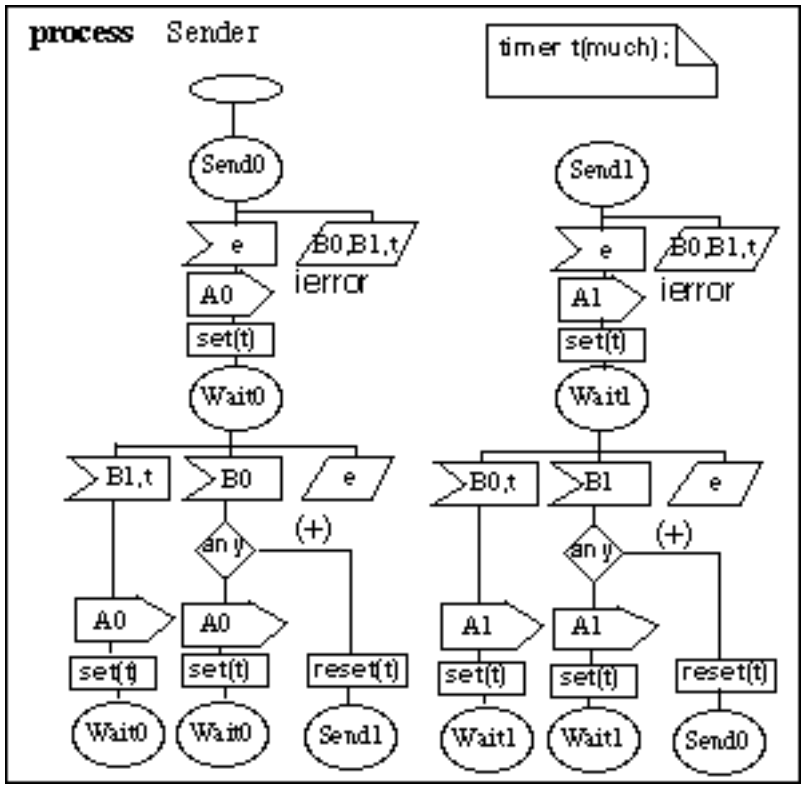


Figure 81: Modified Sender of Alternating Bit Protocol with Timer

found to be corrupted. We are back to the starting situation where the A-signal is sent to the lossy channel. Since the lossy channel cannot be lossy forever this tight loop will be resolved by the lossy channel finally forwarding the A-signal.

If the B-acknowledgment is lost on its way from the Receiver to the Sender, this is a similar situation as above from the point of the Sender. The Sender cannot know what has happened beyond his sending the A-signal. Whether it is lost on its way to the Receiver or from the Receiver is not different at the Sender. At the Receiver, the situation may be slightly different as the Receiver may know that the A-signal has been properly received, and thus output the external signal. Still a lost return from the Receiver is similar to a corrupted return, the Sender should retransmit the original A-signal, which is exactly what it does when the timer expires. When the A-signal eventually occur at the Receiver again (due to earlier argument), the Receiver will again issue the acknowledgment, but refrain to output another external signal. Eventually the lossy communication from Receiver to Sender will forward the necessary number of B-acknowledgments such that one is correct and also the Sender understands that the message has been successfully transmitted.

Considering confluence, the argument is still simpler. Since we are only interested in the race conditions between the external input signal e and internal signals to Sender B0,B1, the use of save in the Sender makes sure that such race conditions are made illegal or impossible. Thus ABPT is confluent.

Since ABPT is both progressive and confluent it is also reducible.

3.7.4.3 Reducible does not mean error free!

Here it is in its place to recall that reducibility is not the same as lack of errors. Even if the ABPT is reducible it may not do the job it was designed to do. What we know is that the reduction will show the same behavior as the full system, but that behavior may not be desirable.

In ABPT we know that the behaviors of the original version ABP is included in the newer version since ABP is equal to an implementation of the channels which always forward the signals. Still there may be more possible behaviors of ABPT than in ABP. Such additional behavior patterns may be due to the timer expirations.

We said in Section 3.7.1 (p. 119) that a timer could trigger whenever it is active. If we start reducing ABPT and let the timer expire very quickly, we see that the Sender will retransmit A-signals not as a result of a lost signal, but just because the timer has expired too early. This may quickly lead to an internal error. Assume that the lossy channels are reliable for a while. Assume that the quick timer expiration results in two A0-signals having been sent to the Receiver. The first A0 results in an acknowledgment of correct reception B0, the second results in an acknowledgment in Rec1 designating corrupted signal which in that state is also B0, The Sender then consumes the first B0 and concludes that the A-signal was correctly transferred and moves to Send1 state. Before another external input has arrived the Sender has to handle the second B0. This is an internal error.

The small fix of ABPT which consumes any B-signals in the Send states will theoretically work provided the external input is finite. The solution is not infinitely progressive very much the same way as process G in Figure 41 (p. 70). Very similarly to that example the Mn-procedure does not halt without manual intervention with induction proofs. The clue here is that a message can be conveyed from Sender to Receiver by any number of (equal) A-signals and the acknowledgment can also be with any number of corresponding B-signals. The eventual consumption in Send states will make the situation stabilize. A practical problem is of course that the very quick timer expiration produces more new signals than what is being processed and then this means that the number of internal signals in the system will increase regardless of new external input or not.

Conclusively it is important that the timer does not expire before it is certain that the signal is actually lost and not just on its way.

3.7.4.4 Sufficiently big duration of timer t

We must make the timer sufficiently big such that it will not expire before it is certain that a signal either is lost between Sender and Receiver or the other direction. On the other hand we do not want the timer to be set to a duration much above what is needed because that would slow down transmission on a lossy line.

Firstly let us assume that the timer is set to a sufficiently large duration. Firstly we shall assume that “sufficiently large” means that the ABPT can always finish other internal signals before the timer expires. Under this assumption one half of the reduction algorithm execution is given in Figure 82 (p. 125). The other half is symmetrical taking the other totally stable state S1R1 as starting point.

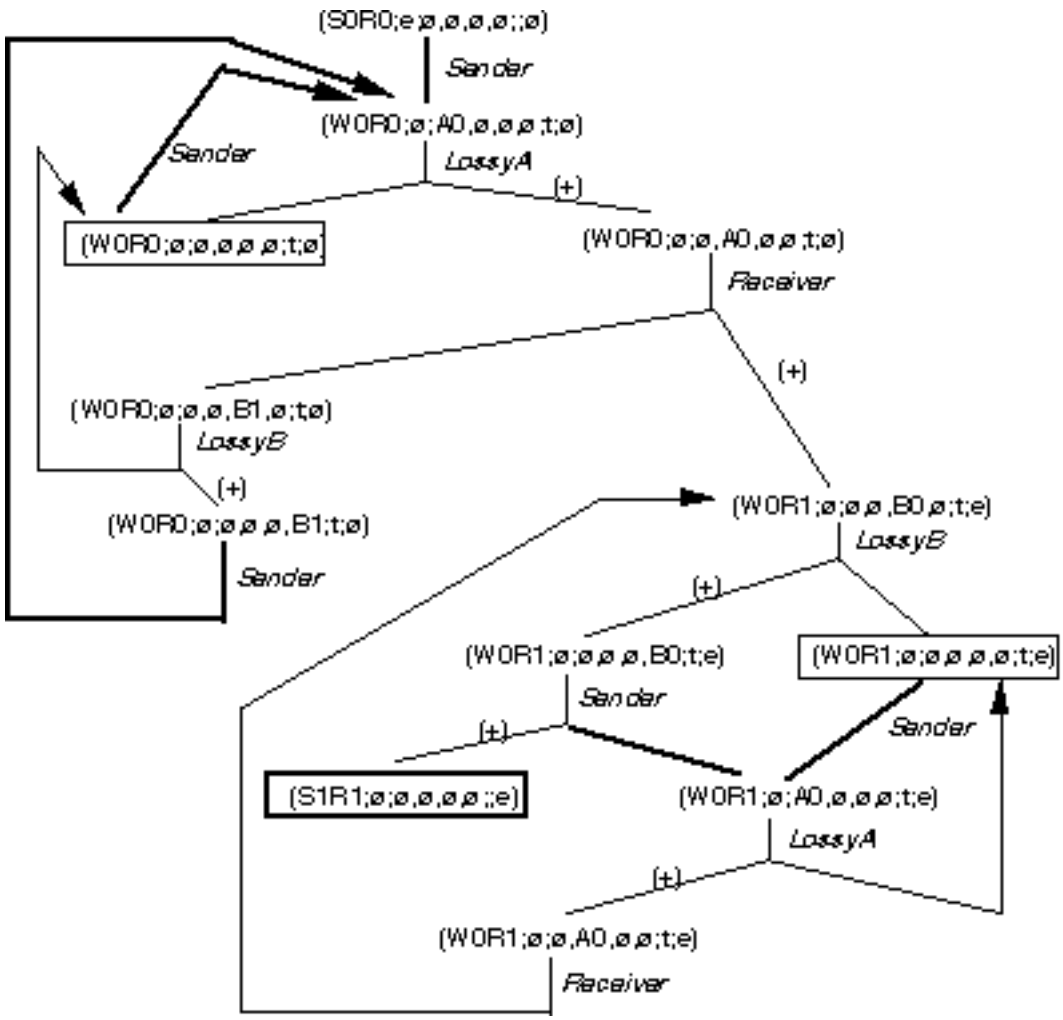


Figure 82: Reducing ABPT

The legend is that a complete state has the form (state Sender state Receiver; external input; internal channels:Sender->LossyA,LossyA->Receiver, Receiver->LossyB, LossyB->Sender; eventual active timers; external output). The states of Sender is abbreviated “S” for “Send” and “W” for “Wait”. States of the Receiver is abbreviated “R” for “Rec”.

Since progress is due to fairness, the reduction tree is a directed graph with cycles. Transitions leading to states which has already been reached is shown by upwards arrows. The cycles represent loops which will eventually terminate by the process escaping through branches marked with (+). The branches denoted by fat lines are transitions where the timer is set (again). Thin line rectangled states are semi-stable when considering an active, but not expired timer as a saved signal. The totally stable state is shown by the fat rectangle.

By pruning all the loops, there is only one possible stable state which is the S1R1 state. This is exactly equal to the original reduction in Figure 61 (p. 105).

Our next question is how we may find such a sufficiently large timer duration.

3.7.4.5 Reductions and timed executions

We want to find some measure of how long the worst case execution of ABPT is without lost signals. It may not be certain that there is any upper bound. Let us rush to emphasize that our basic model does not actually contain time. The arguments in this section is therefore ad hoc in relation to the mainstream of this thesis, and yet another example of how one verification approach alone may not be sufficient.

Firstly we assume that saving an external signal takes no time at all. Otherwise we may always postulate a burst of any number of external input signals such that any given upper bound is exceeded.

Secondly we assume that all transitions have a common upper bound of the duration of the transition itself. When confluence is determined by M0-procedure only, we can conclude that all different execution traces depending on the choice of input channel have the same upper bound. If also timers are included we must make sure that both sides of every potential non-confluence pattern contain the same number of timer time-out transitions.

In ABPT the save construct ensures that only one course of action is legal at any point in time. Race conditions never apply.

Still the loops of our reduction graph of Figure 82 (p. 125) may in principle be such that any duration timer could be too small. The loops must not be such that internal signals are executed unboundedly (not infinitely, because progress has been determined) without the timer being set again. Inspection of the reduction graph reveals that every loop has at least one transition which sets the timer. Thus we may conclude that the execution without loss of signals is bounded. Closer inspection shows that 5 transitions are the maximum including the transition setting the timer.

In the general case to find the worst case trace is by no means trivial and our reduction strategy does not offer much help since the different choices cannot be described in advance. We have summarized below the cases where our strategy can be of some help. We assume that the system is reducible and the problem is to find the smallest suitable duration for a timer.

1. Restrictions on the possible race conditions (by **save**) make sure that only one choice is valid at any point in the execution.
2. There is a reasonable common upper bound for all transitions and for every potential non-confluence pattern the two branches are equally long.
3. It is possible to determine from the Mn-procedures for every potential non-confluence pattern which choice will make the longest execution.

Otherwise manual invariants must be found.

3.7.4.6 The reduction which retains the time behavior

We showed in Figure 82 (p. 125) the reduction graph. Our reduced process can have two forms as pointed out in Section 3.7.4 (p. 121), either eliminate the timers completely or elevate the timers to the system level. In this situation where the timers are used only to break a possible deadlock situation, time as such is of no significance. Therefore the natural strategy is to eliminate the timers and reach the reduction shown in Figure 61 (p. 105).

Alternatively we could keep the semi-stable states as basic states in the reduction and reach the reduction shown in Figure 83 (p. 127). The diagram shows half of the total graph as the other half is symmetrical by exchanging `0` by `1` in all identifiers.

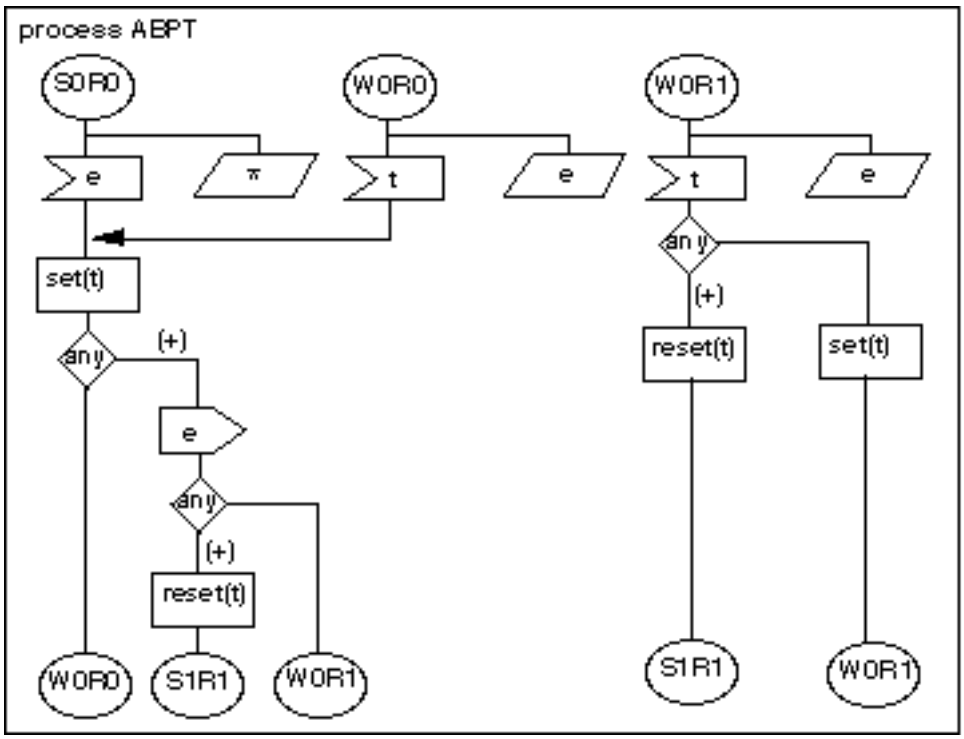


Figure 83: Retaining the timer information

This example is hardly a very good example to show the use of this kind of reduction since the only thing that can happen in states `W0R0` and `W0R1` is that the timer triggers. The normal signals (here: `e`) are saved.

3.7.5 Concluding timers

Timers are a common means to ensure progress in real systems. Message loss and hardware break down can be made less critical if the continuation of the system execution is supervised by timers. We consider timers a special variant of non-determinism and in general we assume that an active timer may always time out. The problem arises when the timer expires too fast. We show that it may be necessary to assume a “large enough timer value”. To find such a timer value may be difficult and the Mn-approach offers little help other than in fairly special situations.

Having established progress and confluence (with sufficiently large timer value), there are two variants of reductions. One variant eliminates the timer and leaves a process graph without any trace of timers. This is the normal choice if the timers have been used exclusively for progress purposes.

The other variant of reduction keeps the timers and thus the reduction may also keep loops where the timers are involved. The benefit of the latter variant is that it may be simpler to use when there is a need to simulate a system and time is of the essence. In the first variant, the duration of a transition of the reduction is typically unbounded in time.

3.8 Procedures

The SDL procedure mechanism is used to structure the behavior of SDL processes. The Mn approach can easily be applied also for such systems by transforming it to independent processes. Once the procedure is working, the mother process is inactive. We transform an SDL process with SDL procedures into a system of two (or more) processes which together is reducible. The point of this transformation is to get the non-basic SDL into the framework that we have already developed.

When the SDL process occurs in larger contexts, the only part which has to be analyzed for confluence on the first generation is the mother part of the SDL process. The procedure processes only communicate one by one with the mother part. This scheme is recursive such that each procedure process themselves may be a system of a mother part and a procedure part. In subsequent generations procedure processes come into play.

3.8.1 A Fictitious Procedure Example

The transformation can be shown by an example in Figure 84 (p. 128).

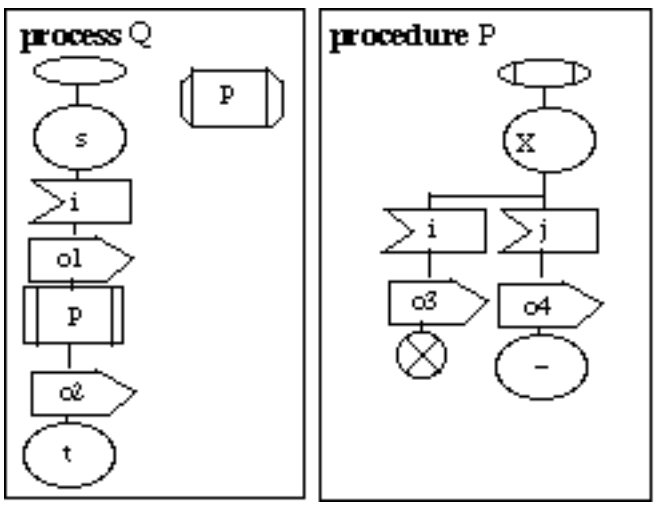


Figure 84: Introducing SDL procedures

By the transformation rules given in Section 3.8.2 (p. 129), the process with procedure is transformed into the reducible system of two interacting processes shown in Figure 85 (p. 129)

The two transformed processes are shown in Figure 86 (p. 129).

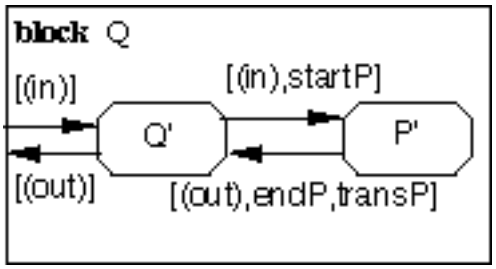


Figure 85: The structure of the transformed process

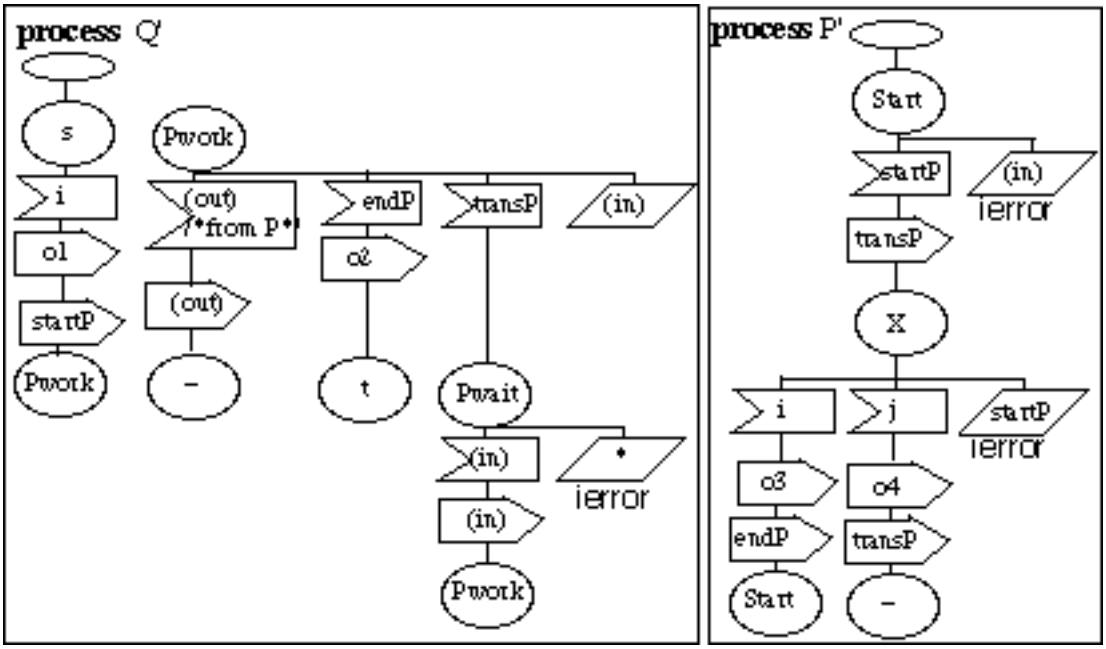


Figure 86: The transformed processes

3.8.2 The transformation scheme

The transformation rules can be summarized as follows:

1. Create the structure of the transformed block from the process (here: Q) by:
   - 1.1 Create a process for the mother part (here: Q) and one process for each procedure (here: P).
   - 1.2 For each incoming channel, create a corresponding internal channel between the mother part and the procedure part.
   - 1.3 For each outgoing channel, create a corresponding internal channel between the mother part and procedure part.
2. Create the mother part from the process main body by:
   - 2.1 For each invocation of the procedure transform it to an output of a starting signal to the procedure P (here startP) and move to a state with a unique name (here: Pwork). Such states are called “work-states”.

2.2 For each such created work-state, handle four different kinds of signals:

- *Normal external input signals*. They are saved.
- *External output signals* which come from the procedure. They are relayed on to the surroundings of the mother process.
- *endP*. This is a signal which designate the **Return** of the procedure. This transition should contain the rest of the original transition after the procedure invocation.
- *transP*. This designates the termination of a transition of the procedure, but no **Return**. This means that the procedure waits in a state. Nextstate should be another unique state (here: Pwait). Such states are called “wait-states”.

2.3 For each such created wait-state handle two different cases:

- *Normal external input signals*. They are relayed to the procedure.
- *Everything else* which designate an internal error and is represented as a **save**.

3. Create the procedure process from the procedure by:

3.1 The start transition of the procedure process is empty leading to a new state (here: Start).

3.2 From this start state handle two different cases:

- *startP*. This is the real starting of the procedure and the transition should be equal to the start transition of the original procedure which should end by either the output of endP or transP (see explanation below).
- *Everything else* designates an internal error.

3.3 Every transition of the original procedure should be repeated in the procedure process, but they should end by either outputting endP or transP.

- *endP*: should be output when the transition terminates with a **Return**. The nextstate should be the start state.
- *transP*: should be output when the transition terminates in a nextstate.

There are some minor problems concerning this transformation scheme. Firstly there is the problem of modeling recursive procedures. The transformation scheme is a static transformation and it cannot handle the unlimited number of procedure invocations that recursion needs. If we used process creation instead of sending the startP signal, and letting the process terminate at **Return**, recursion could also be handled. The *dynamic link* of the call stack would be represented by the predefined SDL function parent. In this thesis we have not covered how systems with dynamic process creation should be handled by the Mn-approach.

Secondly there is the problem of naming and of name scopes. This is a trivial problem which can be handled by clever naming schemes which we find no reason to cover here. If we also want to model a procedure accessing or modifying variables in its enclosing scopes, we need to introduce a pointer (SDL: PId) called a *static link* between the transformed processes which would represent the linkage between different levels of enclosure.

3.8.3 How to analyze procedures

It is quite evident that the transformed block is reducible to a process which dynamically corresponds to the original process with procedure. The transformation scheme creates a one-to-one mapping between transitions of the original and transitions of the transformed. To perform the reduction is of little use as it will amount to expanding the procedure in the original. If the procedure has been used more than once, this is not effective. The gain in work load must lie in considering the mother part of the process separate from the procedure part. The procedure part should be analyzed separately. The gain becomes even clearer when one considers the situation where the procedure is local to an enclosing scope unit and thus used inside different parts.

How can we analyze the procedure (as process) separately? When we analyze for reducibility a block of processes, determination of confluence can be done piecewise taking each process by itself. This corresponds partly to separate analysis.

3.8.3.1 What is special with procedures

Procedures have no gates. Thus we have no idea from the procedure definition itself what input channels and output channels it has. The procedure takes on the channels (signalroutes) of its enclosing process, but in SDL-92 procedures may be local to a block and therefore it may be used in principle in all processes of the block. Thus we must cover all situations represented by the intersection of signal lists on channels and the input signal set of the procedure to find the potentially problematic signal pairs. We analyze the procedure for confluence relative to these potential situations. They represent the use of the procedure in each of the components processes.

Procedures are tightly coupled to its enclosing process (or procedure) which means that there is no need to consider progress or confluence problems involving the special control signals between mother part and procedure part (Here: startP, transP, endP).

3.8.3.2 Progress

Often progress cannot in general be determined piecewise. Non-progress (i.e. livelock) may in certain (very odd) cases be determined. We may determine whether the signal ordering criterion holds within the procedure. This may indicate whether an enclosing scope unit is in turn progressive.

Some procedures may not produce internal signals at all, and then the procedures cannot cause live-lock and as such they can be said to be locally progressive.

3.8.3.3 Confluence

Can the procedure be analyzed separately wrt. confluence? The answer is not a simple “yes” or “no”. The transformation scheme makes it possible to see the mother part as transparent wrt. the incoming and outgoing signals of the procedure. Furthermore the control signals (Here: startP, transP, endP) can surely not be involved in any non-confluence pattern.

We have a few different alternatives corresponding to the evaluation categories of the Mn-procedure. For each possible channel configuration the following verdicts can be reached.

1. *Confluence*. During M0-execution confluence of all branches is established.

2. *Non-confluence*. A potential non-confluence pattern is found. We must be aware that stabilization are often dependent upon other components and then it cannot be performed separately.
3. *Sequence permutation*. M0-execution shows sequence permuted internal output. The non-decisive output of the separate analysis is the M1-internal signal sequences, and the M1 input alphabet. M1 of the procedure is not executed by the procedure itself, but rather the components which communicate with the caller of the procedure.

3.8.3.4 Conclusions of separate analysis of procedures

Conclusively we can say that a procedure does not lend itself easily to separate analysis for reducibility of its users or enclosing scopes. This is not very surprising and the causes can be divided in two groups:

1. A procedure has *no channel interface* (or signalroutes, or gates). This means that we must look at the usage of the procedure to find which channel configuration we need to compare against. The reason for this is mainly because procedures defined in blocks are basically shorthands for many definitions of similar procedures inside the processes of the block. When the procedure is defined within a process, the channel interface of the procedure is the same as the one for the process.
2. Communication is central to the concept of reducibility. To conclude confluence by separate analysis the procedure needs to be confluent regardless of the communication. We would expect this to happen rather seldom.

We feel that expanding the procedures during the analysis will probably often pay off. When a separate analysis concludes confluence, which is what we want, this would probably be quite simple to conclude also from the expanded process where the procedure calls have been expanded. Expanded procedures also have the advantage that all reachability situations have been resolved, while the separate treatment of the procedure also handles situations which can never occur.

Separate treatment does have an edge when the procedure is recursive because then the expansion may not be statically decidable. Furthermore separate treatment is more robust wrt. changes in the use of the procedure. Typically errors occur when procedures are used in ways which they were not really designed for.

3.8.4 Concluding procedures

We apply a simple transformation scheme to convert a procedure call/return into asynchronous communication between the mother part and a process corresponding to the procedure. This transformation makes it possible to treat procedures in the same framework of processes as we have devised already.

The separate analysis of procedures shows strong limitations, but it is very dependent upon the procedure, what a separate analysis can find. One important problem is that the procedure, unlike a process, does not have clear channel interface. This makes it more difficult to establish which race conditions are potentially present.

3.9 Object orientation: Inheritance and virtuality

How does the object-oriented concepts of SDL-92 affect the Mn approach? By “object-oriented concepts”, we consider pure types, inheritance and virtuality. In Section 3.8 (p. 128) we handled procedures which are an example of a pure type. We shall go in more detail into what we can expect to reach when analyzing a type separately. This we shall use as base for our handling of simple inheritance. Virtuality adds another dimension to the analysis as an internal part is partly unknown.

3.9.1 Pure types

The main idea about pure types is that instances of the type can appear several times within the defining scope of the type. Thereby the same type definition is applied several times. This is what is normally called “reuse” of a pattern. In SDL the instances of the type will be identical up to the interface while a superfluously similar construct macro may lead to very different situations when applied several times since the semantics is dependent upon its expansion environment.

Thus a reduction of a **block type** B to a **process type** BP means that during the analysis of blocks and block types surrounding instances of the **block type** B, we may instead of B use BP to find out the behavior. In many cases this simple compositionality explained in Section 4.1 (p. 143) is very practical to reduce the complexity of the analysis. In cases where we do not want to use our reduction strategy on this enclosing level, we may still use the reduced BP in stead of B as it externally behaves identically. This use of the reduced version in place of the original one, we may call the *inwards* use of reduction for the analysis of an encloser.

The *outwards* use of reduction is when we want to use our reduction strategy also for the encloser. We recall that progress typically is not done piecewise while confluence typically is determined piecewise.

Let us here concentrate on confluence. Assume that we are analyzing the encloser for confluence. In the first place this means analyzing whether each component (and therefore also any B) can produce a non-confluence pattern. Whether the B can produce a non-confluence pattern is dependent upon whether the input and output channels of B (described as **gates** since B is a type) are internal or external to the encloser under analysis. Seen from B only, paying no attention to which enclosing unit we analyze, we may qualify its confluence by the kinds of the channels (gates).

We assume first that B is confluent as such, meaning that it is confluent given that all input channels are external and all output channels are external. This we may call *basic confluence*, but we have both weaker and stronger forms of confluence. The *strongest form of confluence* is when a B never can show a non-confluence pattern in any encloser. This happens if B is confluent when all input channels are internal. We notice that such a configuration is typically a component of a system since the input channels are not necessarily connected to the outputs. Therefore stabilization will in general need other processes as well. Conclusions can only be drawn from this when stabilization can be decided from this component alone. Otherwise eventual strong confluence is dependent upon a successful stabilization. The *weakest form of confluence* is when all the output channels have to be internal because B produces sequence permutations onto the chan-

nels and it must be up to the process receiving signals from B to compensate the non-confluence of B. Between the strongest and the weakest forms of confluence there are all variants depending on which kind the problematic channels are. We call this outwards use of reduction because during the analysis of B we can specify the role of B in the confluence of enclosers.

We have here concentrated on the M0 confluence of B. It is possible also to keep as annotations to B, higher generation information. Assume that during the analysis of some enclosing entity C of B, we found that component A to be weakly confluent dependent upon the compensation by B on M1. This means that we find an M1-alphabet and an M1-origin[1] from A which we try on B. If B is able to compensate for the non-confluence of A, this ability to compensate could be stored as a property of B. It is not practical to try and solve the M1 situations in advance since there is an infinite number of possible M1 situations. Such practical considerations will be covered in greater detail in Section 5. (p. 177).

3.9.2 Simple inheritance

Inheritance in object orientation means that a new type is derived from another already existing type. The relation to the existing type is established only by referring to the type in an inheritance clause. In SDL this referencing is done in the header as can be seen in Figure 88 (p. 136). The semantics is that the new type starts as a copy of the inherited type and adds new features in its own description. Hierarchies of inheritance represent conceptual structures where the most general concept is at the root and the more specialized concepts closer to the leaves.

Inheritance without virtuality is not much more than plain aggregation when it comes to analyzing it for reducibility.

3.9.2.1 Inheritance of block types

Inheritance of block types means that a mother part and a specialization part are aggregated as two blocks besides each other. There are three different possibilities.

1. The specialization and the mother part share nothing.
2. The specialization and the mother part share gates.
3. The specialization adds channels which go to/from the mother part.

Let us assume that we have proven reducibility of the mother part. Our aim is to prove reducibility of the inherited type.

The first case is very simple. It suffices to prove reducibility of the specialization part. The second part is slightly more intricate as we in principle need to include fair merge components (with one **merge** state as shown in Figure 67 (p. 111)) wherever the outputs are merged in gates. The third case is the most intriguing as it is necessary to study new potential non-confluence patterns of the mother part since new input channels imply new race conditions.

1. Mn-origin is a double complete state where the generation change take place. Here the pair of sequences of signals are of highest significance.

3.9.2.2 Inheritance of process types

Inheritance of process graphs means that the transition matrix is expanded. During the analysis for confluence this means that for the specialization many of the potential non-confluence patterns have been handled by the analysis of the mother part. They naturally need not be taken again. Thus it boils down to analyzing the new situations defined by the specialization. We need to consider new states with all signal pairs, and old states for new signal types.

3.9.3 Virtuality

Here we shall reengineer the Brock-Ackerman example presented in Section 3.5.4.2 (p. 109) by letting the varying part Pk be a virtual process. Sk and Tk are then stable block types representing the context of the variations as shown in Figure 87 (p. 135).

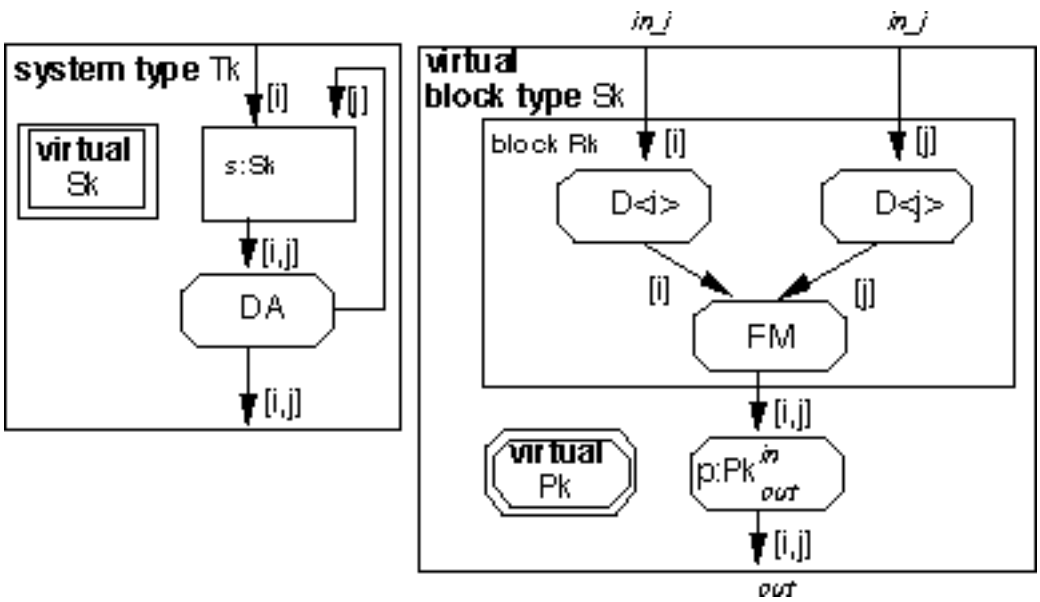


Figure 87: Brock-Ackerman and virtuality

The specific variations can then be described as inherited system types T1 and T2 where the different variants of Pk are redefined as shown in Figure 88 (p. 136).

To analyze the Brock-Ackerman example relative to this object-oriented approach, we start by trying to reduce the remainder of the block type enclosing the virtual Pk. This is depicted by Rk in Figure 87 (p. 135). The remainder is not necessarily reducible as such, but in this particular case Rk *is* reducible. Progress of Rk follows immediately from the signal ordering criterion, and confluence follows from D being incapable of showing non-confluence patterns on any level as it has only one input channel. FM is by definition also confluent for all enclosing levels.

3.9.3.1 The reduced Rk

To reduce Rk and to observe the reduced Rk (Figure 89 (p. 136)) may give valuable insight into the requirements to Pk and the potentials of Sk and Tk.

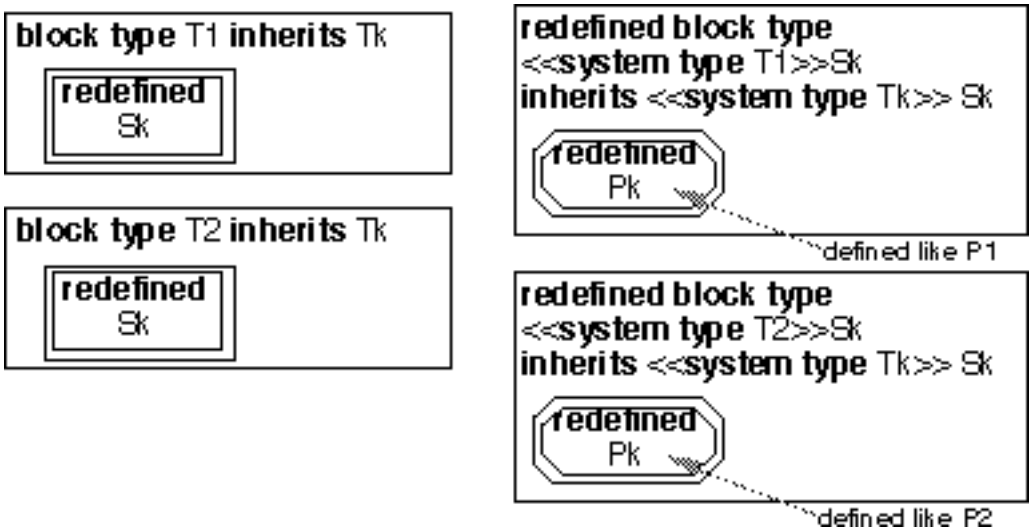


Figure 88: Variants of the Brock-Ackerman example

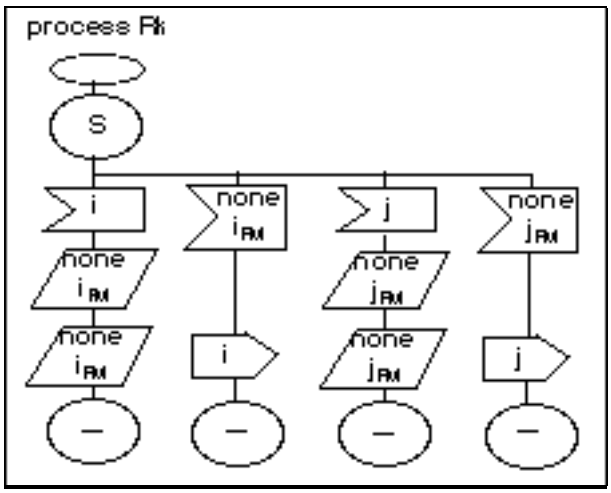


Figure 89: Reduced Rk (Sk except Pk)

The reader should make sure he accepts Figure 89 (p. 136) as a reasonable reduction of the block Rk.

3.9.3.2 Progress of Sk

Sk will be progressive as long as Pk is internally progressive since there are no feedback loops in Sk as such. With variants of Pk, P1 and P2 they are trivially internally progressive since there are no signals internal to P1 and P2.

3.9.3.3 Confluence of Sk

Since Rk is reducible as such, Sk is confluent because Pk cannot include any non-confluence pattern since it has only one input channel. Any non-confluence pattern in Rk relative to Sk would also be a non-confluence pattern in Rk relative to itself since Sk and Rk share input channels.

3.9.3.4 Progress of Tk

Tk has a feedback loop as DA may produce a j which is fed back into Sk, which in turn produces [i,j] onto DA. Since DA produces a j from an i, but nothing from a j, it suffices that Sk does not produce any i's from consuming a j to secure progress of Tk. This sufficient condition may prove to be too strong. Conversely it is quite simple to find a redefinition of Sk (by redefinition of Pk) such that Tk is not progressive. The clue is of course to produce i's from j's.

In our cases T1 and T2, the fact that P1 and P2 actually terminates after having issued two signals ensures progress (or termination) in a very abrupt way.

3.9.3.5 Confluence of Tk

Provided that Tk is progressive according to Section 3.9.3.4 (p. 137), Tk is also confluent since DA cannot contain a non-confluence pattern since it has only one input channel. Sk is reducible and Tk's channels are a subset of the Sk channels. Therefore Sk cannot contain a non-confluence pattern of Tk.

3.9.3.6 Possible restructuring of Tk

If we consider it important to give a description of everything except the (unknown) virtual part, we should try and restructure the block such that the virtual part becomes isolated also in Tk. This is shown in a slightly non-standard SDL-diagram Figure 90 (p. 137) mixing processes and blocks.

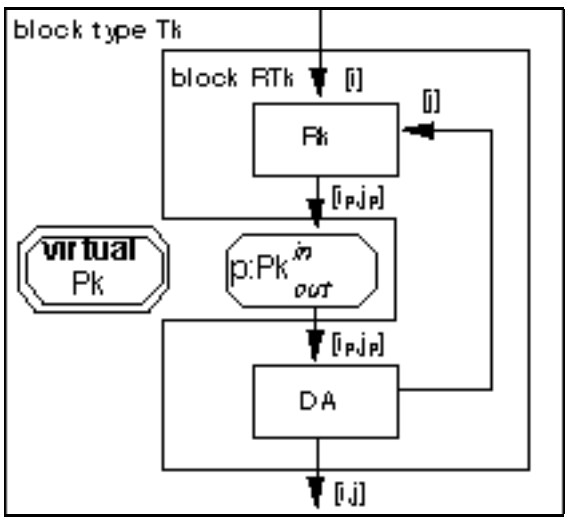


Figure 90: Restructuring of Tk making RTk as Tk except Pk

It is simple to assert that also RTk is reducible and the reduced RTk is shown in Figure 91 (p. 138). In order to avoid confusion concerning the different i- and j-signals, we have annotated some of the signals with FM to signify that we talk about signals internal to the original FM process, and by P to signify that we talk about input/output to/from the Pk process(es).

What can be seen from the RTk is dependent upon the knowledge and the experience of the reader, but at least it shows the whole environment of the virtual Pk in one process.

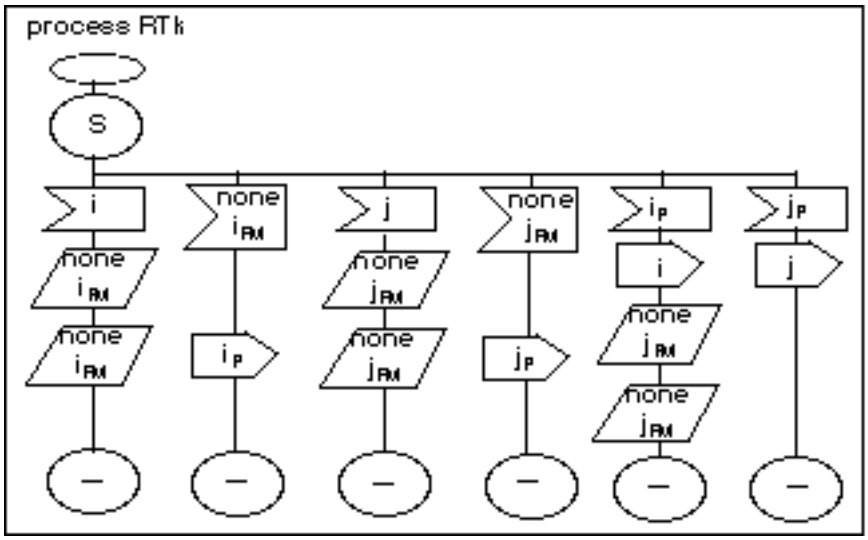


Figure 91: RTk reduced

3.9.3.7 Virtuality constraint

SDL gives the possibility to specify a virtuality constraint for the virtual entity. In fact the default is that the default definition of the virtual also acts as the virtuality constraint. A virtuality constraint is a type which every redefinition of the virtual must have as mother part (on some generalization level). Informally speaking the virtuality constraint is some properties that at least must be present. In our context of reducibility, we are sorry to say that virtuality constraints do not help much. We cannot by a virtuality constraint specify that the redefined entity must be reducible, or that the encloser of the virtuality shall have to be reducible for any legal redefinition. Reducibility, progress and confluence are features that are “fragile” since it is simple to have just a few new pieces of behavior ruin the property.

Still the virtuality constraint plays the same role as any mother part does in inheritance as described in Section 3.9.2 (p. 134).

3.9.3.8 Concluding virtuality

We summarize our attitude towards analyzing types containing a virtual type.

1. Collect all parts not virtual in an entity and analyze it for reducibility.
2. If the remainder is reducible, reduce it and observe the reduced process to get some intuition about the potentials of the system and the variability of the different possible redefinitions.
3. If the remainder is not reducible, this means that the virtual parts have some second generation requirements to fulfill in order for the container to be reducible. These confluence requirements can be made explicit (but not in standard SDL).
4. Consider progress and confluence of the types containing a virtual type by figuring out what requirements the virtual parts must fulfill to secure progress and confluence of the enclosing type. These derived requirements can be made explicit as a (non-SDL) virtuality constraint.

5. The (non-SDL) virtuality constraints can fairly easily be checked for every redefinition of the virtual type. Thus the original analysis work was well worth the job.

3.9.4 Concluding object orientation

The object-oriented features of SDL-92 which serve well to support the design process, give some support for our Mn-approach since reuse of types also means reuse of analysis (wrt. reducibility).

The concept of a pure type is practical, because if the type can be proved reducible, the reduction can be used in stead of the expansion of the type for all instances. More about compositionality of reducibility can be found in Section 4.1 (p. 143).

Simple inheritance may also give some gain in Mn-procedure efficiency since it amounts to reusing the mother part (or large parts of it).

We treat virtuality by conceptually transforming the system such that the virtual parts are outside a “core” part which is checked for reducibility. If there are several virtual parts, it is reason to believe that the core may not be reducible, or the reduction does not really give much gain in perceived complexity.

Virtuality constraints expressible in SDL-92 cannot ensure that the property of reducibility is maintained, which of course is a disappointment. Virtuality constraints mean, however, that redefinitions must reuse large parts of formerly analyzed types which implies that much of the analysis for reducibility can also be reused.

All in all, extensive use of object orientation will make the analysis for reducibility more efficient through the use of reductions and of earlier analysis efforts provided such efforts are properly recorded.

3.10 SDL Service

Services are light weight processes in SDL. They are communicating finite state machines that share the same input port and that execute alternately i.e. only one service of a process runs at any point in time.

That the services share the same process input port is of no relevance relative to our basic model (see Section 2.1.2 (p. 42)) since we consider every channel/signalroute a queue.

That the services execute alternately is not very important either, since we assume that the transitions are atomic in our model. The only consequence is that we do not need implicit fair merge components where the services output to the same channel/signalroute since the transitions have no need to be merged with other transitions which could have run concurrently.

We conclude that SDL services are very much like ordinary SDL processes.

3.11 Priorities

Basic SDL gives neither priority to specific processes nor to specific channels or signals, but the concept *priority input* modifies the picture. There is also the question how eventual priority schemes would influence the Mn-approach.

3.11.1 Priority input

In principle the priority input feature is a way to permute the consumption of signals similar to the effect of save. But the differences are more evident than the similarities. Priority input is significant when there are more than one signal present in the input port of a process and later signals are priority input while the head signal is not. For analysis, however, most often we cannot know whether we have one or more signals in a queue. The Mn-approach emphasizes the independence of signal receptions.

One could think that priority input would play a role in resolving sequence permuted situations on the next generation as the permuted sequences could have a priority order of signals which would harmonize the result. Then the reader should bear in mind that the Mn-procedure does not state that the signals are actually present at the same time, but only that there is a certain sequence of signals on the different channels. At which point in time the signals appear in the sequence is not part of the Mn-procedure.

Rather the contrary, if we have sequence permutation, this means that on one channel the sequences of signals may differ from situation to situation. The signals have to be sent in sequence and we have no guarantee that the receiving process cannot consume them at once. Therefore we cannot ascertain that the second signal is given priority over the first since when the first signal arrives, the second may not be known to the receiving process at all!

The situation is of course different when we have a process which does feedback to itself without delay. Our example process D can be seen as such a process with immediate feedback. If we gave priority in all states to the internal signals (0,1), we would trivially have that the process was confluent since all internal signals produced had to be handled before the next external signal is consumed. Again if the feedback channel was with delay, we are back to the more general situation again where nothing can be certain about the concurrent appearance of different signals.

A special case of the immediate feedback situation is a process which is decomposed into services and the services communicate via priority inputs (and there are no other priority inputs). This will be very close to the SDL-88 [25] definition of services and as noted above the process as a whole will be confluent.

3.11.2 Priority for internal signals in blocks

The immediate feedback situation can also be implemented on a higher level than for each process. If an SDL system (or SDL block) is implemented on one processor (in one operating system task) then it is reasonable to assume no delay on communication and atomicity of each transition.

If we also have a scheme of priority to the internal signals, we find ourselves in a situation very similar to the situation with services in Section 3.11.1 (p. 140). But the situation is not identical as the processes of such a one-task block are still in principle independent and share no input port. Therefore if several internal signals are output from one transition to several different processes, their individual orders are not predetermined. We must also enforce the invariant that the internal (prioritized) signals should be consumed in the order they were output. If there is a central scheduler in the block, this invariant can easily be implemented. The result would be a block which is reducible by its concrete implementation.

We have in our special situations considered only possibilities where the signals are given priority in all states of a process. We have not exploited the possibility to let a signal be prioritized in some states and not in others. In fact our special implementation have given priority to specific (internal) channels rather than to signals.

3.11.3 Concluding priorities

Priorities seems more promising at first glance than after more careful study. It is clear that priority input may often optimize the execution of a system, but the Mn-approach analysis is not automatically made simpler.

Combined with global scheduling of signals, priorities may be used to facilitate achieving confluence and reducibility of a block.

3.12 Concluding Mn-procedure for SDL

In this chapter we have studied how the Mn-procedure must be modified to cope with SDL systems which have more features than the simple process studied in Section 2. (p. 41). To our great satisfaction, the Mn-procedure can be modified through simple means to cope with the most important mechanisms of SDL.

Firstly we established that the Mn-procedure performed well on systems with multiple processes communicating through multiple channels. We found that the Mn-procedure was approximately linear in effort with the number of processes since a non-confluence pattern could only appear inside an individual process.

Secondly we found that the **save** mechanism was actually very practical to ensure confluence. We had to make a distinction between totally stable states and semi-stable states, and correspondingly between weak progress and strong progress. An interesting result was that strong progress very often could be deduced from weak progress and reducibility by studying the reduction.

Thirdly we introduced explicit non-determinism and the Mn-procedure had to work on more complicated set structures. Still the extension made the Mn-approach more expressive. We suggested extensions to SDL which also would make SDL specifications more expressive.

Fairness in non-deterministic decisions makes it possible to determine progress without introducing data. Our definition of fairness has been labelled extreme or probabilistic fairness, and it is an imperative style of fairness. In a non-deterministic decision some branches may be labelled (+) designating a positive probability to happen. We applied this to the Alternating Bit Protocol example.

Spontaneous save was introduced to cope with race conditions which were explicitly acceptable. Typically spontaneous save was used in connection with fair merge components. The spontaneous save (or merge state facility) is only a syntactic reformulation which makes it possible to express race conditions in cases where only one competitor is actually known to be present. This new feature uses a well known SDL mechanism, the save, to express in a finite way an infinite set of merge situations. By applying this notation to the Brock-Ackerman anomaly we showed that the expressive powers are adequate.

Fourthly we showed that timers could also be handled as special cases of non-determinism. Still we realized the general shortcoming of SDL and our approach to cope with real time constraints.

Fifthly we applied a simple program transformation scheme to convert SDL procedures into a system of communicating processes such that procedures could be handled in the same framework as we have already established.

Sixthly we showed that object orientation could give performance effects since reuse of types also means reuse of reducibility analysis. We distinguished between inward use of reduction, meaning reductions used in place of originals in subsequent analysis, and outward use of reduction where the aim is to analyze a type separately and specify more closely what assumptions it makes about its usage environment. We were disappointed to realize (but hardly surprised) that it is not possible for an SDL virtuality constraint to ensure reducibility of the redefined type (or its encloser).

In addition to the above points we also discussed infinite input streams, SDL services and priorities, neither of which had much influence on the Mn-approach as such.

4 The Mn-approach and formal analysis

Experts

Experts have
their expert fun
ex cathedra
telling one
just how nothing
can be done

4. The Mn-approach and formal analysis

Once we have used the Mn approach to reduce a system or a part of a system, how can this new form of the system be used for verification purposes?

4.1 Compositionality of reducibility

Having shown how reducibility of a system (or block) can be performed piecewise (Section 3.3.6 (p. 93)), it becomes interesting to discuss how reducibility results of analyzing components can be used for the analysis of the whole block. We would prefer that when we have found a component to be reducible (on its own), the reduction can be used when the enclosing block is to be analyzed for reducibility. This is indeed the case and this section explains how and why.

4.1.1 Confluence and context

Reducibility is dependent upon the boundaries of the unit which is to be reduced. Even though the Mn-procedure can be applied piecewise, this is not the same as asserting that reducibility of a part is equivalent to reducibility of some enclosing unit. Any reduction is relative to a set of external and internal channels. The distinction between what is external and what is internal may be crucial to the property of reducibility since the whole purpose of reducibility is to show independence of internal actions.

The importance of the distinction between internal and external is easily seen from the block UV shown in Figure 50 (p. 92). By itself process U is non-confluent since the output on c4 may be permuted and when seen as external output this is sufficient to be non-confluent. In the wider context of block UV the channel c4 becomes internal and the process V remedies the permutations of c4 by separating the external output.

We may consider the position of a process relative to confluence of its enclosers. The wider the context, the more sensitive the process is for input channels because external input channels may become internal and thus require independence. On the other hand, the wider the context the less sensitive the process is for output channels as external output becomes internal and therefore subject to allowed permutations.

4.1.2 Defining: compositionality

Compositionality means in general that the result of the analysis of the components are used in the analysis of the whole system.

Compositionality of reducibility in an SDL system means that the result of the reducibility analysis of the components can be used in the analysis for reducibility of the whole system. The result of reducibility analysis is a reduction, a reduced process. Thus compositionality of reducibility means that in order to analyze a full system it suffices to use the reductions of the (reducible) components to replace the original components.

4.1.3 Proving progress

We argue that progress of the whole system can be found by analyzing a system containing reductions of the original components.

Progress is basically the absence of eternal loops of consumption and production of internal signals. When a component is proved to be progressive, it is without eternal loops regardless of in which order the external input signals come. Therefore enclosing this component in a block cannot make the component have an eternal loop inside itself.

Any eternal loop of the block must come from a loop which is at least partially external to the mentioned component.

In such a situation the reduced version of the component is just as good as the original since the signalling interfaces are identical.

4.1.4 Proving confluence

In this section we argue that confluence of the whole system can be found by analyzing a system containing reductions of the original components.

The reader may already be convinced that our compositionality statement above is correct, but we shall go through the argument in greater detail.

For any system which is non-confluent, the non-confluence shows up in its individual components. Therefore it is meaningful to state that a system AAA is non-confluent by non-confluence in its component A.

Assume that we have a system AAA consisting of subsystems AA1 and AA2 as shown in Figure 92 (p. 145). Furthermore AA1 contains a process A. We have established that AA1 is reducible, and the reduction we name AA1Reduced. AAASubst is AAA where AA1 is replaced by AA1Reduced.

4.1.4.1 If AAAsubst *is reducible, so is* AAA

Is it possible that for AAA, non-confluence can show up in A?

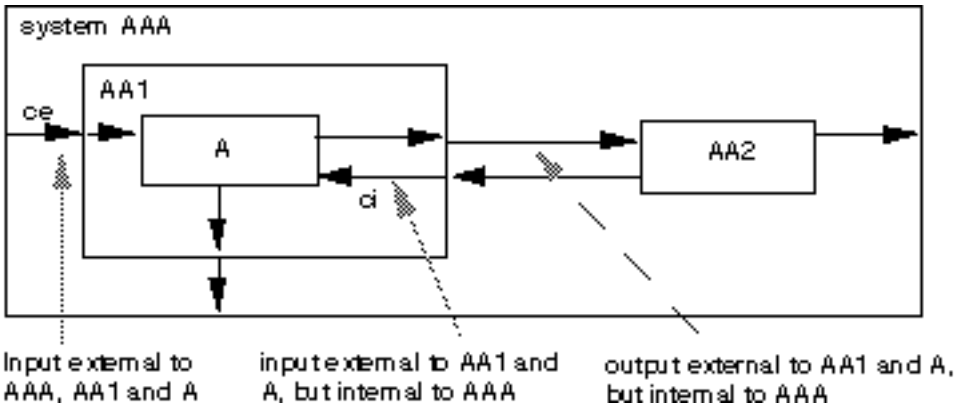


Figure 92: Compositionality

If non-confluence can show up in A relative to AAA, there must be a non-confluence pattern in A consisting of a race-condition with at least one internal queue. We have different cases:

1. The internal queue involved in the assumed non-confluence relative to AAA is also internal relative to AA1. Then the situation has been covered when A was analyzed relative to AA1. Since AA1 is reducible by assumption, there is no such non-confluence pattern within AA1.
2. The internal queue involved in the assumed non-confluence relative to AAA is external relative to AA1 described in Figure 92 (p. 145). This means that the pattern has not been considered during the analysis relative to AA1.

The question with 2 above is whether the situation has been considered during the analysis of AA1Reduced relative to AAA.

For the non-confluence pattern to be really interesting it has to be reachable which means that there is an initial state of the whole system AAA (which includes the initial state of A) with a sequence E of appropriate AAA-external signals such that some execution of it will reach the assumed non-confluence pattern and from there two different sets of stable states relative to AAA can be reached.

1. Assume complete non-confluent state of A: $N_A = (S;x_{ce};x_{ci}\varphi;\theta)$ reachable in AAA. (Irrelevant other parts of AAA has been omitted in the complete state shown)
2. Assume N_A reached by applying the external signals E onto the initial state and performing an appropriate sequence of internal transitions.
3. Assume that L_1 is the result of applying to N_A the signals $x_{ce}x_{ci}$ and stabilizing. Assume correspondingly that L_2 is the result of applying to N_A the signals $x_{ci}x_{ce}$ and stabilizing. We have that $L_1 \neq L_2$.
4. Then it is obvious that L_1 and L_2 are reachable in AA1 only be projecting the execution within AAA onto AA1. The involved external signals relative to AA1 to reach L_1 is $E+x_{ce}x_{ci}\varphi$ and to reach L_2 it is $E+x_{ci}x_{ce}\varphi$.

5. Applying E to AA1Reduced will bring it to some state X. Then applying $x_{ce}x_{ci}\varphi$ will bring it to L_1 since L_1 is stable and the reduced process is identical to the original when external signals have been applied and the result stabilized. Applying $x_{ci}x_{ce}\varphi$ to X leads to L_2.
6. Then we have two different execution results from X depending on the permutation of x_{ce} and x_{ci}. Since ci is internal relative to AAA, this means that X is a non-confluence pattern of AA1Reduced relative to AAASubst.
7. If AAASubst has been shown to be confluent relative to AAA, such a state X cannot occur.
8. Thus the conclusion is that confluence can be established in a compositional way.

4.1.4.2 If AAASubst *is non-confluent, so is* AAA

This is trivial. If AAASubst has a non-confluence pattern, this must show up in either AA1reduced or in AA2. If it shows up in the latter, it will of course also show up in AA2 in AAA.

If the non-confluence pattern is in AA1reduced, the same non-confluence pattern must occur in AA1 of AAA since AA1 has a path corresponding to every path of AA1reduced due to the way the reduction algorithm works.

We must conclude that non-confluence of AAASubst implies non-confluence of AAA.

All together we have that AAASubst is confluent iff AAA is confluent.

4.1.4.3 The reduction of AAASubst *is the same as the reduction of* AAA

That we can use AAASubst as the base for reducing AAA is also almost trivial.

We assume that we have found AAA reducible by finding AAASubst reducible. Assume that we want to perform the reduction algorithm on AAA. Whenever there are internal signals which are also internal signals of AA1 we execute these. This is legal because the reduction algorithm opens for executing any internal signal as pointed out in Section 3.2.4 (p. 89). By this strategy the reduction of AAA will include elements of the reduction of AA1 into AA1reduced and thus executing the result of that reduction (namely AA1reduced) must give the same effect.

4.1.5 Concluding compositionality of reducibility

Since both progress and confluence are compositional wrt. reducibility, reducibility is also compositional wrt. reducibility.

In enclosing blocks, reduced versions of components may be used when analyzing reducibility. The reductions can also be used in the reduction algorithm.

4.2 Verifying refinement

Refinement has become a buzzword in software engineering. The idea is that by giving a description on a high abstraction level first and then giving corresponding descriptions on lower abstraction levels afterwards, it should be possible by formal means to prove

that the descriptions actually correspond. If the increments can be made small, it is supposedly possible to move from abstract to concrete solutions and keep the description complete and consistent at all times.

As a practical way to design systems, we do not believe in this paradigm, as we describe in more detail in Section 5.1.3 (p. 180). Still the idea of refinement is a very attractive and practical concept. It is beyond doubt that there will be both abstract and more concrete descriptions of the same system, and there is a definite need to keep the descriptions inter-consistent. In this section we shall give an example how the Mn-approach can be used to support verification of refinement.

4.2.1 The Refinement model

SDL

We assume that we have an abstract description in SDL and a more concrete description also defined in SDL. In general refinement may also be defined as a relation between descriptions of different languages, but relating different languages is beyond our current Mn-approach. To compare descriptions of different languages, it is necessary to have a common semantic base to which both descriptions could be translated. It is possible to suggest that CFSMs (processes) could constitute such a common base. Here in this thesis we shall limit ourselves to descriptions in SDL.

Definition

We define *refinement* to mean the following. By R refining M, we will understand that if an environment E acts with R or M, any behavior of E with R should also appear in E with M. By *implementation* we shall mean the same as refinement. The two words appear both in the literature and sometimes they may have different meaning, but here we give them the same meaning. We summarize this definition in Figure 93 (p. 147).

Let R and M be two processes $<S_R;C;Z_R;T_R>$ and $<S_M;C;Z_M;T_M>$ with the same external interface shown here by assuming the same alphabet.

R is a *refinement* of M **iff**

1. $\exists m \forall (s \in S_R) \bullet m(s) \in S_M$ **where** $m : S_R \rightarrow S_M$
2. $\forall (e \in E) \forall (s \in S_R) \bullet m(L_R(s;e;\emptyset;\emptyset)) \subseteq L_M(m(s);e;\emptyset;\emptyset)$ **where** the m function is extended to complete states and sets of complete states in the obvious way.

By *implementation* we mean the same as refinement.

Figure 93: Refinement (implementation)

The idea is that any behavior of the implementation is also a behavior of the more abstract description. This means that the implementation is always a restriction of the abstract description. In practice this seems often too limited as implementations often offer features which was not thought about in the abstract version. This is a practical and methodological question and we shall cover this in more depth in Section 5.3.4 (p. 205). In this section we take for granted that the two descriptions do talk about the same universe of discourse and that the descriptions are directly comparable wrt. which services they provide.

Behavioral refinement

We are interested in a behavioral refinement, which means that we need to have a common representation of the two descriptions which we shall compare and we shall use the process form as our canonical form and try and see if one process is a sub-automaton of the other by comparing the processes transition by transition corresponding closely with the definition given in Figure 93 (p. 147). This is illustrated in Figure 94 (p. 148), and

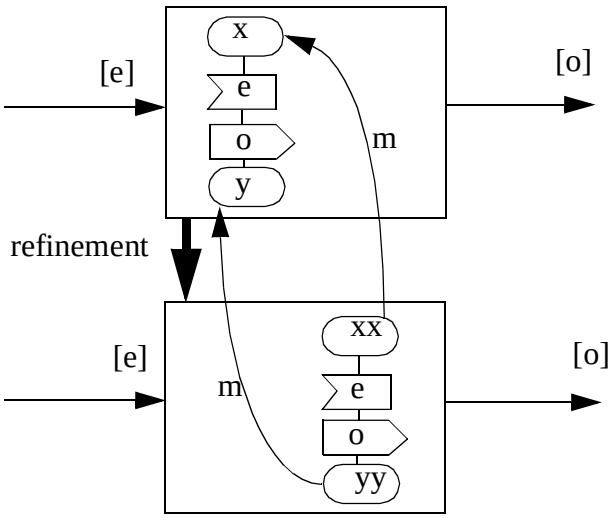


Figure 94: Behavioral Refinement

described more closely in Section 4.2.2 (p. 149).

Interface refinement

Behavioral refinement in its pure form requires that the abstract and the concrete description have the same signal interface. It is typical, however, that more abstract descriptions have interfaces of a higher granularity than the more concrete descriptions. When the behavior is described in more detail, also the pieces of communication must be detailed. When the interfaces of the abstract and the concrete descriptions differ, we talk about *interface refinement*. Our model of interface refinement is largely inspired by the same concept of FOCUS [20] and it is schematically shown in Figure 95 (p. 148).

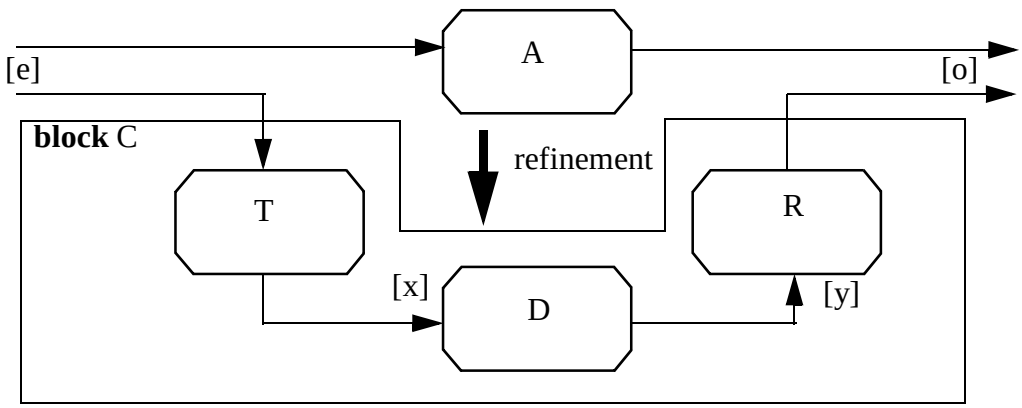


Figure 95: Interface Refinement

The idea is that we add two more SDL processes, T and R, which transform the interfaces. T transforms from the abstract input interface to the concrete input interface, and R transforms from the concrete output interface to the abstract output interface. We say

that A is interface refined to D subject to the interface mappings T,R. This corresponds to A being behaviorally refined to C where C consists of T, D, R. This corresponds to the U-simulation as defined in [35].

For our purpose it is equally applicable to combine the interface mappings with the abstract description A, and let the concrete description D stand alone. This would correspond to a U^{-1}-simulation according to [35].

Compositionality

Refinement is compositional (see Section 4.1.2 (p. 144)). Assume that the abstract description A appears in an enclosing block AA. If we substitute A by C where C is a behavioral refinement of A, the resulting block AC is a refinement of AA.

This is quite obvious. Firstly AA and AC have the same interfaces since the interfaces have not changed due to the internal substitution of A by C. Furthermore as every behavior or C is also a behavior of A, every behavior of AC must also be a behavior of AA, and that is the definition of refinement.

We may want to go one step further. Assume that AA consists of abstract descriptions A1, A2 etc. connected. Assume furthermore that for every abstract Ai there is an interface refinement Di with interface mappings Ti and Ri. Assume that DD is the system where every Ai is substituted by a Di. The question is whether DD is a refinement of AA. This cannot be concluded in general. It is not even certain that DD is a valid system since we do not know whether the interfaces match.

Our approach would be the following. Substitute in AA every Ai with the corresponding Ci. Then we know that the resulting system CC is a refinement of AA. In CC there are clusters of Ti's and Ri's which help combine the Di's. In DD these clusters are reduced to mere channels. The question we ask ourselves is whether these clusters can really be reduced to channels. A channel is some kind of identity process, what comes in, has to come out. The question is whether all these clusters have the proper identity process as a refinement. With the Mn-approach we would try to reduce the clusters and compare the resulting process with the identity process. This would amount to finding out whether the reduced cluster could possibly act as an identity process.

4.2.2 Mn-approach

We now summarize the Mn-approach to (interface) refinement.

We assume that we have SDL descriptions A and D. We want to determine whether A is (interface) refined to D.

Sub-automaton

1. Reduce A and D to two process descriptions. If any one of them is not reducible, our Mn-approach is not the right thing to use, or the design is possibly wrong?
2. Specify the interface mappings T and R. Strictly speaking there is no real reason to keep closely to the scheme of Figure 95 (p. 148). T and R may be any SDL processes such that the interface of block C is identical to the interface of A. Thus there is technically no obstacle to R receiving input from T in addition to D.
3. Reduce block C (consisting of T,D,R). This is usually a simple task if T and R are mere transformations of signal formats.

4. Compare the reduced A and the reduced C transition by transition (if possible). The basic state mapping is built during the comparison (see below).
   - 4.1 Map start symbol of A to start symbol of C.
   - 4.2 Compare start transitions of A and C.
   - 4.3 Map the nextstates of the start transitions.
   - 4.4 Take one of the basic states of C which is not analyzed. Find the corresponding state in A. For every input signal do:
     - Compare the two corresponding transitions.
     - Map the set of nextstates of the C transition to the set of nextstates of the A transition.
     - Repeat this step until there are no more basic states in C which are not analyzed.
5. If all transitions of C has been analyzed and no critical discrepancy has been found, then we may conclude that A is refined to C.

Compare transitions

We still have to be more precise about what the comparison of two transitions boils down to. The following criteria should be sufficient. We compare the transitions by going through the C transition in the order of execution.

1. Every output of the C transition has a corresponding output in the A transition.
2. Every decision of C has a corresponding decision of A.
3. Every answer to a decision in C has a corresponding answer in the corresponding decision in A.
4. Any answer to a fair decision in A with positive probability (+) should have a corresponding answer with positive probability in C. An exception to this is if the positive probability is not needed in C to terminate a feedback loop.
5. It may be necessary to do some local semantics-preserving rewriting to cope with tasks, timers and procedures.

Ad hoc adaptation

Our technique will not be able to prove all proper refinements, but it catches interesting ones and it applies the general Mn-approach. The final comparison between the processes can obviously be improved and it is possible to utilize other formal methods to prove refinement between two processes such as e.g. FOCUS if it is sufficiently important. If we assume restrictions on the size of channels, variables etc. we turn into the pure finite state machine situation and the model checking methods mentioned in Section 1.6.2.1 (p. 30) applies.

Typically the structure of decisions may vary between A and C. The ad-hoc approach is to separate the analysis in two where the finite behavior is analyzed first and then the infinite behavior. In the analysis of the finite behavior a hierarchy of non-deterministic decisions can be flattened to one non-deterministic decision with a set of alternatives. If the analysis of finite behavior succeeds, the infinite behaviors of C are considered in A. If the infinite behaviors of C are possible also in A, we have correspondence between the decision structures of this transition.

4.2.3 Example: the rejected or accepted signal

We want to give a very simple example which shows the principle behind our approach to interface refinement. We could not resist the temptation to show that a quite reasonable design is easily proven wrong during our search for reducibility. After having corrected the flaw we give the proper concrete implementation and show the interface refinement.

Our example is a component which takes some input and either accepts it or rejects it. The abstract definition is given in Figure 96 (p. 151).

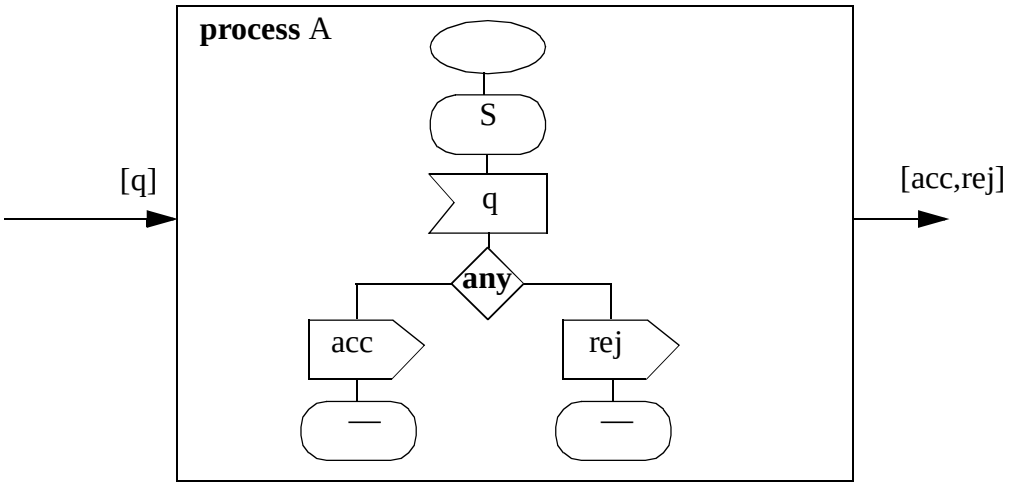


Figure 96: The abstract process A

We want to implement this in a system which takes a number as input and decides whether the number is above, below or between two given bounds. The input number consists of a sequence of digits d terminated by a point p. The system should of course handle an unbounded stream of numbers. The system has two equally structured bound checkers, one checks the upper bound and one checks the lower bound. Their results are combined by the outputting process. The structure is given in Figure 97 (p. 151).

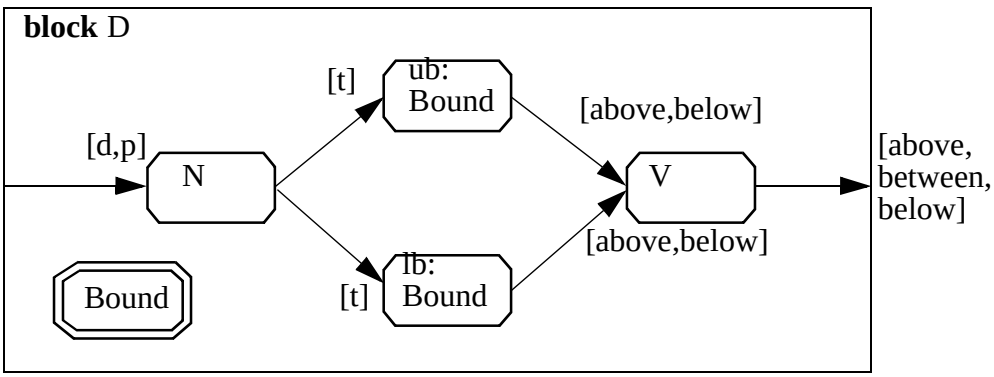


Figure 97: Structure of the implementation

Process N compiles the digits and when the point arrives, calculates the number and passes it on to ub and lb as shown in Figure 98 (p. 152).

The boundary check type Bound just checks and returns above or below as specified in Figure 99 (p. 152).

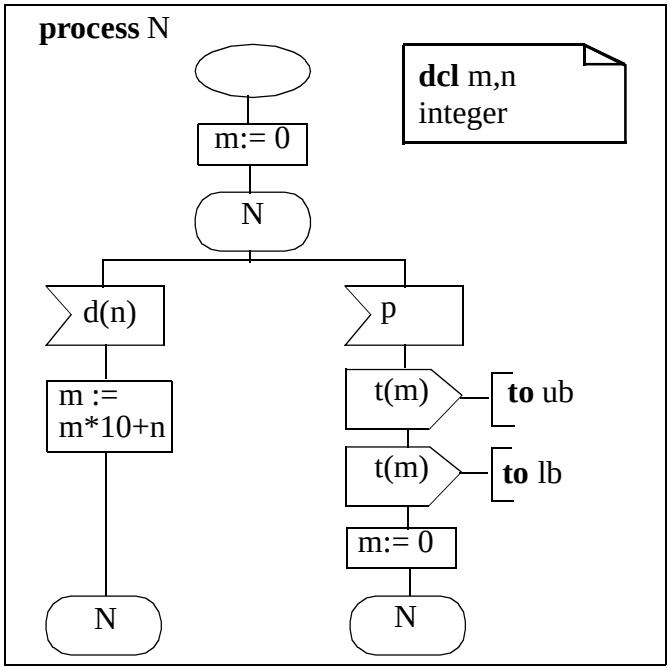


Figure 98: Process N: compiling the number

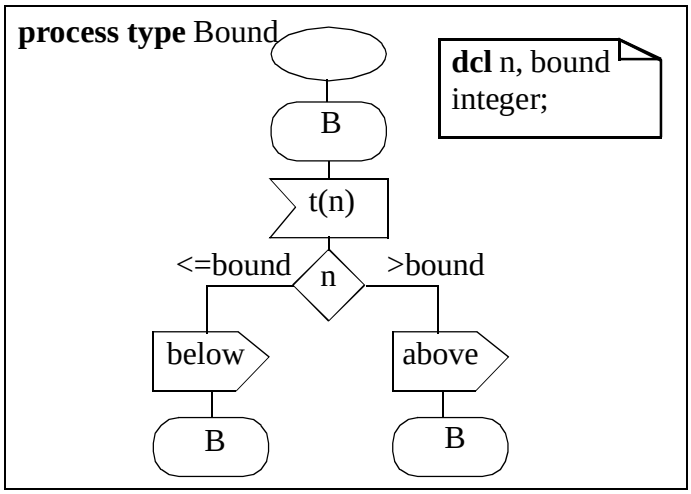


Figure 99: Bound: the bounds checker

Now finally we want to design V, the verdict producer. V should take one signal from ub and one from lb. If it gets two above-signals, the verdict is that the number is above the upper bound. If V gets two below-signals, the verdict is that the number is below the lower bound. If it gets one above and one below, the number must be between the bounds. This informal specification should lead to the definition of V given in Figure 100 (p. 153).

We follow our strategy outlined in Section 4.2.2 (p. 149) and try and see if the block D (Figure 97 (p. 151)) is reducible. Firstly it is obvious that it is progressive since there are no feedback loops, and there are no internal data loops which are independent of external signals. Secondly, then we should have a look at confluence. N, ub and lb are definitely confluent wrt. D since they have only one input channel each. The potential problem must be with V.

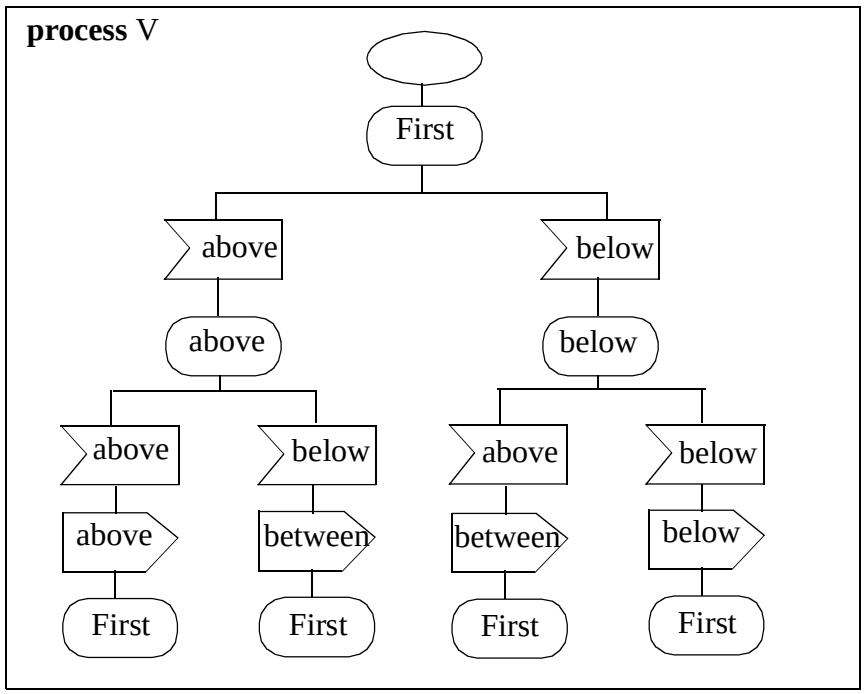


Figure 100: Process V: the verdict, first attempt

No potential non-confluence patterns with basic state First give problems, but for states Above or Below there is non-confluence! Our first thought may be that we have to do with states which are unreachable, but alas, they are very reachable. The problem is that V is not ensured that the inputs alternate between ub and lb. Due to asynchrony and the concurrent processing of numbers, it is quite possible to get a series of signals from ub before the first from lb. This will obviously lead to unreliable results! This example gives us valuable experience:

- The solution which meets the eye may not be correct.
- When we have merge situations (i.e. more than one input channel and decisions based on signals from more than one), confluence is an important criterion.

Here we shall have to restructure. We find that it is important that V can control which signal it gets. To make it as simple as possible, we want to force it to alternate between the channels. Still we need a way to distinguish between the channels, and in SDL there is no sign of the channel in the signal. Therefore we must make sure that ub and lb send different signal types. This can be done easily by letting Bound be parameterized wrt. signals above and below. The modified version of V is given in Figure 101 (p. 154).

In the modified version we use **save** to be certain that we alternate between the upper bound and the lower bound. We always start with the upper bound. Since we can be fairly sure that the two bounds checkers execute at comparable speeds, this deliberate sequencing is acceptable. If their execution times both varied, better throughput is achieved by a version where either upper bound or lower bound could be taken first, but that the second input always had to be the other.

That the system D is confluent is now obvious and we may reduce the block. This is automatic and result in the process definition found in Figure 102 (p. 154). We have given the single basic state of the reduction the name New. The reader should notice that

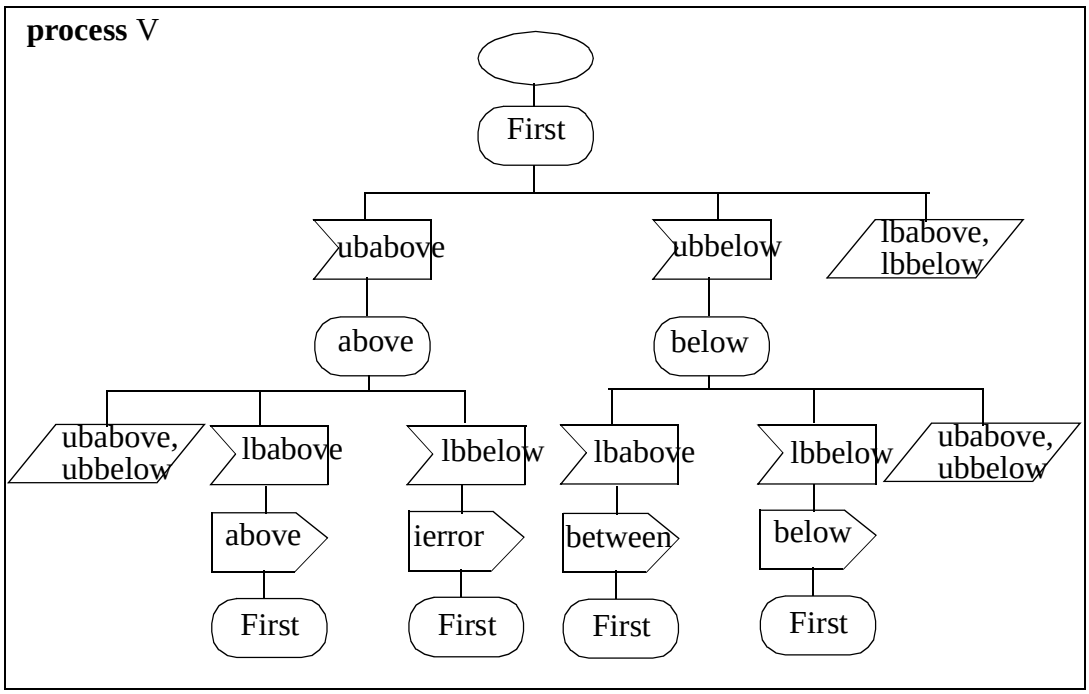


Figure 101: Process V: the verdict, second attempt

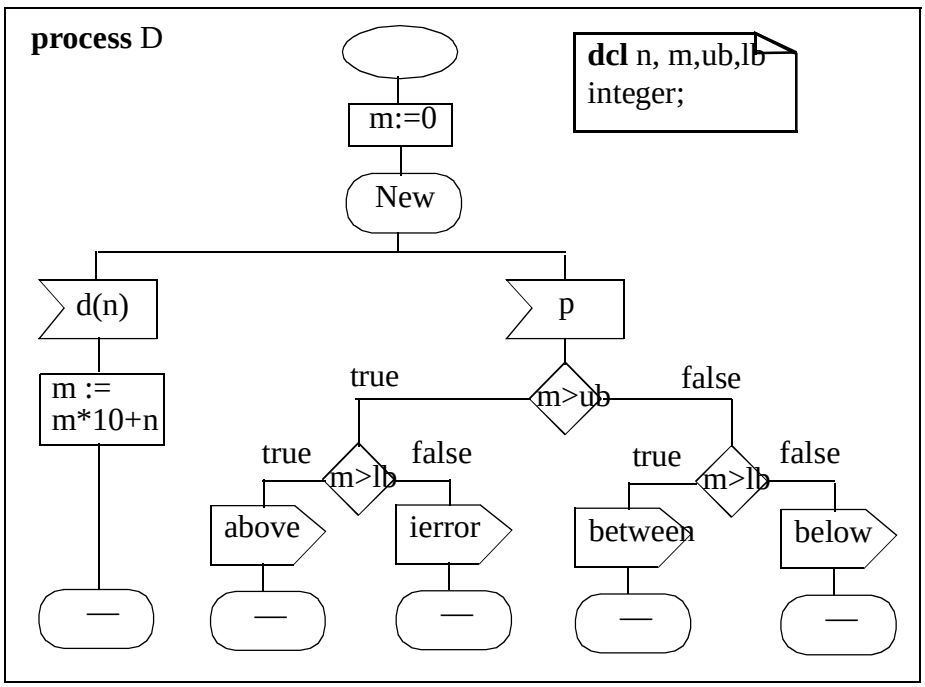


Figure 102: The reduced process D

we have paid no attention to the initialization of ub and lb, which are the constant bound. So far so good. The question now is whether this reduced process D of Figure 102 (p. 154) is a refinement of the abstract process A of Figure 96 (p. 151)? We make a configuration as sketched in Figure 95 (p. 148) and try and reduce block C. Progress is no problem since T, D and R are each progressive and there is no feedback loop between them. Confluence is also simple since T, D and R have only one input channel each.

We suggest the definitions of T and R in Figure 103 (p. 155). T takes the signal q and

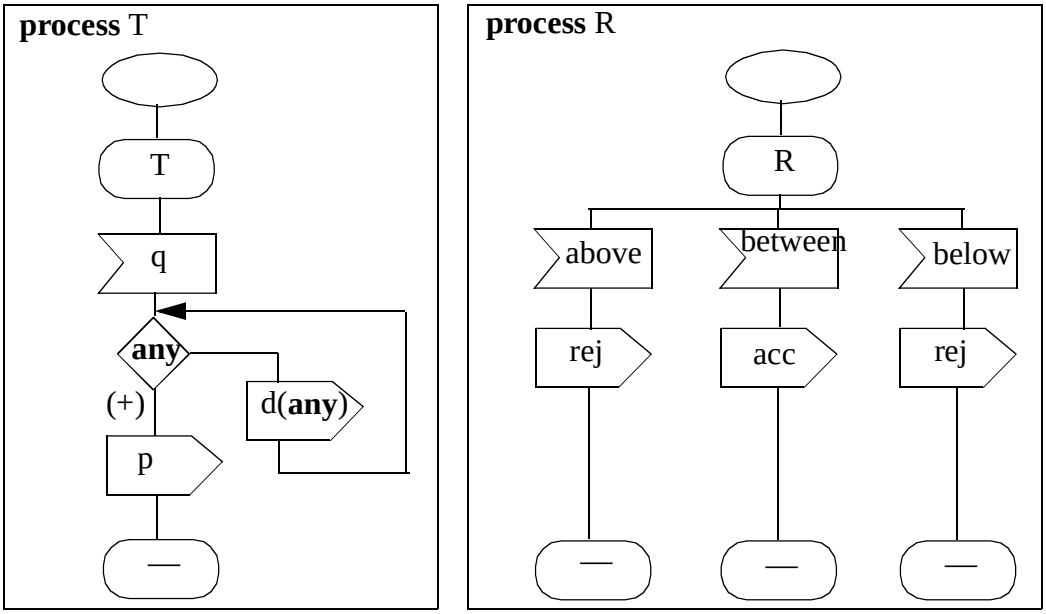


Figure 103: Interface mappings T and R

produces a non-deterministic sequence of digits before the p. Notice that both the number of digits and their values are non-determinant. Notice also that the loop producing digits are terminated by a fair decision with a positive probability alternative. R simply defines that the number should be between the bounds to be accepted.

The reduction of C is shown in Figure 104 (p. 155). We compare it with the abstract pro-

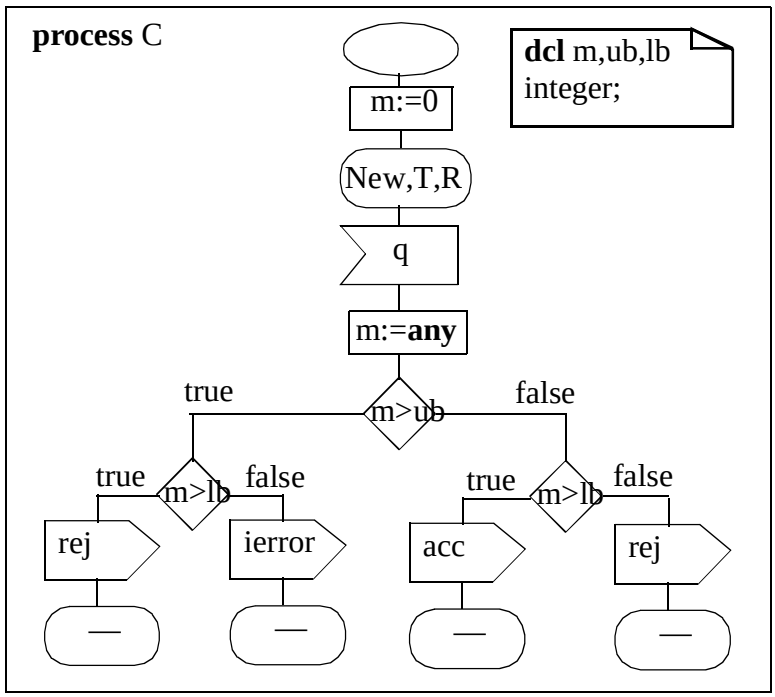


Figure 104: The reduced process C

cess A of Figure 96 (p. 151) by the scheme in Section 4.2.2 (p. 149). The mapping of states is simple as there is only one for each of the processes A and C. S maps to New.

We need to apply some simple, manual arguments to compare the two transitions. An expression comparing something which has been produced from **any**, is very close to **any**-valued itself. The implementation has introduced an extra alternative which ends in internal error. This is not according to the refinement rules since all alternatives of the implementation should find its counterpart in the abstract definition.

Therefore we have to conclude that D is only conditionally interface refined from A with interface mappings T and R. The condition is (as usual) that the execution does not end in an internal error. Actually A and D are conditionally interface equivalent.

4.3 Simplification

In more general terms both Mn-reduction and refinement are examples of a wider class of transformations which may be called simplifications or abstractions. The goal is to find a system which has the same properties as the original in specific areas.

The original is not analyzed as such because the method used for analysis cannot handle it. The reason for this is either that the original model is outside the range of the analysis technique or that the original model is too complex.

An example of a model which is outside the range of the analysis method is when the original model is infinite state while the method can only handle finite state.

An example of a model which is too complex is typically exhaustive search in a real SDL system modeling a telephone switch.

4.3.1 Conceptual clarification

We shall make a few distinctions and separate between some subclasses of simplifications.

1. Pure simplification
2. Abstraction
3. Projection
4. Optimization

We shall present these classes, but shall not go in great detail about this general area which is an important field of research.

Our distinction between subclasses of simplification are more based on their use in the software engineering development process than the theoretical differences. The overall framework is depicted in Figure 105 (p. 157). The original system is simplified via a simplification procedure into a simplified system. In the simplified system it is possible to prove abstract properties relative to a purpose such that the corresponding concrete properties are implied by the abstract ones.

The simplification purpose is crucial. The simplification procedure is dependent upon the purpose and so are the abstract and concrete properties.

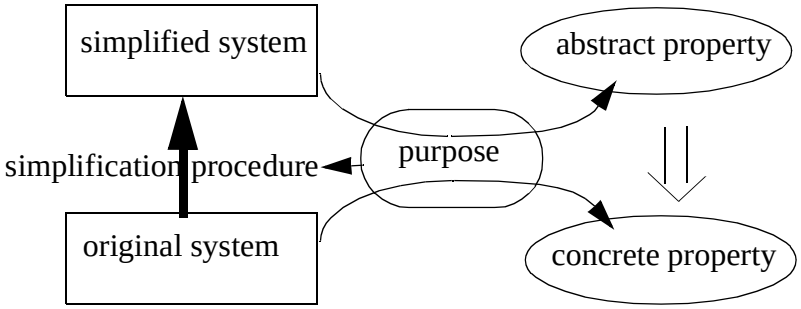


Figure 105: Simplification framework

4.3.2 Pure simplification

Pure simplification means that the simplified system covers all aspects of the original system, it is just more clever built. The simplification purpose is universal: to cover all (interesting) properties of the original. There should not be one single situation where one would rather have the original than the simplified.

This does not mean that they have exactly the same properties, since the original is still different from the simplification, but the difference is not beneficial in any known way.

This kind of simplification occurs when new research discovers new algorithms which are better than the old ones. We mention this extreme kind of simplification since it represents one end of the simplification spectrum.

The Mn-reductions may lead to pure simplifications. It may become clear after a few iterations using the Mn-approach that the distribution of a certain activity into several sub-processes may not be needed or beneficial. Then the reduction could take the place of the reduced unit also in the original.

4.3.3 Abstractions

The word abstraction is used in a number of different contexts. When abstraction is intended to mean simplification we are often in more formal contexts. The simplification procedure is represented by an abstraction mapping and the purpose is often to preserve properties described in a certain language or such general properties as deadlock freedom [21].

The preferred engineering strategy is to define abstraction mappings and reason generally about their characteristics.

When a specific analysis problem is encountered, the abstraction mapping is applied, and some abstract properties (faithful to the purpose) are proved. Then the characteristics of the abstraction mapping leads to finding concrete properties which are satisfied in the original system

The significance of the simplification is relative to the purpose, which often is wider than the abstract properties. In this way the abstraction can be used for experimentation on a wider basis than only predefined properties.

In approaches which combine different proof techniques [126; 127] there is a need to divide the original system such that the different approaches can be applied to only one aspect. Typically abstractions are made which abstracts to finite state situations, e.g. by factoring out the induction aspects which generalizes over numbers.

In formal contexts the abstraction mappings are formally defined and the proofs can in principle be performed formally. In practice, however, abstractions are made more informally and it is not always so obvious what their characteristics are.

Since formal methods have little success with large programs, it is commonplace to define a simpler system. The construction of the simpler system is meant to be an abstraction wrt. the original purpose, but the truth is often that the simplification is done manually by the verifiers mostly guided by what they are able to verify rather than what the needs for verification are.

Refinement is the inverse of abstraction as presented in Section 4.2 (p. 146). The idea is that the simplified (abstract) system is made first, and the refinement which corresponds to the original (concrete) system is made afterwards. The abstraction purpose is that the behavior of the refined system is also possible in the abstract system. This means that universal properties of the abstract system should be preserved in the concrete, while existential properties of the abstract system may not be preserved since there is behavior in the abstraction which is absent in the implementation.

4.3.4 Projections

While we characterized abstractions by the focus on preservation of a wide class of properties, projections are characterized by focus on the simplification procedure.

The simplification procedure is well established, but it may be more fuzzy what the characteristics of the implicit abstraction mapping are.

We have suggested abstractions for data in Section 3.6 (p. 117) which is obviously practical for the simplification of the system, but the characteristics of the data abstractions wrt. reducibility is not absolutely trivial.

Assume that we have applied the projection described in Section 3.6.1 (p. 117) to eliminate all explicit data. It is clear that the simplified system may be reducible while the corresponding original is not. In Figure 106 (p. 159) we show extracts of a process which consumes external input e and internal input i producing external outputs x,y on one output channel. The behavior is dependent upon a Boolean variable b which is not changed in the transitions shown here. The Mn-procedure shows that there is non-confluence since regardless of whether b is true or false, there is a difference between executing e first or i first. Assume that b is **true**, then executing e first yields xy, and i first yields yx. If b is **false** the situation is the opposite.

If the data variable b is abstracted, the decision becomes a non-deterministic decision where either branch can be executed. This gives confluence since both executing e first and executing i first yields the set {xy, yx}.

Thus reducibility is *not* preserved by the data abstraction. Still it is not without value to perform the data abstraction, but our example shows that we need to be sober in our generalizations.

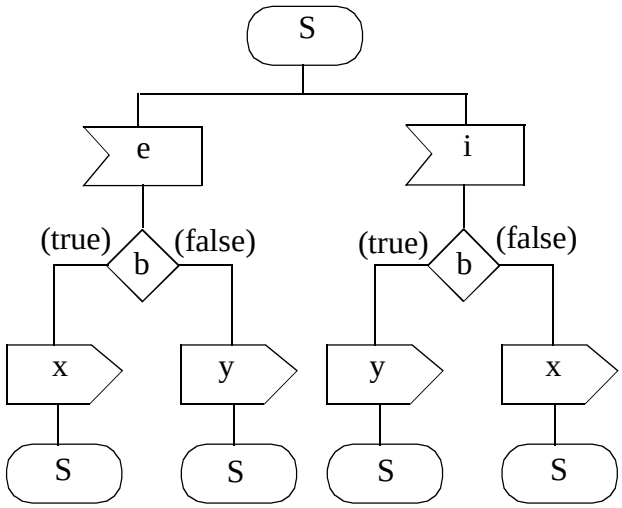


Figure 106: Non-confluence pattern eliminated by abstraction

Other simple abstraction procedures include elimination of “uninteresting” signal types or parameters to signals.

This kind of abstraction, which we choose to call projection, because we deliberately eliminate certain syntactical elements of the original system, has its great worth in *suggesting* problems rather than proving the existence of certain properties.

Lam and Shankar [95] defined projections which they called *image* specifications. They were based on manually subdividing the state space of the processes. They also showed how stepwise refinement of the image specifications could be applied to produce the proper image specification strong enough to verify a given abstract property. In general their image specifications would preserve safety and liveness properties of the original.

Seltveit [121] describes *filters*, which are also projections (in our terms). Their main purpose seems to be to present a complicated original system in a more manageable way to the developers. This is also a purpose we have with our Mn-reductions. She also shows that certain modifications made to the projections (filtered systems) can be faithfully brought back to the original.

Bræk [11; 12] has developed a projection technique to study interfaces between SDL entities. The technique also includes simplification of both sides of the interface. The technique will calculate a measure of how great the risk of complication is, given certain signs of inconsistency. The idea is that the risk situations should be analyzed again in the original. The risk index of [12] resembles our notion of a complexity profile based on the Mn-procedure as presented in Section 5.2.2 (p. 193).

The Mn-approach is a projection in the tradition of Lam and Shankar, but the reduction of the state space is produced automatically provided that the original system is confluent. We claim that any property expressible in terms of the reduced system is also preserved in the original system limited to its stable states. This is basically a tautology since a reduction is defined (Section 2.2.1.1 (p. 48)) as being equal to the original for the stable states. Still it is what we want. From outside the system, the instable states cannot be observed.

4.3.5 Optimization

By *optimization* (here) we mean that the process of verification is optimized. More precisely: from a given a concrete property produce a simplification procedure which makes a simplified system (on the fly) such that proving the property (or its derived abstract correspondent) becomes simpler in the simplified version of the system.

The idea here is to focus on the actual property to be verified, and transform the system relative to the concrete property. This is a meta-strategy used to optimize verification and Holzmann [75] reports that using a partial order reduction method based on the given LTL property, reduces the execution time and space by between 10% and 90%. The idea is that for a given correctness criterion many execution sequences are indistinguishable and it is not necessary to visit more than one representative of each such class of sequences.

4.3.6 Simplifying large numbers

Finally we want to touch upon a special kind of simplification which is very common in systems in our domain (i.e. real SDL systems). Real systems are big. This means that the description of the system is big – and that the executing system is big. In systems with dynamic structure the size of the executing system may not be reflected in the size of the system description. Our validation should have a complexity which is proportional to the size of the description of the system and not the size of the executing system.

In SDL systems there are two constructs which create executing systems which are significantly different (and larger) than the description.

1. Block sets.
2. Process creation and addressing by pointers (PId[1]).

4.3.6.1 Block sets

Block sets are described by one symbol which covers a set of statically created block instances. All channels connected to the block set represent sets of singular channels. For the Mn-procedure block sets do not represent serious complication. Since the Mn-procedure checks channels in a pairwise fashion, there is little complication related to whether there are 20 rather than 2 block instances in a block set. The distinction is, however, that there are two rather than one instance. This means that what appears as one channel as input to another block, is actually two channels, and their interaction should be analyzed. During such analysis the signals should be annotated by some identity of the SENDER or the channel on which they came.

Multi-gate

The architecture is typically like Figure 107 (p. 161). The block set is connected to some common part X by a channel C. The channel C is therefore what we could call a *multi-channel* since it contains a set of normal channels. The same holds for the channel D and therefore also for the gate g. We will therefore call g a *multi-gate*, which means that it actually contains a set of normal gates. We assume that every individual gate can be

1. PId = Process Identifier, SDL data type which represent pointers to processes

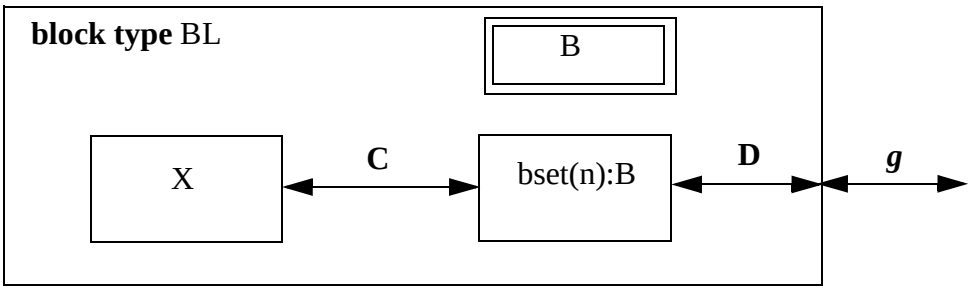


Figure 107: Block set architecture

identified such that the signals can be annotated by the identity of the individual gate. For input, very often such identification can be the PId of the SENDER of the signal on g.

Independence

The next question is whether the communication via the block set bset is such that it is significant that there are several Bs in the bset. We find often that the different blocks of the block set are independent. By *independent* we mean that the behaviors of one block instance has no direct effect on the behavior of others. Even more precise we say that a block set is independent if for every consumption of one external input in the enclosing block type only one block of the block set need to be executing.

In an architecture as in Figure 107 (p. 161) this means that an input on g will be handled by some block (say: b) in bset which communicates with X via C. The eventual return will pass through the same block b onto the same gate of the multi-gate g. If the return communication involves another block of the blockset, the block set is *not* independent.

If reducibility has been established, independence is easily seen when performing the reduction.

Block set representative

When independence of a block set is established, the functional behavior of the block set can be determined from assuming only one block instance in the block set.

Block set reduction

The reduction of a block (type) enclosing an independent block set consists of an extended SDL description. Firstly there is the reduced block where only one block is seen as representative for the whole block set. This system we call the *simple reduced system*. Secondly there is a state-vector indexed by the individual identifiers of the multi-gate which corresponds to block identifiers of the block instances. The state the reduced total system is in, is depending on which individual gate the external input comes via. Because of the independence and the confluence, the simple reduced system will show the needed transitions. The interpretation of the full system reduction is that any state operation (such as nextstate) is performed on the state-vector and any operation on variables inside the blocks of the block sets must also be considered operations on an element of such vectors indexed by a process identification.

This composed description is a shorthand for a reduction with many states and many (similar) transitions. If each of n blocks of a block set may end in k basic states when reducing, we have n*k stable states. The n may not even be known.

Practitioners' induction

We summarize our approach to block sets in a principle which we call *practitioners' induction* which is based on very simple experiences:

1. A real system is never such that a block set with large n makes a significant functional difference from the same set with rather small n.
2. The individual blocks in a block set behave independently. Therefore we can describe the overall functionality with a simple system where the block set is represented by one singular block.
3. If a block set is not independent, there is good indication that the block set should be conceptually divided.

The experience behind what I have called "practitioners' induction" was also probably the reason why SDL-88 had no block set concept at all.

There are examples of systems conforming to the practitioners' induction in Section 6. (p. 229).

4.3.6.2 Process creation

Process creation and use of PId addressing does not make problems for the Mn-procedure to determine confluence. The Mn-procedure must be careful to annotate signals with their SENDER which is implicit in SDL anyway.

In situations where we have process sets, the practitioners' induction can be applied if the processes are independent. We may consider each communication with a unique external PId as an external channel.

4.3.7 Integrating the Mn-approach with other methods

The Mn-approach is friendly towards other methods. This means that since the Mn-procedure delivers an SDL process description from an SDL block description, the nature of the system does not change when having applied the Mn-approach. The system is still an SDL system if we consider our suggested notational extensions as parts of SDL.

This means that any other method which is able to handle SDL, can also handle the reduced system. The question is whether the analysis must be restricted when including a reduction rather than the original.

We claim that since the reduction is observationally equivalent to the original, any analysis which does not address properties internal to the reduced sub-system may equally well work on the modified system where the reduction has substituted the original subsystem.

We should make the reader aware that this is not the same that every property expressible in the terms of the reduced is faithfully preserved from the reduction to the original. A simple example is depicted in Figure 108 (p. 163). There is no doubt that the process X is reducible and that the reduction will also include the declared variable ni. The reduction is simply a process which consumes e and produces z, but does not change ni at all. Thus it is simple to see that ni=0 always. If we checked for the LTL formula saying that ni is always 0, we would get that it is valid for the reduction. For the original, however, it is not difficult to see that there are states where ni is not 0.

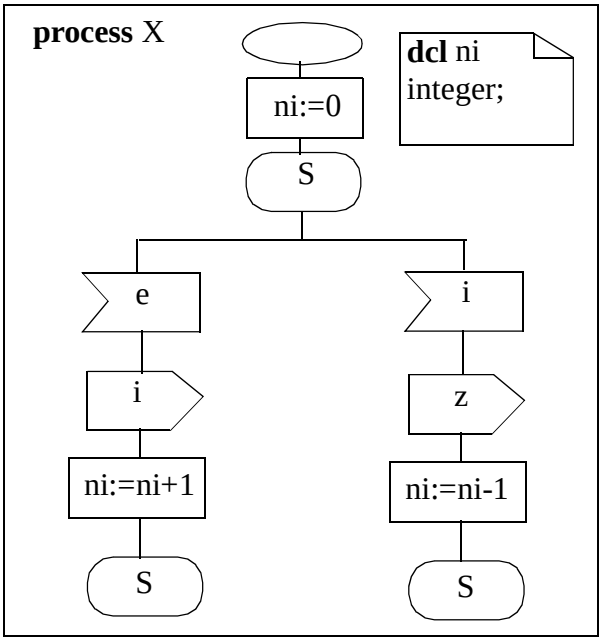


Figure 108: Original process

If we restrict ourselves either to only stable states in the original or to indicators which are external to the reduced process, the original is faithful to the reduction.

Since some of the most popular methods to verify SDL systems have serious problems to keep the state explosion under control, the Mn-approach should be a valuable contribution to reducing the state space without losing the interesting aspects.

4.4 The expected behavior of the Mn-procedure

Here we should summarize how the Mn-procedure behaves with different kinds of systems.

4.4.1 Studying Progress

Progress enters into our Mn-approach in two ways:

1. The Mn-procedure assumes (weak) progress.
2. The Mn-procedure itself should preferably terminate.

4.4.1.1 Progress of the system itself

The progress of the system itself was studied in Section 2.3 (p. 50), and we emphasize that this thesis is not about termination or progress. Still for our Mn-procedure the established progress of the process under analysis is important as we showed by the example in Section 2.4.3.2 (p. 53). It is possible to find processes which the Mn-procedure will evaluate to be confluent, but which contains a livelock. Thus the reduction is not a proper reduction of the process as it may fail to recognize the livelock.

Just as many formal methods we may resort to “partial correctness” [30] which has as an assumption that the program (or program statement) covered terminates normally. When progress is established, we talk about “total correctness”. In practice, however, there is a big difference between a system which is in a livelock and a system which reports an internal error. We shall cover these aspects closer in Section 5. (p. 177).

Since communicating finite state machines can be seen as rewrite systems, and progress in our terms corresponds closely to termination of rewrite systems, we should seek advice in the strong literature of rewrite systems to find the most appropriate means to establish progress [37]. Our “signal ordering criterion” is in this tradition. We find a well-ordering of the signal types (which corresponds to elements of the symbol universe of rewrite systems) such that every transition produces signals of less value than they consume. The implicit ordering thus becomes a reduction ordering which implies that the system terminates.

To find such an signal ordering is simple and automatic if it exists. If there is no such ordering, there are cycles in the directed graph that represents the attempted ordering. Such loops may or may not imply that there is a livelock. If the loop includes external signals as well as internal ones, we may have finite progress (Section 3.1.3 (p. 86)), but not infinite progress. Finite progress is what is needed for the Mn-procedure. Finite progress is achieved when the internal signals can be partially ordered.

The pure signal ordering criterion does not consider the basic state. By also considering the basic state in the production of a reduction ordering for the process, more cases may be covered.

Our approach to resolve loops constructively is to let fair decisions with helpful escapes from the loops, or to let a timer do it as time itself can be certain to progress.

4.4.1.2 Progress of the Mn-procedure

Termination of the Mn-procedure was studied in Section 2.4.7 (p. 69). There are two different situations where the Mn-procedure does not terminate. Either it loops during the execution within one generation, or the sequence of generations is infinite. In practice a pragmatic constant limit to the length of an execution path within one generation and a limit to the number of generations will be sufficient to terminate the Mn-procedure such that few interesting confluent systems are considered impossible to prove confluent by the Mn-procedure.

Through theory we should be able to give some indication to what these two numbers should be.

Execution path

Our proof of the correctness of the Mn-procedure (Section 2.4.6 (p. 65)) is not dependent upon when a generation change takes place. We say that if a node of the Mn transition system is evaluated to sequence permuted, there must be a generation change sooner or later. This is not completely true since the criterion “external stuttering” modifies this. With our example process D, we use exactly this in Table 4 (p. 64) where state 12-1 is sequence permuted, but we continue the execution in this generation and find external stuttering already in state 12-1-1.

The disadvantage of changing generation is that we have to continue on the next generation from all conceivable basic states reachable from the origin of the generation change. This will often mean that we must include situation which cannot occur in prac-

tice. The higher the generation the more unreachable situations must be covered. The more unreachable situations which must be found confluent, the bigger chance that a problematic, but fictitious situation will prevent the Mn-procedure from concluding confluence.

On the other hand, another level within the same generation will also produce more situations which have to be confluent. The execution tree within one level of Mn is like other execution trees – it grows exponentially. Going from one node onto the next level will produce exactly the same number of new nodes as there are symbols in the A_n alphabet. But they may be more pleasant to handle by external stuttering or generation change.

It is also possible to backtrack within the Mn-procedure. If the generation change turns into non-confluence situations, a second try may execute one level more on the former generation.

In the end a constant limit to the number of execution levels within one generation will ensure that the Mn-procedure will continue and the game is not lost.

Sequence of generations

If the signal ordering criterion holds, the number of generations is limited, too. This follows from the fact that the number of signal types present in A_n is less than the number in A_{n-1}. Since there is only a finite set of symbols in A_0, we cannot have more generations than the number of symbols in A_0.

If there is a loop, this occurs normally in a system through a cycle of channels and processes. In a piecewise execution of Mn-procedure (see Section 3.3.4 (p. 91)) there will be one new generation per process in the loop. If no decision (either way) has been reached when the sequence of generations “returns” to where M0 started, the question is whether the alphabet A_n is smaller than A_0. If it is not smaller, nothing can be gained from continuing changing generations in a piecewise execution.

The picture is more complicated if the feedback patterns are more complicated than a simple feedback loop. Still we advice to compare alphabets when the process involved in M0 is again involved in a higher generation Mn.

4.4.2 Studying Confluence and the Complexity of the Mn procedure

Here we present some thoughts on complexity of the procedure relative to what we may expect.

Our aim is to determine confluence of a process which normally consists of a number of interacting components which themselves are such systems of components.

Our approach is a practical one where we do not try to cover the worst case, but rather an interesting set of common cases. There is no doubt that the Mn-procedure behavior is very much dependent upon the architecture of the system under analysis.

In this section we shall look at some of the factors which influence the behavior and needs of the Mn-procedure. We shall not go into detail about the complexity of the algorithm in any mathematical way. It is not difficult to construct systems which are non-confluent. It is not difficult to construct cases where the Mn-procedure loops (as we have

seen in Section 2.4.7 (p. 69)). It is not difficult to construct system architectures which makes the actual execution of the Mn-procedure complicated. Such architectures are typically when all components communicate with all other components.

We have, however, shown in this thesis that there are interesting theoretical and practical cases which benefit from the Mn-approach, and we have provided arguments for why it is reasonable that the Mn-approach will work in many cases.

4.4.2.1 The obvious challenger

Could we determine reducibility by other techniques than the Mn-procedure e.g. by execution from the initial state supplemented by some termination criterion?

To eliminate the internal signalling of a system of communicating finite state machines appears to a newcomer to be a manageable problem. The obvious first thought is that merely executing the system from the initial state until some (simple) criterion says that there is no reason to execute beyond this point because “nothing new” will be found. In other words after having produced a finite (and hopefully small) execution tree, it is possible to infer that confluence in this tree implies confluence in the whole, infinite execution tree.

Unfortunately there is reason to believe that the obvious challenger is not a good choice. We list a few reasons:

1. The expressiveness of communicating finite state machines is underestimated. It is far more expressive than meets the eye. Actually a system of communicating finite state machines with infinite buffers has the power of a Turing machine [13].
2. It is reasonable to believe that a (simple) cut-off criterion cannot be found in general. Related unsoluble problems which also seem simpler than they are include to determine whether a given complete state is reachable [48].
3. To execute from the initial state has the advantage of considering only reachable states, but this is outweighed by the fact that such execution may go through similar cases numerous times and fail to reach the problematic ones in proper time.

Our idea is that “nothing new” must mean that all reachable non-confluence patterns have been checked. The plain execution strategy cannot give a limit to when all non-confluence patterns which are reachable have been reached.

4. Even if they could guarantee when all reachable non-confluence patterns have been found, it is probable that the time to reach it is prohibitively large.

It is probable that the criterion would have to check that a reached complete state is sufficiently similar to one already encountered. It is difficult to specify what “sufficiently similar” should mean, but equal state and some strong similarity between the signal sequences seem reasonable. If the cycles produce internal signals, we have to consider their consumption. In general this leads to a cycle of cycles. Futhermore these second generation cycles may themselves produce internal signals etc. Our conclusion must be that the complexity of the worst case is formidable.

4.4.2.2 The factors of Mn complexity

In this section we shall go through some of the factors concerning the complexity of the execution of the Mn-procedure.

Generation structure

The execution of the Mn-procedure is a tree of executions of transition systems Mn on different generations as shown in Figure 109 (p. 167). There is no requirement that the

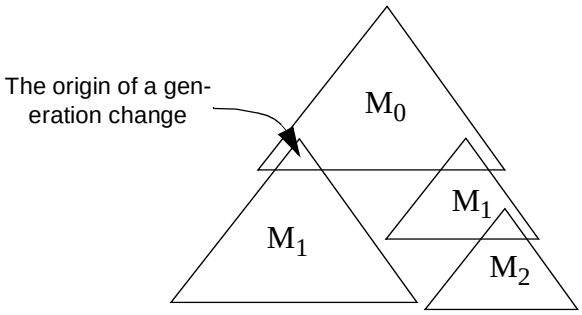


Figure 109: Tree of Mn-executions

tree is balanced in any way. In practice the opposite is the case. Some parts will be problematic and the tree complicated, while other parts will be trivial and the Mn tree likewise. Since the generation structure is a tree, the total number of nodes increase exponentially with the number of generations.

One Mn-execution

One single Mn execution is illustrated in Figure 110 (p. 167). The origin is one node of

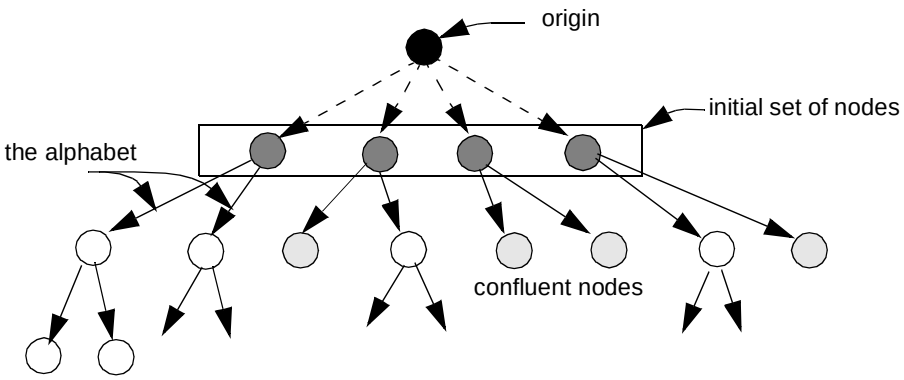


Figure 110: One Mn-execution

the generation above. In M0 this is implicitly the initial basic state. From the origin, the initial set of nodes has a basic state element and a signal element. The basic state element is found by determining the set of basic states reachable from the origin through the execution of the former generation. In M0 we use the set of basic states as we believe it is reasonable that they are all reachable from the initial basic state. The signal element is copied from the internal signals of the origin. In M0 we use all possible pairs of one signal from one channel and another from another channel. External signals are not paired. See the definition of Mn in Figure 37 (p. 60). The alphabet is dependent on the output of the former generation. In M0 the alphabet is the pairs of equal internal signals.

The growth of the tree is exponential and from every node where a new level is needed the number of nodes on the next level is equal to the number of elements in the alphabet. But the total size of the Mn-execution tree is very dependent upon the fact that the tree is not balanced.

Worst case?

From the reasoning above, the worst case execution seems to be double exponential: first a tree of nodes in one Mn-execution and then a tree of Mn-executions. Now it is time to remind ourselves that the changing of generation is *instead of* going another level down in the current Mn-execution. The fact is that the size of the tree is comparable irre-

spective of the choice between changing generations or taking a new level of the same generation. To explain this, let us look more closely at what may happen in one node of an Mn-execution.

If the evaluation is that another level of the same generation should appear, the next nodes on this branch to be evaluated amounts to exactly one for each symbol of the alphabet. If the evaluation indicates a generation change, the next nodes on this branch to be evaluated are the set of initial nodes on the next generation. The number is smaller than the set of basic states.

If we assume that we may perform our execution piecewise (see below), every Mn-execution takes place within one process. The set of basic states of a finite state machine is often smaller than the set of signals, but in general we can only say that they are approximately the same regarding complexity.

Further execution in the new generation will use an alphabet which is normally smaller than the alphabet in the former generation. Thus to change generation will normally slightly decrease the complexity of the execution since the number of states is probably less than the number of alphabet symbols of the former alphabet, and the alphabet of the new generation is smaller than the former. The order of complexity is the same.

From this we conclude that the overall execution of the Mn-procedure is comparable in complexity with a plain M0 execution, which is comparable with a plain execution of the system from the initial state.

But execution from the initial state runs into state explosion problem very rapidly. Why is the Mn-procedure considerably better? And the answer is twofold. Firstly our execution is directly targeted towards the solution of our confluence problem. Secondly we are able to perform the execution piecewise in most cases.

Confluence target

Exactly as any reachability strategy may find what it looks for very rapidly, our Mn-procedure may find confluence along a branch almost at once. In practice the complicated cases are very few compared to the number of potential non-confluence patterns.

Piecewise execution

Piecewise execution (Section 3.3.4 (p. 91)) means that we may execute the Mn-procedure within one process at the time. This reduces the state universe from a cartesian product to a plain set for every Mn-execution. In M0 it is obvious that piecewise execution is possible, but also with higher generations it is highly probable, but it is dependent upon the architecture of the system.

In Figure 111 (p. 169) we show which processes are involved in which generations. The potential non-confluence pattern is in the leftmost process shown grey in M0. If a sequence permuted node is encountered the next generation, which should resolve the permutations of the output channel, takes place in the grey process of M1. The same happens again in M2, while in M3 two new interesting things happen.

1. Two processes are connected to the outputting process of M2. If there are sequence permutations on both of these, both connected processes must be considered on the next generation, but they may be considered separately.

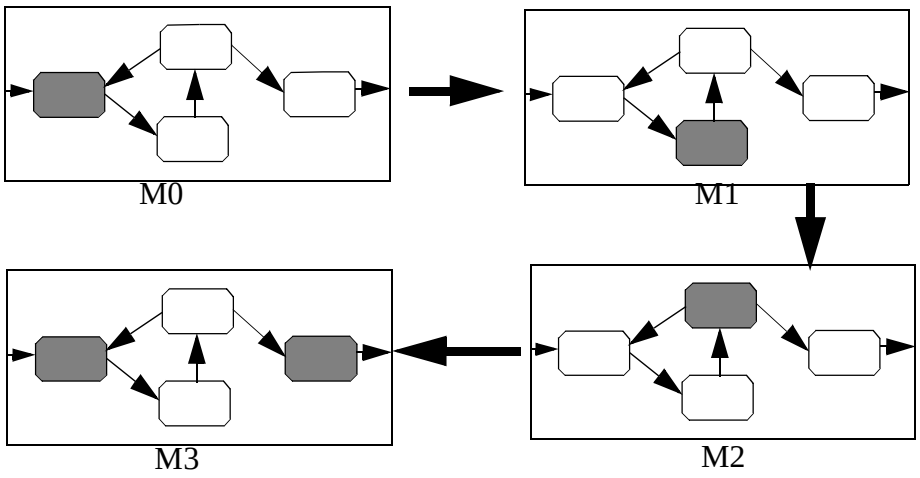


Figure 111: Piecewise execution of Mn

2. The original M0 process is involved again, now on M3. This is the time to check that the alphabet has actually been smaller. Otherwise chances are that more generation changes will not conclude the case. In M3 the initial state space of the leftmost process is only the space reachable from the state where it all started in M0.

Conclusively piecewise execution of Mn makes the Mn-procedure approximately linear wrt. the number of components of a system and the complexity is greatly dependent upon the architecture of the system. The simpler the architecture, the simpler the Mn-procedure execution.

Thus piecewise execution means that the Mn-procedure scales reasonably well, while the opposite situation occurs for reachability analysis where the size of the system in terms of independent components is extremely important.

Nested execution

Piecewise execution refers to the fact that components of the system which are on the same aggregation level [56] can be handled one at a time. Nested execution refers to the property that components on different aggregation levels can be handled separately. This is obviously another important advantage of the Mn-procedure and a property which is at the heart of the aim of the procedure.

The compositionality principle (Section 4.1 (p. 143)) combined with object orientation in SDL-92 (Section 3.9 (p. 133)) make the Mn-procedure even more attractive from a complexity point of view. As long as the systems are composed of reducible components, large systems can eventually be analyzed with the Mn-procedure.

Stabilization

We have in the arguments above concentrated on the number of nodes. The Mn-procedure also specifies that every node needs to be stabilized. This means a plain execution sequence since the order in which eligible signals are executed does not matter. We must, however, take into account that stabilization may involve several of the interacting processes. Thus the stabilization is actually a tree structure which branches on the basic states of the processes activated by the state to be stabilized. See also Section 3.3.4 (p. 91).

The number of internal signals in a node is normally very small since we try explicitly to try out only one internal signal at the time. Furthermore we do not always need to find the sets of leaves explicitly. Whenever the two elements of the node are equal, we can cut off the stabilization. Therefore stabilization is not very time or resource consuming, but it adds a linear factor to each node.

Heuristics

The evaluation of nodes plays an important role in the Mn-procedure. It may be decisive whether a generation change is performed or not for the success of the procedure. Backtracking means loss of time. Therefore the evaluation of the node, the potential alphabets and state spaces could probably be made into smart heuristics. Since we would expect the systems to be analyzed over and over again (after small modifications), such heuristics could also take special aspects of the system into account.

Such heuristic on the evaluation of a node, can be used as a measurement of complexity which is similar to the complexity profile in Section 5.2.2.2 (p. 197).

Time and storage

The Mn-procedure is basically a depth first algorithm. Since we prefer to conclude confluence, there is no gain to go breadth first unless we suspect that we will not find confluence. Depth first requires very little storage. Apart from the system representation itself only the stack of nodes from the initial state is needed with the associated information about every transition system (Mn) which is invoked on the stack. So the minimal storage requirements are almost none.

The time consumption of the Mn-procedure is proportional to the number of nodes generated. To minimize the number of nodes generated, it is possible to include data structures such that nodes which are sufficiently similar to earlier and analyzed nodes can be recognized. The nodes of the Mn-procedure requires more information than plain execution since in principle symbolic execution is performed. In theory of course the bit-state hashing technique known from [73] could be applied if we settle for less than absolute certainty of match. This is most probably not a good strategy with the Mn-procedure since the number of nodes within one process should still be less than the number which is manageable in large computers.

Non-determinism

Non-determinism (including timers) in the system under analysis will make matters worse in all respects. Non-determinism increases the number of simple nodes, and complicates the evaluation. Data is in this respect equal to non-determinism because it involves more alternatives (with guards). The guards may be resolved manually. Any manual intervention will of course have detrimental effect on the overall performance.

Total Mn-procedure depth

The complexity in terms of time and possibly space (when optimizing time) is very much due to the expected number of levels in the tree. There are two different approaches to this issue:

1. What are the chances of succeeding to find confluence when the number of levels increases?
2. Which structural elements normally set the limit to progress of the Mn-procedure?

The first approach is based on the assumption that the probability of succeeding decline with the number of levels. This is probably true simply because more nodes must be confluent. Changing generation also means that the system must have even greater degree of orthogonality. Even more unreachable situations must be covered by the confluence.

But there is also a psychological dimension. Assuming that the designer has intended the system to be confluent, the more levels the more complicated knowledge must have been behind managing to make the system confluent. Unless the designer has guided his specification by some well sort out invariants, the designer will normally be able to overview only a very small number of levels of sequence permutation. More about this in Section 5.2.2 (p. 193).

The second approach searches for the theoretical limits. There are two types of cycles which indicates that the limit has been reached.

1. During one Mn-execution, if a basic state which is in a node on the execution stack reappears in the current node;
2. When changing generations, if the component to execute this generation is the same as a process which has executed earlier generations of the same potential non-confluence pattern (c.f. Figure 111 (p. 169)).

Neither of these indicators are certain indicators of failure. They do indicate, however, that the Mn-procedure may be entering a loop.

4.5 Conditional reduction

It is not always the case that the whole system can be proven reducible through simple automatic techniques. There may exist questions which cannot be solved by the Mn procedure itself. These are “proof obligations” and the reduction is “modulo” these proof obligations. Typically we have proof obligations concerning:

- progress,
- unreachable non-confluence patterns,
- impossible transitions,
- resource restrictions.

4.5.1 Possible attitudes to proof obligations

The Mn-approach has three possible attitudes towards these proof obligations:

1. Prove the obligations by ad hoc techniques,
2. Assume the obligations valid,
3. Check for the obligations during runtime.

It is also possible that a combination of these attitudes represents the best alternative.

To prove the obligations valid, is definitely the most desirable attitude, but it may not be practically feasible. It may be more feasible for small (but interesting) systems such as our process D (Section 2.2.3 (p. 49)), the Alternating Bit Protocol (Section 3.5.3.1 (p. 100)), the Brock-Ackerman anomaly (Section 3.5.4.2 (p. 109)) or the Dagstuhl example (Section 6. (p. 229)).

To assume the obligations valid, is definitely the least desirable attitude. It may be advisory to assume progress without having proven it such that the problems of the confluence search (Mn-procedure) appear. This attitude may also be characterized as "postponing" the proofs until the desired reducibility has been established. Often problems will arise more than one place such that getting preoccupied with one problem may not be cost effective. Still we do not advise to leave too many assumptions unproven at the end of an analysis.

To check for the problems at run-time is the practitioner's attitude. If you cannot prove something, check for it. Even if you *can* prove something, check for it! Checking or exception handling divides the execution of the system in two layers: the system execution layer and the monitoring layer (see also Section 5.3.5.2 (p. 207)). In practical system engineering this is a very attractive approach. The verification results become "modulo" the monitoring. Said differently, the reducibility is dependent upon the system not escaping to the monitoring layer. Either the system behaves according to the reduction or it calls an exception and enters the monitoring layer. The verification is conditional.

Since the conditional verification is not the most desirable, we may want to try and combine the monitoring attitude with a recovery strategy. When an erroneous situation is detected, a warning is issued to the monitoring layer, but the system execution is instructed to try and recover as best as it can. As we pointed out in Section 4.4.2 (p. 165), the save-approach to internal errors may well constitute a good recovery. The signal should not have been here at all, the best thing to do is probably to save it to a state where we have something sensible to do with it. In principle any recovery action could be suitable, but it has to be tuned to the problem at hand.

4.5.2 How do we typically check for the proof obligations?

Progress

Progress is not simple to check for. We may introduce timers at the monitoring layer. The reason for not including them in the system execution layer may be that this would complicate the system more than we want. The monitoring layer may also have access to mechanisms which are beyond the system execution layer such as the supporting run-time system.

Unreachable states

Unreachable states cannot in general be proven unreachable. In particular cases, however, it is possible to do it. To check for reachability is sometimes possible. All complete states can be checked for each component process. One problem with this is basically the overhead in execution time which originates from checking every state reached. Another problem is that the state which we want to check unreachable is divided between independent components. The projection onto each component may be easily reachable, but the combination is supposedly unreachable.

Impossible transitions

Impossible transitions are simpler to test for. They require no extra overhead as the test is not invoked unless the transition which should be impossible is about to happen. The reason for considering the transition impossible can usually be found locally. Explicit or implicit invariants of the basic state accounts for this.

When the Mn-procedure encounters an impossible transition (exception call) during the confluence search, this branch of the search is concluded with success. This means that non-confluence cannot be concluded from this branch.

4.5.3 The impossible transitions

SDL is a language which by definition defines a total finite state machine. It is total in the respect that for every basic state, any signal in the input signal set (alphabet) is accepted. If the transition is not explicitly specified, a “default transition” is used which means merely to consume the input signal. But, there are many transitions which the designer knows are impossible, or equivalently: constitutes an internal error. To examine progress and confluence of impossible or erroneous transitions seems a waste of time. We have the following choices:

1. to accept the default transitions,
2. to define a special exception construction for the internal errors which e.g. writes an error message and terminates,
3. to consider the impossible transitions saves instead of consumptions,
4. to prove that the impossible transitions are really impossible and skip their analysis.

Default transition

To accept the default transition is normally the worst alternative since the default transition is a very rudimentary recover action. We would expect it to produce non-confluence patterns.

Exceptions

To define a special exception construction is often the approach in practice (Section 5.3.5 (p. 206)), but exception constructs have also been seriously discussed within the ITU standardization body [112]. If this attitude is adopted then any execution of such a transition during Mn-procedure should lead to the termination of that branch with success. This means that the Mn-procedure recognizes that the exception mechanism takes care of the case and for the Mn-approach this means that the case can be considered impossible and safely discarded.

Save

To consider the impossible transitions **save**s is often a better approach than consumption. The signal is not consumed, but left to some state where it is welcome, and where its consumption is explicitly specified. Within the standard SDL this is probably the best solution. A mixture of the two latter alternatives could be to give an error message to some console, and use the save as the recovery.

Proofs

To prove that a transition is impossible, means to prove that the state before it cannot have that signal first in the input port. This is not so simple. Backwards execution is made difficult by the fact that the tail of the input port is not known and this is what a backward execution wants to know! We experience that the backward execution gets into infinite loops from unreachable state to unreachable state. There is a need to detect such loops and provide inductive arguments which is not so easily automated. Normally a better strategy is to find a proper invariant which can be shown to hold for every transition. The problem with this is also that it is not so easily automated.

Example: Alternating Bit Protocol

Take the Sender of the Alternating Bit Protocol (Figure 57 (p. 101)) and try and prove that the internal signals B0 or B1 cannot appear in state Send0. We shall see how our four different approaches to the consumption of B0 and B1 in Send0 turn out.

1. B0 *and* B1 *are merely consumed in* Send0. Surprisingly enough it turns out that the ABP system appears to be confluent still, but showing it is quite involved and far too voluminous to be presented here. The complexity of the confluence proof reflects our surprise that it is actually the case. This strategy is even more interesting in the variant of the Alternating Bit Protocol which uses timers in Figure 81 (p. 123). This means that the ABPT could handle situations where the timer occasionally expires too early.
2. *The* B0 *and* B1 *transitions in* Send0 *are considered internal errors and an exception will be invoked*. This means that the confluence is relative to the internal error. We do not prove that the signals cannot occur in Send0, but we are certain to catch them. (See Section 4.5 (p. 171))
3. *We save* B0 *and* B1 *in* Send0. It is trivial to prove confluence since Sender cannot have any non-confluence patterns since either the signals of the external channel are saved (in Wait0 and Wait1) or the signals of the internal channel are saved (in Send0 and Send1). Performing the reduction will result in the system shown in Figure 61 (p. 105) and this reduction shows that there will never be B0 or B1 in Send0 because if there was, we would have a reachable semi-stable state including B0s or B1s since only external signals exit from Send0. Any reachable semi-stable state would have to show up in the reduction as pointed out in Section 3.4.4 (p. 96).
4. *We try and prove that transitions* (Send0,B0) *and* (Send0,B1) *are impossible*. To prove that the transitions of consuming B0 and B1 in Send0 are impossible, we can prove that the signals B0 and B1 cannot appear in Send0. This is done through an invariant. In this case shown below the proof is not so difficult, but finding the invariant is not automatic.

Impossible transition proof through invariant

Alternating Bit Protocol example invariant:

1. When Sender is in Send0 or Send1 there are no more internal signals in the whole system.
2. When Sender is in Wait0 or Wait1 there is exactly *one* internal signal in the system.

This is the invariant which should be considered for every transition in the system.

Initially:

1. Initially it holds trivially since the start transitions of Sender and Receiver do not produce any signals and Send0 is entered by Sender.

Sender:

2. (Send0,e) Assuming 0 internal signals before transition, it produces exactly one, leaving 1 internal signal.
3. (Send0,B0) Not applicable since we are assuming 0 internal signals.
4. (Wait0,B1) Assuming 1 internal signal before transition, it consumes one and produces one, leaving 1 internal signal.
5. (Wait0,B0) Here there are two cases, in both cases we assume 1 internal signal at start which is consumed:

5.1 Produces 1 internal signal (A0) and remains in Wait0 requiring 1 internal signal. This is OK.

5.2 Exits to Send1 without producing anything. that leaves 0 internal signals and this is OK for state Send1.

6. The Send1 and Wait1 transitions are symmetrical to the Send0 and Wait0 transitions and thus they preserve the invariant.

Receiver:

7. We know that Sender must be in Wait0 or Wait1 because otherwise there is no internal signal present for the Receiver to consume. All transitions of the Receiver consumes an internal signal and produces another. The invariant holds.

8. Thus we have proved that in Send0 and Send1 there are no internal signals and thus receiving B0 or B1 is definitely an error.

Conclusion Conclusively it seems that the easiest and strongest approach, at least in this case, is to describe the impossible transitions as saves. The reduction shows that the transitions are actually impossible. This holds because all the internal signals are saved in the state. The most practical solution is probably to combine the exception approach and the save-approach. If the designer really thinks that the transition is impossible, the implementation could issue a warning and then perform a save as recovery.

4.5.4 Bounded resources – Mn on a finite system

An implementation of an SDL system in real life has to have several limitations compared with the ideal world of SDL[11; 12]. In our context the limitations on the number of processes and signals are of major importance. The number of processes is covered in more detail in Section 4.3.6 (p. 160).

Assume that there is a fixed maximum number of signals for each channel. The number needs not be the same for all channels, but for each channel it is fixed. The monitoring system checks that the limit is kept and calls an exception if the number exceeds the limit. This means that there is a finite set of complete states. The queues of internal signals are no longer of any length. This implies furthermore that M_0 suffices in theory. The Mn-procedure collapses to an M0 execution where the execution is cut off whenever the initial internal queue has reached its limit. If we can keep the channel signal number limits small, this may be an attractive approach.

In a situation where the channels are bounded, and data can be handled symbolically, the system is finite state. This means that in principle confluence can be decided by exhaustive simulation. It depends on the chosen size of the channel limits and on the communication structure of the system whether this strategy is applicable. If exhaustive simulation seems too laborious, we may of course apply random simulation and achieve a certain statistical significance for our confluence conclusion.The big advantage with this approach is that there will be no problem with unreachable non-confluence patterns since only reachable states are examined.

5 The Mn-approach in practical engineering

Grook to Stimulate Gratitude (in sour rationalists)

As things so
very often are,
intelligence
won't get you far.

So be glad
you've got more sense
than you've got
intelligence.

5. The Mn-approach in practical engineering

The examples given earlier in the thesis are fairly small and theoretical systems. How can the Mn approach be adapted to more practical engineering environments?

Systems are made and modified by humans. This fact is often overlooked in connection with validation and verification. The issue should not be for the proof theorist to assert "correct" or "not correct" for a given system, but to assist in improving the system quality. Instead of applying immensely complicated verification techniques on a bad program, the program should be made simpler such that an automatic verification technique could be applied.

In this chapter we develop a reference model for the nature of real, reactive systems in Section 5.1 (p. 178) and we apply this to evaluate how the Mn-approach would work for real, reactive systems. summarized in Section 5.6 (p. 223) and to develop a method for system engineering which emphasizes the integrated use of the Mn-approach in Section 5.3 (p. 199) which we call "confluent design". In Section 5.2 (p. 192) we describe our expectations for applying the Mn-approach to real systems. We make a simple estimation model for the execution of the Mn-procedure and develop a concept of perceived complexity reduction. In Section 5.4 (p. 217) we report from a small case study of applying the Mn-approach to a part of a real system at Siemens AS. In Section 5.5 (p. 222) we discuss how Mn-tools should be built.

5.1 The Nature of Real Reactive Systems

Here we postulate what real systems are like, how they are made and how they are perceived. Our aim is to find out whether there is an interesting class of real systems which could be analyzed by the Mn-approach. We conclude that there is such a class of systems.

5.1.1 What is a real, reactive system?

A *real system* is being used, or will be used. A real system is implemented on some hardware and actually executed. A real system could be sold on the market.

Systems which are not real systems, but which still may be very interesting, are systems for education, systems for joy and play and non-implemented systems i.e. systems which have been specified, but not implemented. Typically such systems have been abstracted considerably (Section 4.3.3 (p. 157)) and a practitioner may not feel certain that the system reflects the problems of a corresponding real system.

A *reactive system* is a system where the immediate behavior is directly dependent upon the most recent stimuli received. The system reacts to the stimuli and outputs corresponding signals to the environment. The key characteristic is that a reactive system is preoccupied with behavior which is caused by the consumption of input signals.

That a system is reactive is also very much a matter of how we decide to perceive the system. The same system may be perceived as reactive when studying the signalling and the reactions, and as a (static) database system when focusing on data structures and the storage of data.

A personal computer system with its windows and mouse has definite reactive characteristics, but it may also be considered a data storage system, a mathematical modeling system, a typographic tool, or a video editing machine, depending on its use.

5.1.2 What is typical for real, reactive systems?

Having defined the notion of a real, reactive system in Section 5.1.1 (p. 178), we continue to characterize properties which seem to correlate in practice with real, reactive systems.

The Mn-approach is defined to cope with systems specified in SDL. SDL systems are reactive as they are based on finite state machines and signal interaction. Thus we could limit ourselves to finding typical characteristics of real SDL systems. As pointed out in Section 1.2.2 (p. 3), SDL has been used extensively in the telecom area (for which it was developed) and telecom systems are perhaps the most typical reactive systems for which the Mn-approach is useful.

5.1.2.1 Size

Telecom systems are big. Telephone switches used to be some of the largest pieces of software ever made. Now rumors are that the newest windowing systems are even bigger, but they are also reactive systems even though they are hardly specified in SDL.

The size of computer systems can be measured in a number of ways. The number of independent components is one metric, while the total number of lines is another. It is not very important for our purpose to go in great detail about what metric we use to measure size, because we want to emphasize that size is not synonymous with complexity.

Still the pure size of a computer system usually creates problems for automatic verification techniques.

5.1.2.2 Independent components

It is also typical for a reactive system that it contains a set of concurrent devices which are operated in true parallel. The personal computer has its mouse and its keyboard for input, together with telephone lines and computer network. Some machines may even have video and audio input and output. The interaction between these independent devices and the management of their cooperation give rise to many challenges.

Real, reactive systems are composed of independent, but interacting components. Even though they may be interacting such that the graph of channels forms a connected graph, this does not mean that every external input activates all the components. Normally only a small portion of the components are involved when an input is handled.

In a system of concurrent and independent components, the flow of signals, which actually determines the flow of control in reactive systems, is very important. While many approaches to system analysis focuses mainly on static (data) structures [36; 29], our Mn-approach concentrates on behavior.

5.1.2.3 Nesting

By “nesting” we mean that system structure concepts may have a recursive definition. In SDL blocks may contain blocks, and in the end a block contains processes. In StateCharts [54] states contain states. Thus the structure of a system becomes a tree structure (or a directed acyclic graph). Such structures are well suited for optimization of traversals. Compositional reasoning becomes very attractive [132].

Whether real reactive systems are nested structures depends on the principles used to describe them. SDL systems and systems described by StateCharts are often nested because the languages allows and encourage it, while systems described by OMT [118] and implemented directly in C++ [130] will be less nested because OMT and C++ is less oriented towards nesting.

5.1.2.4 Data

A system of communicating finite state machines have the power of a Turing machine [13; 23] which is sufficient to be able to describe systems where progress (termination) is not provable. Therefore from a theoretical point of view there is no need for data variables of the SDL processes to describe systems where the Mn-approach will meet problems.

For a practitioner, the data variables add flexibility and expressive power which make the descriptions more compact and easier to understand. But data variables will also more easily create specifications which are more difficult to handle by automatic verification means.

Real reactive systems will normally include a considerable amount of data variables, but only in few parts of the systems there will be complicated algorithms or complicated administration of data. In many cases the data-intensive parts of the reactive systems have been isolated either as subsystems or in operators. Subsystems may be described in data-intensive notations such as database languages while operators are often defined in programming languages such as C.

5.1.2.5 Heterogeneous

We cannot expect to use one kind of methodology for all parts. The combination of methods must be exploited.

We may find systems where data play fairly isolated and minor parts, but seldom systems where data variables are absolutely absent. We may find systems where the Mn-approach can apply to almost all of the system, but rarely a system where the Mn-approach is all you need.

5.1.2.6 Real Time

Real systems operate in real time. Transitions do have duration and even the mere consumption of signals (or even the save operation) takes time.

We call a system a real-time system when the actual duration of time or the actual points in time are significant. SDL only provides timers to cope with real time. Timers is an imperative attitude towards time. We would also like to have means to describe constraints on time or in general statements about time associated with the specification.

Real systems' descriptions have real-time constraints mainly in comments and informal auxiliary constructs.

5.1.3 How are real systems made?

Here we want to characterize how real systems are made related to how the Mn approach can be effectively used in system engineering.

In Section 5.1.2 (p. 178) we characterized real, reactive systems from a static point of view. We characterized systems as they *are* and not how they *develop*. We characterized the way they *functioned* and not how they were *described*. In this section we shall describe the dynamic, development dimension and in Section 5.1.4 (p. 185) we shall evaluate the representation dimension [90] (how systems are described relative to how they actually appear).

5.1.3.1 System analysis – the use of different descriptions

In this section we shall point out a few properties which are related to the early phases of a system development and to the fact that a system is described in many different ways. In Section 5.1.4 (p. 185) we shall go into how different description forms are related to how well they are understood.

System development methods which are based on formal methods, like FOCUS [20] and VDM [87] tend to hypothesize that it is possible to develop systems by describing an abstract system first, verify this as far as possible, and then refine this description in small steps into a perfect implementation. This is a very naïve understanding of how sys-

tem development is performed. Even if they in footnotes agree that this one-way street paradigm is hardly likely to succeed and that “iterations” are needed, the impression left is still that every “iteration” or modification is something which is due to imperfection of the actual system development.

Our basic reference model is different. The earliest phases is not characterized by abstract and formal specifications. The earliest descriptions are vague and without much detail, but they contain important concepts and give indications of what the system should look like in the end. In our SISU Integrated Methodology (TIMe) [12; 58] we recognize the dialectics of practical refinement. From the initial descriptions there is both a need for more precise descriptions and a need for more detailed descriptions (Figure 112 (p. 181)). These two needs require different means, and they are definitely different, but they are interdependent. Still it helps to keep them apart as it clarifies why the pure top-down approach is doomed to fail, and that its modification cannot be understood as “iterations”.

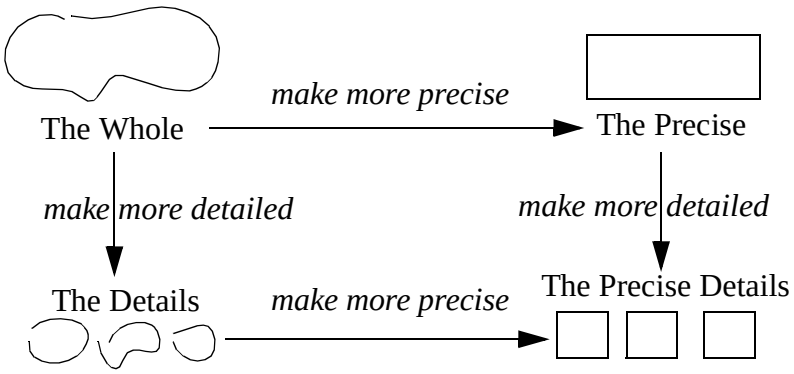


Figure 112: The Whole, The Precise and The Details

Make more precise

To make more precise descriptions can be divided in three subtasks: to formalize, to narrow and to supplement.

To formalize means to transform the description from an informal one to a formal one. This is non-trivial in itself. It involves the definition of concepts and work on interpretation of phrases.

During formalization it becomes clear that the original informal description was wider than intended since it implicitly supposed a “friendly” interpretation. Since formal notations normally do not have the necessary informal interpretation context, it is necessary to supply the description with definitions which narrow the interpretation possibilities. It is also common that formalization discloses that there are interpretations which nobody had thought existed. Narrowing cuts away those interpretations which are incompatible with the overall purpose of the description.

Formalization and narrowing may also discover that there are “holes” in the description. Aspects of the system has not been covered by the informal description. It is possible to label this “underspecification” by saying that uncovered aspects means that all interpretations of this aspect are within the description and what we need is narrowing. This is, however, not the way it is perceived by the developer who is trying to make the descrip-

tion more precise. He distinguishes easily between concepts which are present, but too wide, and concepts which are needed, but non-existent. The non-existent concepts must be supplemented.

It is conceivable to perform this branch of the development ("make more precise") without performing the other branch ("make more detailed"), but it may not be optimal. If the precision branch has been carried out, the result is a formal description on the same level of detail that the original informal description had. Then we are at the point where the formal methodologies want to start.

Make more detailed

To make more detailed can also be divided in three subtasks: to decompose, to break down and to reveal.

To decompose means to find which structural components the system is comprised of. This is in itself not a trivial task. It is not obvious what structural parts a system is comprised of, one cannot merely look at the system and see it. The components are defined through the purpose of the whole system and by the way the system is described. The decomposition principles of SDL and StateCharts lead to different components. Furthermore there may be several alternative decompositions within the same conceptual framework.

While trying to reach the definition of components, the breaking down of behavior patterns and communication may become an issue. Behavior and structure are dually related. A certain behavior may require a certain structure, and a certain structure may limit the behaviors possible. Still to divide the substance of the system is not the same as dividing the behavior. Decomposition is timeless, while breaking down involves timely behavior and sequencing of communication. The practitioner will very often perform decomposition and breaking down in parallel.

While decomposing and breaking down, structural and behavioral details are defined. This development can be compared with applying a magnifying glass even though the designer himself is making reality as he progresses. When we find more details, we also discover that new aspects become relevant. With higher granularity, some details were too small and insignificant to be included, while now they may be as significant as the details. Still these new aspects are not necessarily obvious "parts" of the already described systems. We say that we reveal underlying aspects.

As the precision branch could be performed without the detailing branch, symmetrically the opposite can also happen. Then we are left with a detailed informal description.

Distillery

Having performed both the precision and the detailing branches, we have a detailed formal description. Recognizing that the two branches may happen in parallel, that they influence each other and that performing one branch before the other may not be the most fruitful approach, we must have some means to assert that the precise and detailed result is a good enough result.

It is reasonable to believe that the precise and detailed description is not perfect. If the two development branches have been performed in parallel, the by-products "precise whole" and "detailed whole" may not be present. A distillery purifies the source. In this respect it means to work from the detailed and precise description and the original informal specification to form the precise whole and possibly the detailed whole.

Between the precise whole and the precise details there should be a formal refinement relation which should in principle be asserted. The informal description of the details can be used as comments to the formal ones, and as informal starting points of the making of the next level of abstraction. In Figure 113 (p. 183) we summarize the distillery

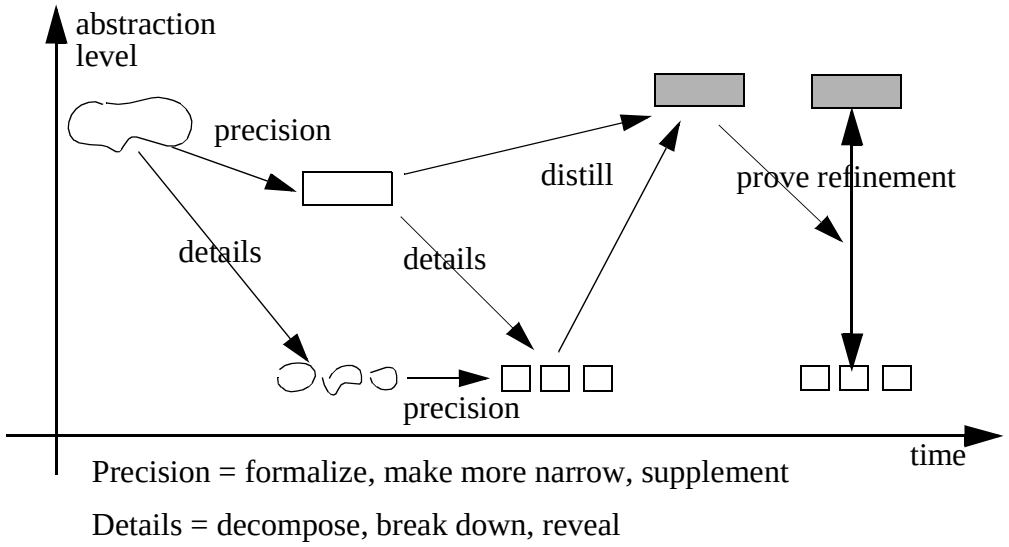


Figure 113: Distillery

development strategy.

Metrics of complexity and aesthetics of the description should also be used to make the descriptions proper for the next phase of description. In this final task, the description is again modified with the evaluation metrics and the need to make a proper refinement relation as inputs.

Iteration?

That the distillery approach represent a kind of iteration is not correct. Iteration means something which is repeated. Here we have not described repetition, but a set of tasks which interact and mutually influence each other. That the mutual influence also implies some loops is obvious, but it is not the main point.

The major difference in thinking is that our approach accepts as fruitful the intermediate descriptions which will not be maintained. Corrective measures are an integrated part of the development process.

5.1.3.2 System design – the dynamics of system development

The Main Description

A system description consists of a number of different documents which are made in different notations. This is recognized by all methods. Sometimes the multitude of notations become a nuisance for the designer and we find in practice that the different notations do not have the same weight. Very often there is one fairly complete notation which is the main notation, and the descriptions in this notation become the main documents. Traditionally the main description has been the program itself. Subsequent maintenance and corrections are made directly on the program and the design and specifications quickly become archaic and partly incorrect. Fortunately the advent of system

development techniques with code generation such as SDL-oriented methodologies [80; 44; 59] and OMT-oriented ones [118] have shifted the weight from implementation orientation to design orientation.

Continuous development

Real systems are never finished. They are maintained before they are delivered. Design and development of a piece of software is a very incremental process. Corrections and improvements are combined with new features. Changes take place all over the system to achieve a new purpose. This means that reachability of a certain system state is not very robust. Some work has been done to analyze the effects of many small changes (ripple effect) [139] such that the need for a total validation analysis can be avoided. The fact is, of course, that a total validation is not performed for every change request because validation efforts are often very time consuming and expensive.

Concurrent development

Design development is highly concurrent. Large systems can only be made by large organizations in the time frame available. Several parts of the total system are developed in parallel, but the different parts are not necessarily at the same level of maturity. Some parts turn out to be more complicated while other parts are simple.

Plans and reality

Projects are always late. Either they turn out to be late in real time, or they become late in working time which means that overtime and extra resources have to be applied. The predictability of development progress is a major concern of technology managers.

5.1.3.3 System validation – how to believe they work

When a system is going to be delivered, it is necessary to try and assert that the system works according to the expectations. We focus in this dissertation on verification as a means for asserting correctness in systems, but we also have realized that real systems are normally not verified in any formal sense of the word.

How is the system and its description validated? How do the developers reach the conclusion that the system and its descriptions are correct and appropriate?

- *User error reports*. The system is shipped once it runs at all. First it is shipped to in-house people, then to beta-sites and then to regular customers. They all report errors which they find when they are using the system.
- *Systematic testing*. The system is systematically tested before it is shipped. The testing can either be relative to a test specification, or it may be relative to the experience of the testers. In the first variant all different outcomes of a test have been given predetermined verdicts while in the latter variant the verdict is given in parallel with the test by the experienced tester.
- *Formal proofs*. The system is proved correct relative to a formal specification. Any transformation of the system description is based on formally defined transformation rules.
- *Walkthroughs*. The system is validated by human examination of the system description.

User error reports

Too often it seems that the in-house validation is not as thorough as the customer wants. The “hotter” the features of the system, the more anxious the companies are to ship products to a market, and the more tolerant the users are with system defects. The concept of “beta-sites” resembles the painting of a fence in “Tom Sawyer” by Charles Dickens.

Tom Sawyer needed to paint a fence, but by persuading his friends that painting was far from boring, he managed to make his friends do the penalty paint job for him. Beta-sites are likewise “friends” of the software company believing that it is very beneficial for them to perform extensive testing of an immature product.

Systematic Testing

Traditionally testing was the only means by which the company asserted the correctness and usefulness of their products. Testing will probably always be an important source of validation and the art of testing has become more scientific [52; 105; 24].

In real life testing is not done optimally. Even though serious companies may have very systematic ways to perform testing with all kinds of recording and special test teams etc., the truth is often that testing is limited to the time available in the end before delivery. When the time is up, companies may resort to the “user error reports” category.

Formal proofs

Very few systems have yet been formally verified, but their numbers increase. Especially in very technical problems, it is possible to state adequate and formal requirements which can be verified through automatic or semi-automatic means. WE refer to [10] for a collection of formal methods applications in industrial settings.

Walk-throughs

In recent years it has become more popular to try and perform some validation integrated with the development. In the rigid step-oriented development methods, such as VDM [87] and CleanRoom [39; 110] advocate the necessity to assert the step transitions through validation.

Since formal verification lacks tools and practical feasibility, the most cost-effective way to perform validation seem to be by walkthroughs [63]. There are a number of well structured ways to perform thorough analysis through the use of reading teams and peer designer scrutiny [50; 140; 51].

5.1.4 How are systems described and how are they understood?

In this section we present a framework for describing system descriptions and for understanding how descriptions are being understood. In Section 5.1.3 (p. 180) we concentrated on the synthesis of system descriptions and how they evolve, but here we look at the fact that systems are not made in isolation. Systems are developed in teams and there is a necessity that the team members understand what other development engineers are expressing. Just as it is important that a description expresses the intent of the designer, it is also important that the description is intelligible for others.

We shall go through a number of dimensions of descriptions which we find significant and give examples how languages, methodologies and system descriptions can be placed in this framework.

5.1.4.1 The language dimension

A good language is not good just because it is expressive wrt. a given domain, a good language is good for a number of purposes.

Syntactic form

How does the language describe the core of the problem? How is the syntax suitable for supporting the understanding of the core of the problem?

- Is the syntax *graphical*?

- Can the core ideas be *sketched* such that the remainder can be built upon it without much changes?
- How much is the understanding dependent upon *natural language* identifiers (or excessive commenting)?

Modern system analysis and development methods use a graphical syntax at least for the early phases. SDL[78] and MSC[86], StateCharts[54] and OOA[29], OMT[118] and UML[Rumbaugh, 1996 #272] are all very graphical. Graphics seems to improve the structural overview, but decreases the compactness such that a description becomes more easily overcrowded.

Whether the languages are “sketchy” is more difficult to assess. Some of the common analysis notations could be said to be merely sketchy as they cannot be code-generated to an executing system. SDL – on the other hand – is a language which performs reasonably well as a notation for sketches as well as it may develop the sketches into a consistent and complete SDL description.

The need for natural language identifiers and comments is dependent upon the topic as well as the language. Informal notations, and notations with very few basic building blocks normally need auxiliary information in the form of natural languages. More formal notations as SDL and StateCharts are not so dependent upon natural language supplements.

Very mathematical notation is again dependent upon commenting when it is used for system description.

Evolutionary aspects

The evolutionary aspects of a language is related to how changes in the descriptions are carried out. What impact does a change have in the description? Is there a reasonable correspondence between the perceived significance of a change and the amount of change needed? The evolutionary aspects of a language is also dependent upon the design of the system, but languages have different ways to cope with changes.

- *Pure modification*: the system is changed in a number of places. No trace of the former system can be found. The changes are not marked.
- *Specializing*: the evolution amounts to specialization of an existing concept.
- *Similarity*: the change is a new concept which is similar to an already existing one.
- *Parameterization*: the change depends on parameterization of an existing concept.
- *Granularity*: the change demands more detail to the description.

While some languages like SDL and common third generation programming languages like C++[41; 130] have strong structuring mechanism for both substance and behavior, other languages work best on small systems or with large pieces of paper. Object orientation is commonplace in system analysis today while the more formally inclined notations like VDM, Focus and Z[64] have put less emphasis on structuring mechanisms.

Topology

The topology of the description relates to the geometrical structure or the referential structure of the description. How would a reader browse around in the description?

- *Locality*: how much can be understood from looking locally, and how much is dependent upon changing view frequently?

- *History dependence*: how important for the understanding is the execution history?
- *Structure intuition*: does the description structure correspond to the substance and behavior structures?
- *Extrinsic relations*. how does the language express relations between objects on the same level, peer-to-peer relations?
- *Intrinsic relations*: how does the language cope with relations which are intrinsic to the understanding of systems, the MAGIC relations? (See below and [56])

Languages based on explicit states are fairly history independent as the current state represents the execution history. Such languages include SDL and StateCharts. Languages with independent and asynchronously communicating components such as SDL can be understood fairly locally while some object-oriented notations seem to motivate for descriptions where the execution context change very rapidly, and it is necessary to follow the slings and arrows of the execution. Auxiliary invariants are needed to facilitate local reasoning. The structure of an SDL system description is very similar to the structure of the running system. Again very dynamic descriptions in object-oriented languages may have the effect that the structure of the actual system only vaguely resembles the structure of the description. Complexities of the actual system may be well hidden within pointer structures that are hard to spot in the description. Also formal notations tend to hide complexities in aesthetically pleasing and compact formulas.

How are the peer-to-peer relations described? Are there explicit relations such as associations in UML[113], or pointers? If there are pointers, are they unqualified like in C, or qualified like in Simula?

The intrinsic MAGIC relations between processes are:

- *Meta-relation*: when one process modifies the description of another. This kind of relation is not normally found in programming languages, but it is found in LISP[137]. This kind of relation is becoming more interesting as modification of systems (reconfiguration etc.) must take place during continuous execution because all stops of the systems (such as telephone switches) cause great economic loss and security risks.
- *Aggregate relation*: which is the plain “consists-of” relation. It comes in many disguises and is present in some form or another in most system description languages.
- *Generation relation*: which in this context means that one process generates other processes. Object-oriented languages have this as one of the most important relations and mechanisms. In SDL processes may create other processes, but higher level constructs block cannot create other blocks.
- *Identity relation*: when processes are similar. Object orientation defines inheritance which is a variant of this relation. The identity relation also covers virtuality (polymorphism) and overloading of operators. To express similarity is very important to limit the description and validation efforts during maintenance.
- *Concepts*: to distinguish between singular processes, process sets and process types. Older languages like SDL-88[25] had problems with the distinction, and this can also be seen in more modern entity-relation oriented notations like OOA[29].

Semantic form

How a description is understood is dependent upon the appearance of the description and the expressiveness of the language, but it is also very much dependent upon the defined semantics of the language.

- *Declarative*: the language appears as a set of predicates which is supposed to be true.
- *Imperative*: the language appears as sequences of instructions to machines.
- *Mixed*: the language is an interleaving of declarations (invariants) and imperatives.
- *Formal semantics base*: the language is based on a formal semantics.

Declarative languages are typically the more formal notations (which then definitely have a formal semantics base), such as Z, VDM and Focus. Imperative languages are those which resembles programming languages such as SDL. Mixed languages are found also in the programming world such as Abel and Eiffel[101]. Also more pragmatic system description languages such as SDL and MSC may have a formal semantics bases[79; 98; 82; 109; 117; 68].

Communication

How does the language describe communication between processes?

- *Communication strategy*, is the communication asynchronous or synchronous or may both strategies be applied.
- *Communication means*, is communication performed through signals, shared variables or remote procedures?

Languages often choose a specific communication strategy with associated means. CSP[71] uses synchronous handshaking as was also adopted by Ada. CCS[103] also describes synchronous communication which seems to be the preferred model for formal notations. SDL and MSC describes asynchronous communication with signals, but SDL can also use remote procedures to simulate synchronous communication.

5.1.4.2 The user dimension

Users of the descriptions come in different categories. We must expect the users to have different competence and different interests and different inclination.

The programmer

The programmer focuses on *sequencing* and loops. He specifies *imperatively* the communication and the variable assignments. He has an imperative approach to time and timers as well.

The specifier

The specifier likes axioms and *invariants* to describe the situations in a system. He generalizes with quantifiers and uses symbols of foreign alphabets. Instead of loops, he understands repetition by *recursion*, and instead of sequencing he understands behavior as a *function* from input to output.

The team

The designer team is interested that *interfaces* are described properly. Furthermore the *independence* of the different components is important.

The observer

The observer wants to understand, more than to influence by creating. He is mainly interested in the *transparency* of the description. As a manager he may also be interested in asserting *progress* in the development of the description.

5.1.4.3 The problem dimension

How well is the problem understood before the design starts?

- *The technical problem*: the problem is well understood, and the work is mainly to formalize the problem and to implement it.
- *The explorative problem*: the problem is reasonably well understood, but there are aspects which need to be discussed and clarified.
- *The vague problem*: the problem is not well understood, there may be differences of opinion and conflicting interests. There is a definite need for improved insight.

5.1.4.4 Comprehension profiles

To give a picture of how comprehension is achieved we define a set of ideal comprehension profiles: the deceptive profile, the aha profile, the steady profile and the 90% syndrome profile. These profiles are intended to describe types of understanding development for an individual. A set of comprehension profiles make up a comprehension body. We want to use the comprehension profiles to formulate how understanding changes over time and to relate this to the language, user and problem dimensions.

The generic comprehension profile in Figure 114 (p. 189) shows that understanding may be perceived differently by the person than what is really the case. For the person the sum of the proper understanding and the misunderstanding makes up his perception of understanding since he has no way to assert his misunderstanding.

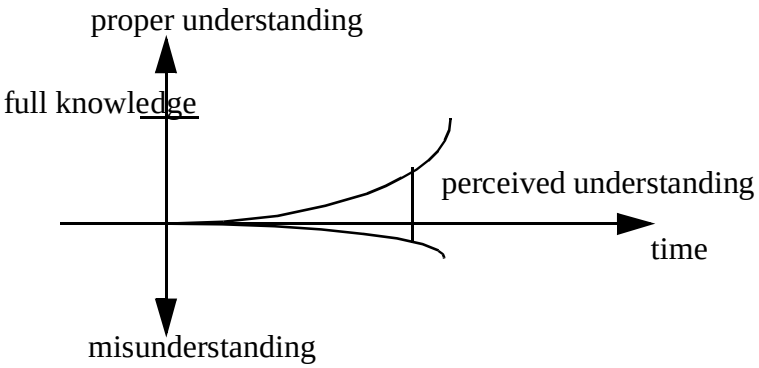


Figure 114: Generic Comprehension Profile

Deceptive profile

The *deceptive profile* (Figure 115 (p. 190)) is characterized by the fact that the person believes that he understands considerably more than what is actually the case. The effect of this may be that he acts with too much self confidence, or that the system gets to be delayed, or that it is eventually implemented in an improper way.

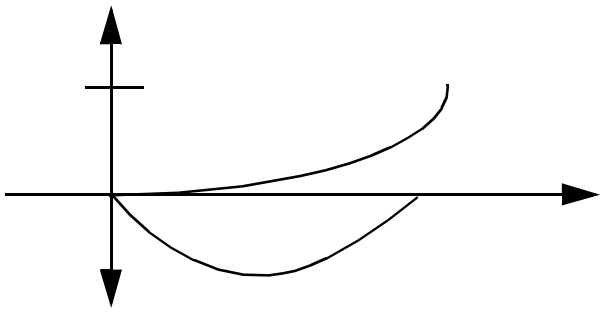


Figure 115: Deceptive profile

Aha profile

The aha-profile (Figure 116 (p. 190)) is characterized by a sharp rise in understanding at some unforeseen point in time (the aha-experience). The problem with the aha-profile

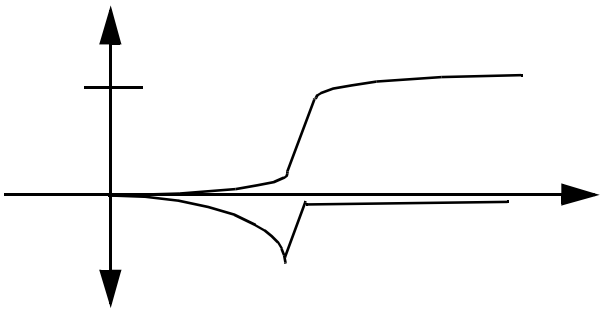


Figure 116: Aha-profile

is the unpredictability of the aha-experience. If a comprehension body consists of mainly aha-profiles, the participants will have very varied comprehension levels which will imply much overhead with discussion and conflict. Still the true aha-profile is positive as the aha-experience is very inspiring and encouraging for the remainder of the study effort. The more predictable the aha-experience is in time, the more the aha-profile resembles the steady profile (see below).

There is also sometimes a possibility of a “false aha-profile” where the person wrongly believes he has had the aha-experience, but in fact there is a sharp decline in understanding and massive misunderstanding. This is the worst case of mismatch between the perceived understanding and the actual one. If the false aha-profile exists in a team it will most surely exist in combination with true aha-profiles and the conflicts and discussions will be even more confusing.

Steady profile

The steady profile (Figure 117 (p. 191)) is characterized by a steady and predictable rise in understanding. The rise is not necessarily linear, but the clue is that the development is predictable from a fairly meager prediction base. The amount of misunderstanding is small and spurious. The steady profile gives rise to no surprises, but sometimes an increased understanding is hoped for in the project. The project leader of a project where the profile body is full of steady profiles may hope that he has a body of aha-profiles right before the burst of aha-experiences.

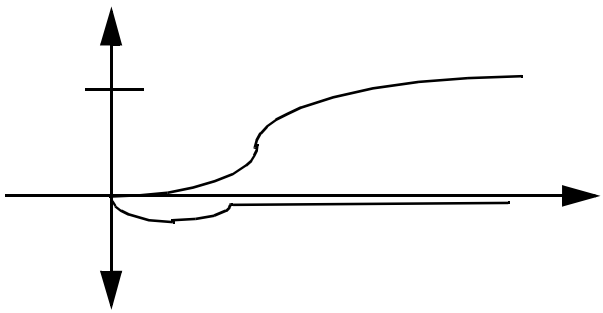


Figure 117: Steady profile

90% syndrome profile

The 90% syndrome profile (Figure 118 (p. 191)) is characterized with a quick rise to 90% of full understanding, but the latter important 10% understanding takes far more time than expected. The problem with the 90% syndrome profile is that it may be mis-

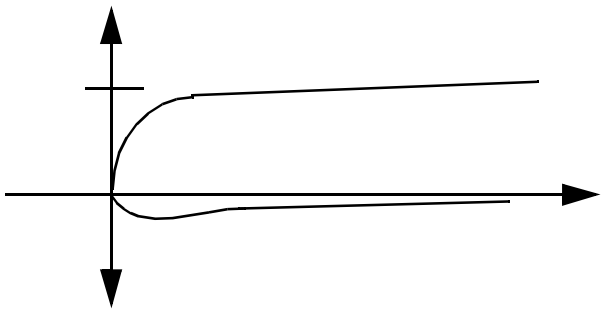


Figure 118: 90% syndrome profile

interpreted as the steady profile with a pleasantly short time frame. Not enough effort is put into covering the last 10%. A project with a body full of 90% syndrome profiles may spend an enormous amount of time not believing the last 10% are really there.

5.1.4.5 The system development dimensions and the comprehension profiles

The comprehension profiles are correlated with a number of other dimensions. Individual differences play an important role, but also our development dimensions language, user and problem are correlated with what comprehension profiles one could expect to find in a project.

Deceptive profiles are most frequent where there is a mismatch between the problem, the user and the language. A problem which is vague, a user which is an observer, should not be offered a language approach which focuses on declarative, formal and textual documents. Conversely a problem which is technical, a user which is a specifier should be using a language which has a formal semantics base and is capable describing the intricate aspects of the problem.

Aha-profiles are e.g. found in projects with premature use of formal techniques. Formal techniques are characterized with focus on languages with a formal semantics base, and where formal proofs are the major approach to validation. Formal techniques seem to require a certain state of mind to understand, and this happens to different people to different times. The problem is well understood (a technical problem). That helps in establishing the positive context for reaching the aha-experience.

False aha-profiles occur almost always in situations where there is a pressure to achieve the aha-experience. A programmer in a project with a majority of specifiers may feel compelled to admitting that he does understand the issue, even though he does not.

Steady profiles are the safest ones. They occur in traditional programming and in specification work using imperative languages like SDL and simple notations like MSC. They give little surprises, but sometimes the project leaders want faster progress in the development.

90% syndrome profiles may occur when a project estimates the problem to be technical. while in fact it is explorative (or even vague). The misinterpretation leads to inadequate resources for the last, but important 10%. Sometimes the last 10% understanding takes a lot of resources of validation. When the first 90% were reached quickly it is easy to ship a product before it has been adequately validated. The poor customers keep reporting errors for a prolonged time.

We try and summarize system development in Table 9 (p. 192) which shows three ideal types, the formal technique, the programmer's approach and the sketchy attitude. The

Table 9: System development ideal types

	formal technique	programmer's approach	sketchy attitude
languages	Mathematics, Z, VDM, Focus	SDL, MSC, C++, Simula, Java	OOA, OMT, UML
semantic form	declarative	imperative	mixed
user	specifiers	programmers, team	observers, team
problem	technical	explorative	vague
validation	proofs	systematic testing	walkthroughs
comprehension	aha, false aha	steady, 90%	aha, deceptive

table shows "ideal types" which means that in reality there are few cases exactly as the ones in the table, but we believe that these "ideal types" represents centers of gravity in clusters of system development approaches. In a given project it may be wise to adopt a sketchy attitude in the earliest part of the project and a programmer's approach later. If the problem turns out to have aspects of pure technical character, a formal technique could very well be applicable.

5.2 The Mn-procedure on Real Systems

5.2.1 The two facets of Mn

One should be aware of the two quite different faces of Mn:

1. As a way to make reductions, which in turn may be used

- for analysis of enclosing levels (Section 4.1 (p. 143)),
- inside other techniques (Section 5.3.8.5 (p. 217)),
- as an aid to understanding (Section 5.3.6 (p. 209)).

2. A technique to reveal engineering problems and errors
   - potential errors (Section 5.3.7.3 (p. 214)),
   - complexities (Section 5.2.2 (p. 193)).

It should be noted that Mn was created mostly to serve the first purpose. Still in practice, wanting Mn-tools, the second purpose is equally important. For engineers who want to use the Mn-approach, there is less risk to use it to find problems than to find reducibility. This follows from the fact that to find reducibility, all intricate as well as trivial problems encountered must be solved either automatically or manually.

5.2.2 Complexity expectations

We discussed the theoretical complexity in Section 4.4.2 (p. 165). Here we want to consider the practical complexity which we should expect in real systems if they are as we have supposed in Section 5.1 (p. 178).

Complexity come in three variants:

- Complexity of the Mn-procedure
- Complexity of the system under analysis
- Complexity of the reduced process

These three variations are not independent, but not perfectly correlated either.

5.2.2.1 Complexity of the Mn-procedure in Real Systems

Since the Mn-procedure executes transitions which are the same transitions as those of the system under analysis, it is reasonable to predict that the complexity of the Mn-procedure is comparable with the complexity of the system. We have also shown in Section 4.4.2.2 (p. 166) that the execution of the Mn-procedure is comparable with an exhaustive execution of the system itself. Since it is not obvious from just looking at the system under analysis how complex it is, we shall use the presumed correspondence to assert the complexity of the system in Section 5.2.2.2 (p. 197).

General

Here we want to discuss how the complexity of the Mn-procedure relates to the typical characteristics of a real time system as described in Section 5.1.2 (p. 178). Size is important for the analysis. The Mn-procedure is not extremely dependent upon the size of the system as it scales well. As long as the different components of the system are relatively independent, the size is not a limiting factor by itself. Large size is also be counterbalanced by nesting where each block can be proved reducible on its own. Complicated data expressions and especially if the behavior is dependent upon decisions where such complicated data expressions are used, may increase the complexity of the Mn-procedure considerably since all kinds of non-determinism adds another dimension to the Mn-procedure. Restructuring of the data involved or abstractions (c.f. Section 4.3.3 (p. 157)) may be necessary to be able to perform the analysis. That a system is heterogeneous does

not really affect the Mn-procedure as it works only on communicating finite state machines. Other models and description techniques will have to use other verification techniques as well. A partial analysis or a conditional analysis may be the solution. Timers are also a source of non-determinism and complicates the Mn-procedure.

As pointed out in Section 4.4.2.2 (p. 166) the overall complexity of the Mn-procedure is very much dependent upon the expected number of execution levels (either levels within the same generation or in several generations). We pointed out that the expected number may be dependent upon both psychology and system architecture. It is our belief that the psychology of the designer is more limiting than the system architecture.

Number of generations

Let us first look at what the reasonable maximum number of generations may be. We recall from Section 4.4.2.2 (p. 166) that the number of generations can be expected to correlate with the communication structure of the system since we expect to perform the Mn-procedure piecewise.

Let us therefore assume that we have a fairly simple system architecture where there is basically a sequence of processes. Let us assume that on a certain potential non-confluence pattern in the first process we detect a sequence permutation on the channel onto the second process. Let us also assume that the designer is conscious about this sequence permutation possibility. He knows that he may compensate for this sequence permutation in the second process. Otherwise he can make sure that the sequence permutation does not become worse. This will normally mean that the signals involved in the sequence permutation of the first process are independent in the second process. To introduce more sequence permutation among these signals in the second process would most probably go beyond the designers capacity to handle mentally. If we assume that the designer does leave the original signal permutation alone in the second process, this goes on to the third, and the same argument can be put forwards for that process. But sooner rather than later the designer must seek to compensate the sequence permutation. The farther the compensation is from the origin of the sequencing problem, the more probable it is that new complications enter the scene, or the designer loses control of the sequence permutation.

Let us summarize. We think that a designer cannot voluntarily introduce more than one sequence permutation without compensation and not lose control of the logic. It is possible to have some distance between the introduction and the compensation, but this is usually not longer than one or two processes. Since the communication structure corresponds closely with the generations in a piecewise execution of the Mn-procedure, we conclude that the number of generations for real systems will not exceed 3 or 4 and still be successful wrt. reducibility.

In already existing systems where the Mn-approach has not been used as guidelines, we expect to find that the introduction of sequence permutation is not intended, and that the compensation is either unplanned or due to some hidden invariant.

Number of execution levels

Let us now turn to the execution levels within a generation. The criterion for continuing to execute in one generation is normally the state different criterion. This means that from a potential non-confluence pattern, the two branches lead to a pair where the basic states are different. Still the two complete states of the node are supposedly equal in some sense since we want to prove confluence. We may also assume that stabilization is all right. Usually the state different node is a situation where there are signal sequence

differences as well as the basic state difference. The basic state difference compensates the signal difference. Very often to change generation is not so attractive because the next generation alphabet is easily non-parallel (Section 2.4.5.2 (p. 61)) due to the difference in basic states.

The Mn-procedure fans out quickly on continued execution within a generation, and we are looking for the reversal of the state difference on *every* branch. If more complications are introduced during such an execution, the designer is sure to lose his overview. Therefore for every signal in the alphabet, either the state difference is reversed or it is kept. If it is kept, the states are possibly changed, and there is another chance on the next level. Again we do not believe that the designer is able to have a conscious attitude to more than a very small number of such levels (i.e. state transitions).

Sometimes external stuttering resolves such a state different situation. The stuttering cycle is expected to be very short, often only one signal.

Conclusion

Conclusively our educated guess will be that the number of levels of one generation could normally not exceed the 3 or 4 without obvious chances that the designer has lost control. Altogether, we would expect the total depth of the execution from one potential non-confluence pattern not to exceed 5.

Complexity estimate

From what we have said above, we could make the following very rough estimate model:

1. The number of processes is p.
2. The number of basic states per process is on the average s.
3. The number of external signals per process is on the average e.
4. The number of internal input channels per process is on the average c.
5. The number of internal signal types per channel is on the average i.
6. Non-determinance factor is n. The non-determinance factor is how many more nodes there are on the next level of execution due to non-determinism. E.g. if every transition contains a non-deterministic decision which branches in two possibilities, the factor is 2.
7. The non-conformity factor is f. The non-conformity factor measure the number of nodes which need another level of Mn-procedure compared with the total number of nodes on this level.

The number of potential non-confluence patterns is $t=(p*s*(e*c*i + i*i*c*(c-1)/2))$. The number $e*c*i$ is the number of potential non-confluence patterns involving an external signal and $i*i*c*(c-1)/2$ is the number of non-confluence patterns involving two internal channels. t is also the number of nodes on the first level. The number of nodes needing another execution level is $t*f$, and the result of another execution level from these nodes will result in $(t*f*n*i*c)$ new nodes. The level factor is thus $a=f*n*i*c$. If we accept 5 levels as the maximum we get the following total number: $T=t*(1+a+a^2+a^3+a^4)$.

We have here applied the assumption (c.f. Section 4.4.2.2 (p. 166)) that a new execution level within a generation or a new generation give approximately the same final result.

The estimate will be even better if we instead of using averages for parameters s,e,c,i used actual values for each process and then calculated the number of potential non-confluence patterns t as a sum of the individual values for each process.

In Table 10 (p. 196) there are some examples of what the estimates turn out to be:

Table 10: Mn-procedure complexity estimates

Example	p	s	e	c	i	n	f	t	T
process D	1	3	1	1	2	1.0	0.3	6	14
block ABP	2	3	1	1	2	1.5	0.3	12	49
block Tk	5	1.2	1	1.2	1.2	1.2	0.2	10	15
imaginary system 1	20	10	2	2.5	7	1.2	0.2	25 K	10 M
imaginary system 2	20	20	2	1.5	7	1.2	0.2	15.8 K	1 M
imaginary system 3	20	20	2	3	5	1.2	0.2	42 K	10 M
real system 1	7	5	3	3	8	1.2	0.1	9.2 K	1 M

We see that the number of channels to each process is very important for the total number of nodes. The number of internal signals per channel is likewise extremely decisive for the estimated number of nodes. If we assume that it takes 10 ms to produce and resolve a node, it takes almost 3 hours to resolve the Real System 1 in Table 10 (p. 196). The Mn-procedure is, however, easily distributed since the analysis of every potential non-confluence pattern is independent. Put 3 machines to work, and it is done in 1 hour.

We also notice that in this model there is no gain by compositional application of the Mn-procedure unless there is reuse of block types within the total system.

The estimate is also an estimate of the execution of the Mn-procedure if the channels had been bounded to maximum 5 signals, and the execution was only M0 execution.

We conclude:

1. For small systems the estimate is not very good. It is better to take the actual values for each process and estimate from there by adding the individual estimates.
2. The complexity of the Mn-procedure is estimated to be mostly dependent upon the number of channels into each process and the number of internal signals on each channel.
3. The non-conformity factor is also very decisive and it is hard to estimate without proper empirical data. Changing it from 0.1 to 0.05 in the Real System 1, makes the estimate decline by a factor of 10. It is typical that real systems have much smaller non-conformity factor than theoretically interesting examples such as the Alternating Bit Protocol or the Brock-Ackermann example.
4. The numbers of nodes are large, but not necessarily devastating.

5.2.2.2 Complexity of the system under analysis

Having developed the estimation model in Section 5.2.2.1 (p. 193), we can now work the opposite way. From starting the Mn-procedure analysis we may estimate the complexity of the system.

Our basis for complexity estimation is simply the set of initial nodes of M_0, called Z_0. The main categories are just the categories of the evaluation known from Section 2.4.4.2 (p. 57).

1. *Confluence*. We suspect that a very large majority of the nodes will fall in this category. If all nodes fall in this category our system is “commutative” and directly reducible as discussed in [92].
2. *Non-confluence*. This is the most critical verdict. Either there is a design error, or we need to apply other techniques to prove that this node is unreachable.
3. *Sequence permutation*. There is a sequencing problem on an internal channel leading out of the process under consideration. It is necessary that subsequent processes compensate for the sequencing problem. It is likely that a generation change is needed to establish confluence. The possible exception is when external stuttering can be used as confluence criterion.
4. *State different*. There is a difference between the two basic states in complete state pair of the node. Compensation can be achieved by continuing on this generation.

Complementing these four main categories there are some subcategories related to the fact that a real system is not as simple as the basic systems handled in Section 2. (p. 41).

5. *Omitted*. We run into default transitions when determining Z0. In general we consider execution of default transitions harmful and consider that an exception. This is why we consider this situation confluent, but exceptional, and it should be reported and preferably mended.
6. *Double-sided error*. We assume that the system contains error exceptions. If both paths from the potential non-confluence pattern to the elements of the pair in the node of Z0 go through such error exceptions, we consider the node confluent even though the error exceptions may not be equal. The system is certain to end in an error when the potential non-confluence pattern is reached. If we are able to prove reducibility, and the reduction contains no error exits, we can conclude that the double-sided error could not occur.
7. *Single-sided error*. If only one of the two paths leading to the node under analysis goes through an error transition, we have a single-sided error. This is not as “attractive” as the double-sided error. Still we consider single-sided errors also confluent and the reducibility will be conditional (see Section 4.5 (p. 171)). If there is a need for a recovery for such single-sided errors, we advice to use *save* as the recovery. A given error transition of course may turn up in a large number of different nodes.
8. *Warning*. A warning is a situation where either one or both sides of the node record internal warnings. Still the situation is not considered fatal by the designers, and the monitoring system is not taking over, which means that the internal recovery is con-

sidered satisfactory. We shall consider these situations like normal situations, but we do not require that both sides of the node issue the exact same internal warning. A one-sided warning is more worrying than a double-sided warning.

9. *Save problems*. Save is the only legal way to permute signals in SDL and therefore practical. But save offer new problems. The save problem is when one path of the complete state pair in the node ends in a save, and the other contains two consumptions and no save. We characterize this as sequence permutation (or possibly state different). The save problem nodes are considered harmful and changes should be made if possible.
10. *Non-determinism problem*. Non-determinism is a source of complexity, but not necessarily a source of error. Relative to the Mn-procedure an added difficulty is that it becomes more problematic to distinguish between state different and sequence permuted situations since each element of the pair is comprised of several complete states. We characterize such a node by the term “non-determinism problem”.

Each node of Z_0 can be characterized by these categories. Some nodes may even fall in several categories. It is possible to produce a normalized *complexity profile*, which will give an overview of the process and an indication of the workload of Mn-procedure to cover this process wrt. reducibility by using the data of the profile as input to the estimation model of Section 5.2.2.1 (p. 193).

The estimated total number of nodes could be used as a *complexity index*, but since the estimation model does not distinguish between more than the main categories, it may pay off to create an index which is tuned to the application domain of the system under analysis. We feel that a linear combination of the categories probably gives a good indication.

5.2.2.3 Complexity of the reduced process

Even when the system is reducible, the reduced process may not appear very simple. Here we want to discuss when reduction does not seem to reduce perceived complexity.

Perceived complexity is definitely important when the reduction is intended to be used for improved understanding as discussed in Section 5.3.6 (p. 209). Perceived complexity may not be very important if the reduced process is only used as a preprocessor for other methods Section 5.3.8.5 (p. 217).

We have to develop a more precise notion of what we shall understand by “perceived complexity” of a process (CFSM). We could try and evaluate the process according to the criteria laid down in [11], but those criteria are not very absolute. We believe a better strategy is to try and measure how much the process has been reduced. The idea is that if the process has not been very much reduced, chances are that the result is not perceived as less complex.

Since a finite state machine consists mainly of states and transitions, we concentrate on these aspects. Furthermore we add criteria for non-determinism.

1. Compute the ratio r_s between the number of basic states of the reduction and sum of the numbers of states of the processes of the original unreduced system.
2. Compute the ratio r_t between the number of transitions of the reduction and the sum of the number of transitions of processes of the original system.

3. Compute the ratio r_d between the number of decisions in the reduction and totally in the original system.

The reader may be puzzled about why we apply the sum rather than the product to combine numbers of the original system. The combined process where no reduction has been performed would have figures equaling the product of the numbers. Our intention, however, is to capture “perceived complexity” or “the complexity which meets the eye”. We feel that it is not until one starts to simulate the execution that the sense of the product turns up. What meets the eye is the sum.

Let us in Table 11 (p. 199) see how the reduction manages in our toy examples.

Table 11: Perceived complexity reduction

example	r_s	r_t	r_d
process D	1/3 = 0.33	1/9 = 0.11	0/0 = 1
system ABP	2/6 = 0.33	2/10 = 0.2	0/4 = 0
system T1	2/6 = 0.33	5/10 = 0.5	0/0 = 1
system T2	2/7 = 0.29	4/12 = 0.33	0/0 = 1

We shall not make too vivid conclusions from the figures in Table 11 (p. 199), but for very successful reductions the ratios may be quite small. We consider the ratios very small if they are less than 0.5. If the ratios are higher than 1, the reduction is not very successful in terms of reducing the perceived complexity.

A dissatisfactory perceived complexity may also indicate that the system under analysis is not a very good unit. It is possible that reconsidering the boundaries of the block might result in a more satisfactory reduction.

5.3 Mn Methodology

Traditionally verification is a process which takes place *after* the designers believe they have a correct program. They experience, however, that correctness is hard to achieve. Verification leads to necessary changes even though the verification techniques are not necessarily targeted to improve the design. Their major aim is to determine the assumed correctness of the specification. Very often formal verification techniques must work on abstractions rather than the real system because “implementation details” confuse the verification issue. There is of course some danger that abstractions do not closely correspond to the real system, or that the removed details are more significant than anticipated (see also Section 4.3.3 (p. 157)).

The methodological impact of verification has, however, been recognized for many years. Early formal verification inspired by Hoare logic [69; 32] led to methodological programming guidelines such as “gotoless programming”. Furthermore the experience from projects involving considerable amounts of formal verification is that much improvement is gained by the insight needed by the verification effort [108; 127].

Our Mn-approach is in the tradition of “gotoless programming”. We believe that a suitable design improves the quality of the design as well as the chances of succeeding with the Mn-approach to verification and validation. If we should make a slogan to describe our general approach it must be “confluent design”. A major point is that if the verification fails to decide the quality of the system under analysis, the blame is not put on the verification method or the abstraction, but on the design. The reasons for Mn-procedure failure are analyzed and the designer should have to defend the adequacy his specification.

5.3.1 The Mn-approach assumptions: “confluent design”

“Confluent design” is based on some assumptions of how good quality design looks. Our assumptions are based on the descriptions of the nature of real reactive systems in Section 5.1 (p. 178) and the reasons for complexity described in Section 5.2.2 (p. 193).

5.3.1.1 Race conditions

Race conditions are usually harmful if they imply non-confluence. Non-confluence means that the haphazard order of signals is significant for the final result of the system. We believe this to be harmful because a system should have a purpose. This purpose is not haphazard, but definite. Still this does not mean that the signal output from the system has to be deterministically inferred from the input signals. We accept that there may be sources of non-determinism, such as alternatives induced by decisions and timers. We do not, however, normally accept that the relative speeds of the processes should introduce non-determinance.

This is a methodological standpoint and not entirely inferred from the difficulties of the Mn-procedure. Our attitude is that at least non-determinism should be explicitly stated and thus explicitly wanted. Race conditions represent a form of “hidden non-determinism”. Even when we explicitly specify a state as a **merge** state, it is still not certain that the in principle possible alternatives are actually possible.

For the expressiveness of our approach it is important that we can also express race conditions which are considered appropriate such as in specifying the Brock-Ackerman example (Section 3.5.4.2 (p. 109)) and the RPC-Memory example (Section 6.3.1 (p. 239)).

Accepting race conditions as appropriate should not be common. Acceptable race conditions should be expressed explicitly by the **merge** state mechanism (Section 3.5.4 (p. 106)).

5.3.1.2 Reducibility

SDL blocks should normally be reducible. The block concept of SDL may be used for a number of purposes [11 p 208], but they all emphasize that a block is a unit which can be conceptually understood by itself. This is not sufficient to require that they should be reducible, but it indicates that reducibility should be probable.

Our standpoint that SDL blocks should be reducible is again a methodological standpoint which will make it simpler to analyze (by the Mn-procedure) SDL systems and hopefully also to achieve an understanding of the system.

5.3.1.3 Complexity

It is our assumption that complexities in the establishing of reducibility reflects complexities in the system as such. We have not as much discussed the complexities of establishing progress, but we have given some insight into the presumed psychology behind making non-confluent systems in Section 5.2.2.1 (p. 193).

We believe that our complexity profile presented in Section 5.2.2.2 (p. 197) gives a good approximation to real complexity and that the quality of the software will be improved if the values on the complexity index is decreased.

5.3.1.4 Data

Data represents a major problem for our technique. It is important that data algorithms are structured such that they do not interfere with the Mn-procedure as such. Our assumption is that it is possible to package data in ways which make it possible to make good use of the Mn-approach.

This is according to how we perceive the nature of data in Section 5.1.2.4 (p. 179) and the ways data can be handled as discussed in Section 3.6 (p. 117).

5.3.1.5 Time constraints

Even though real, reactive systems are also usually real time systems, it is not necessarily such that time constraints play an extremely important role in the design work.

One reason for this is that the worst case scenario is often easily spotted. An implementation of this scenario is then tested and if the time constraints are not satisfied, optimization alternatives include hardware alterations as well as common software optimization.

A second reason is that practitioners will normally use surveillance timers instead of or in addition to intricate reasoning about response times.

We assume in the following that for a large and interesting class of real, reactive systems we can assume that reasoning with time constraints is not necessary.

5.3.2 How to ensure Progress?

As mentioned several times in this thesis, progress is important for the Mn-approach, but it is not the main theme of this thesis. Therefore our suggestions regarding progress are tentative and should be supplemented with other techniques.

5.3.2.1 System structure for progress

The system structure is the topology of the system as a whole. How are the components connected and what signals pass through the channels? How is the nesting structure and how are the object-oriented features used?

Progress is violated by feedback loops which never terminate. The possible existence of feedback loops is therefore interesting. A system structure where the channels form a dense web of bidirectional connections is susceptible to many feedback loops. On the other hand a sequence of processes connected by unidirectional channels form a structure which is virtually without feedback loops. The problem with complicated, web-like

system structures is that the feedback loops may be difficult to isolate and they may be so involved in each other that the total number of feedback possibilities becomes very large.

Therefore the Mn-approach wants simple architectures where feedback loops are easily isolated. We realize that for a practitioner's verification, we shall have to settle for less than formal proofs of termination. This means that it is even more essential that the feedback loops are well isolated and simple to perceive.

5.3.2.2 Process behavior for progress

The actual feedback loops can be found by trying to apply the signal ordering criterion presented in Section 2.6.4.1 (p. 80). A by-product of the search for the signal ordering criterion is a directed graph where the cycles indicate possible feedback loops. Having isolated the feedback loops, we need proper means to ensure the termination of the feedback loops.

Decisions

Very often termination of a loop is dependent upon a decision where a data expression finally reaches a specific value. Sometimes this is trivial to ensure like for a counting variable running from some very low number to a finite and constant higher bound. Other times it may be very intricate to prove that the data decision is actually going to be reached. In such situations we suggest to abstract the data from the decision by introducing a fair decision (see Section 3.5.3.2 (p. 103)) where the exit branch has positive probability.

Timers

For a practitioner, it is common to resort to timers in order to ensure progress. In an implementation there may be two kinds of timers, timers which are integrated in the system and described in SDL, and timers which are used only for surveillance and which resides in a monitoring layer.

The integrated timers (see Section 3.7 (p. 119)) induce non-determinism which complicates the Mn-procedure and sometimes timers make the specification more diffuse and less comprehensible. On the other hand they are part of the SDL specification and their expiration is considered normal rather than exceptional. The important distinction is that the expiration of the integrated timers cause merely internal recovery actions, while the expiration of monitor timers triggers actions external to the SDL system.

The alternative approach is to have the monitoring layer introduce timers which are intended exclusively to monitor progress. Whenever the timer expires, this is considered an exception and the monitoring layer will perform recovery actions which are external to the SDL system. The advantage of this kind of ensuring progress is that it does not affect the SDL system itself. The progress is determined conditionally and so will reducibility be. Either the system acts according to the reduction or an exception is called by a monitoring timer.

To choose one of these two timer strategies, it is important to have an idea of how normal the timer expiration is. If the designer believes the timer will expire sometimes, the integrated strategy should be used. If the designer believes that the timer will never expire, the monitoring strategy is applicable if it is available. We refer to Section 5.3.5 (p. 206) for more discussion on a layered approach.

A special kind of monitoring timer could be a timer which monitors the **save**s. In order to ensure strong progress, it would be interesting that the saved signals are not saved forever. Therefore a monitor timer associated with each save could be practical.

5.3.3 How to ensure Confluence?

Confluence is our business! Confluence is what the Mn-procedure detects, and confluence is what makes race conditions harmless. We also believe that a confluent system is more transparent than those which are non-confluent.

5.3.3.1 System structure for confluence

We should make all merge situations explicit. This means that we prefer that channels go all the way to a process (or block) rather than merging with another channel on the way. When there is a merger of input channels, an implicit merge component must be inserted for the Mn-procedure. A basic fair merge component results in sequence permutation such that the actual process receiving signals from such a fair merge component should compensate for the sequencing problem.

In general a web-like system structure will have more input channels into each component process than more linear structures. Since we know from Section 5.2.2 (p. 193) that the number of input channels is very deciding for the complexity of the Mn-procedure, our advice would be that one should look into whether the communication structure could be simplified such that the number of input channels decreases.

To decrease the number of channels is not necessarily to the benefit of confluence, since increasing the number of output channels actually makes it simpler to achieve confluence (see Section 4.1.1 (p. 143)). For an internal channel, it is both output and input channel. Therefore it is not obvious whether confluence becomes more difficult or less difficult to determine if an internal channel is removed (or added). As a rule of thumb a channel should be used for each individual communication dialog [11].

The structure is also easily analyzed by Mn-procedure if many of the processes are either multi-lane processes or channel-state mapped processes. These Mn-friendly categories of processes are described in Section 5.3.3.2 (p. 203).

The designer should also be careful with using the same signal types in many different places in the system. Especially if the same signal type may appear on several different channels into one process, this means that the SDL process cannot actively distinguish between signals of the two channels. Very often it is desirable to control the sequencing of the channels by saving all signals of all but one channel. There are of course situation where this concern does not apply.

5.3.3.2 Process behavior for confluence

Since the Mn-procedure works quite well in a piecewise manner, it is likely that good structure within the processes is more important for confluence than the structure of communication lines in the system.

First we define two categories of processes which are confluent by definition.

Multi-lane process

A *lane* is a tuple of the one input and zero or more output channels of a process.

A process is a *multi-lane process* if it is possible to find a set of lanes such that:

1. There are no overlap between the channels of different lanes of the process.
2. Each transition of a process defines flow on exactly one lane, meaning that the input is from the lane's input channel and its output merely onto the lane's output channels.
3. What happens next on a lane L on input I is not affected by any intermediate transitions on other lanes than L.

That such multi-lane processes are always confluent, is simple. All initial nodes will be confluent since the independence between the lanes is strong both with respect to signals and states.

It is actually the case that a lane is a specialization of an SDL service. An SDL service has its own state space and their sets of input signals must be disjunct. Our lane concept has in addition that also the sets of output signals must be disjunct and that the signals must be on different channels as well. Our third criterion corresponds to the services having their own state space.

In practice multi-lane processes come in even more specialized classes. Either the whole process has only one state, or every lane except one is defined through an asterisk state which means that their underlying service state space has only one state.

Having identified a process as a multi-lane process, it is reasonable to consider whether the process should have been divided in several services or subprocesses.

Channel-State mapped

A *channel-state mapped process* is a process where:

> For each state, only inputs from one channel is acceptable. For all other inputs, we either define **save** or internal error.

Conversely this means that for every input channel there are specific states in which input from this channel is legal. Input from this channel at other times will result in internal errors or **save**s.

It is obvious that a non-confluence pattern cannot occur in such processes because the legal input is restricted to only one channel at all points in time.

We need not be so restricted as the channel-state mapped processes, but to use save and internal error to restrict the race conditions is a good idea.

Save

The simplest way to control sequencing is to use **save**. This will normally make it easier to establish confluence. Strong progress may be more difficult to prove, but weak progress is sufficient for using the Mn-procedure to establish confluence.

Internal errors

Conceptually to apply a **save** is a way to describe that the signal may well appear at this point, but the process is not ready to consume it until later, in a more suitable basic state. Sometimes the designer want to express that a certain signal is not welcome at all in this state, in fact its appearance should have been impossible. In SDL this cannot be expressed. SDL defines default transitions for every transition which has been left out. In practice internal errors are warned through raising an exception in some way and then a recovery is performed. If there is a monitoring layer, it is reasonable that this layer takes over and resumes operation of the SDL layer at a proper complete state. Possibly a complete initialization and restart is the only proper action.

Seen from the SDL layer, an internal error is something that should not have happened, but we are happy to have caught it. Confluence is often made conditional to the existence of internal errors.

5.3.4 How to simplify Refinement verification?

Practitioners will claim that they perform refinement, but in practice the practitioners concept of refinement does not fully comply with our more formal notion presented in Section 4.2 (p. 146). The reason can be found in the way systems are actually made as described in Section 5.1.3 (p. 180).

5.3.4.1 The distillery and refinement

There are several reasons why refinement is not simple to prove in practice. If the abstract description is informal and the implementation (normally) formal, there is not much we can do until we have distilled a formal, abstract description as well.

Even a formal abstract description may not correspond to the implementation. The reason can be found in the subactivities of supplementing and revealing. The abstract description has been made at a time where the understanding of the problem domain was less complete and where not all features of the product had been settled. When the description is being made more precise and more detailed, it is also supplemented and new aspects revealed.

Some of this achieved knowledge can be described inside the interface mappings, but not necessarily all of it. The final distillery is very much to make the abstract description cover a comparable area as the implementation.

5.3.4.2 Using interface mappings

That the refinement mappings can be made in SDL as pointed out in Section 4.2.1 (p. 147), makes it possible to perform formal refinement verification even for practitioners. The main problems with the refinement verification will be concerned with the data expressions and decision structures. It is probable that not all mapping and comparisons can be done automatically.

We may assume that the abstract descriptions will often have non-deterministic decisions where the implementation has decisions with data expressions. In general data expressions will often appear only in the implementation as they are abstracted in the abstract description.

Making the interface mappings may in some cases also reveal new problems and provide new understanding. Left out situations are highlighted by having to actually specify the interface mappings.

5.3.4.3 Object orientation and refinement

Inheritance resembles refinement. There is an important difference, however, because we do not want the implementation to add more behavior to the abstraction such as the specialization does wrt. the general type. Inheritance can be used to describe an unconditional refinement relation only when the specialization merely specifies redefinitions.

With this restriction on specialization, we can see that the compositionality of refinement corresponds closely to the default constraints on inheritance in SDL-92 as illustrated in Figure 119 (p. 206). SDL has as default virtuality constraint that the redef-

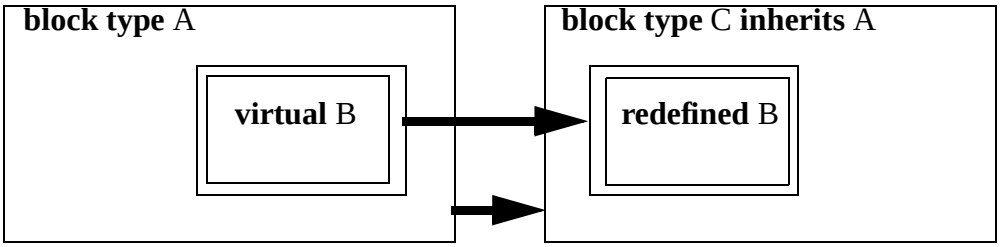


Figure 119: Refinement and Inheritance

inition shall be a specialization of the default virtual. Let us assume the invariant that inheritance is restricted such that it means refinement. In Figure 119 (p. 206) this means that the redefined B (in C) is a refinement of the virtual B (of A). According to the compositionality of refinement (Section 4.2.1 (p. 147)) we get that the encloser of the refinement, here C, is a refinement of the encloser of the refined, here A. This means that the invariant is kept at the enclosing level.

We conclude that if all inheritance relations involve only redefinitions, and if redefinitions of virtuals are specializations of the virtual, we have that it is sufficient for the refinement on the top level that there is refinement on all bottom level redefinitions.

Examples of this approach can be found several times in Section 6. (p. 229).

5.3.5 The benefits of a layered approach

In [56] there is a thorough discussion of hierarchies in system description and programming. Here we shall only point out what hierarchies we take advantage of in our Mn-approach. We will discuss virtual machine layers, monitor layers, refinement levels, nesting trees and inheritance structures.

In general we can say that these kinds of hierarchies support more effective analysis and more transparent results.

5.3.5.1 Virtual machine layers

A virtual[1] machine is a an entity which acts externally as a machine meaning that it offers a set of services to the outside. This model corresponds well with our model for interface refinement illustrated in Figure 95 (p. 148).

Typically a series of virtual machines are used to describe protocols in the well-known ISO OSI model. Each virtual machine is a protocol layer. As shown by the Alternating Bit Protocol (see Section 3.5.3 (p. 100)), reduction of a protocol can result in a very simple description. In fact in an OSI model we know that the lower level protocol is supposed to be understood as a simple signal on the upper level. If we want to show how a plain signal on the upper level is made into a protocol on a lower level we use an adaptation shown in Figure 120 (p. 207) of the general approach to interface refinement. In such a protocol layer setting we have all reason to believe that reducibility of the proto-

1. Do not confuse the use of “virtual” in connection with “virtual machines” with “virtuality” in object orientation.

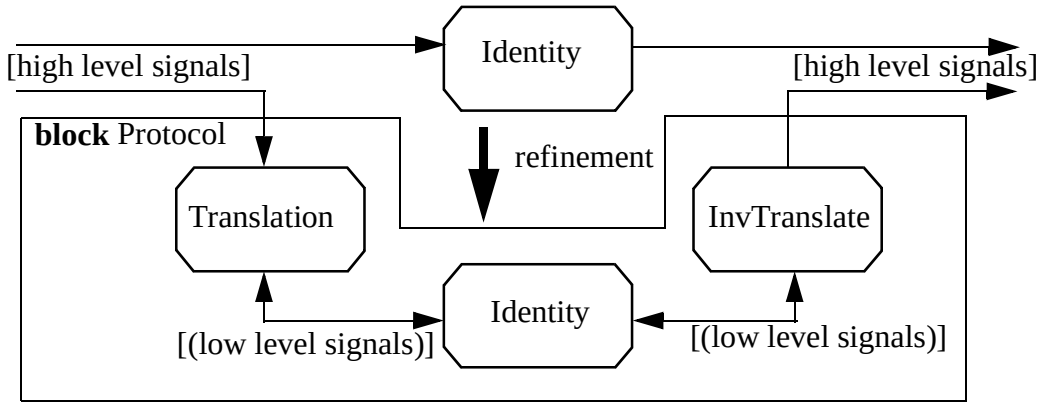


Figure 120: Protocol layers

col should hold as this is the overall purpose of protocols. We may be prepared to settle for conditional reductions as pointed out in Section 4.5 (p. 171) as the protocol may return errors instead of the wanted high level signal.

5.3.5.2 Monitor layers

We have pointed out a number of times in this thesis that our results may be conditioned by the normal execution of the processes (e.g. in Section 5.3.3.2 (p. 203)). The idea is that either the system fails, or it behaves normally and our verification results hold. The verification results which we want to get are usually progress, reducibility and refinement.

For practitioners, however, there is a very important distinction which must be made here. There is a very big difference between *assuming* error-free execution and *knowing* that all errors will be caught. If we just assume normal execution, we say absolutely noting about what happens if the execution for some reason is not normal. Practitioners often find that they spend more time on the exceptions than on the normal execution. Stronger than assuming error-free execution is to prove that undesired behavior is impossible. Practitioners will often consider this only slightly better than assuming error-freedom since their experience tells them firstly that proving is very difficult, and secondly that the sources of errors are often beyond the language semantics.

The Mn-approach is a practitioners' approach, and therefore the catching of errors is of major importance. Again there is a distinction. Either the system itself may discover an error situation or there is an external monitoring system which detects the fault. The Mn-approach is based mainly on the system finding the errors itself, such as calling an exception on impossible transitions. We do not, however, spend much time in this thesis on proper recovery. Our assumption is that there is an exception handling system which performs the proper recovery and brings the system itself back on track. Our only attempts at recovery is to use **save** as recovery on impossible transitions. There is of course full freedom to cover error situations completely within the description of the system, but this means only that error handling is a part of the application and the characteristic as an error is not significant outside the system.

We also suggest certain external monitoring features such as monitoring the speed of transitions to ensure progress. Even though such monitoring is definitely external to the system, it must interact with the system to be able to monitor it. For instance if the monitor tries to control whether a loop takes too long, it must know when it starts, and also if it terminates.

We summarize: in the Mn-approach the verification results concerning progress, reducibility and refinement may be conditioned by the detection of internal errors. To be conditioned by internal errors means that either errors are caught or the system will act according to the verification results.

The classification of execution situations is shown in Figure 121 (p. 208). The main Mn-

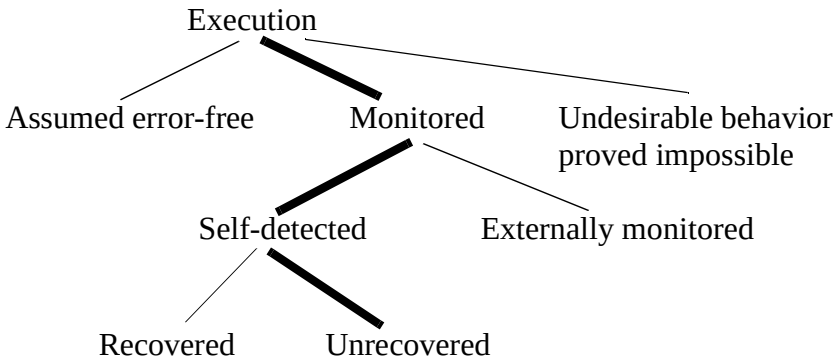


Figure 121: Monitoring of executions

approach is shown by the fat lines as unrecovered, self-detected, monitored executions.

The monitor layer may itself be described as an SDL system, but this is irrelevant for this discussion. It only means that we could apply the very same arguments on the next layer.

5.3.5.3 Refinement levels

The identification of refinement levels help the structuring of the system and the improving of the relations between the early descriptions and the subsequent design descriptions. As explained in Section 5.3.4 (p. 205) proper effect of refinement between early analysis documents and later design documents is dependent upon sufficient distilling of the abstract layer. If the early descriptions are too informal or incomplete the refinement relation will be hard to establish.

Also within the design phase, it may be fruitful to establish refinement levels to ensure that the development progresses in reliable steps. In the Mn-approach, however, we do not advocate that every development step is characterized by making another description which is a refinement of the former one. As the distillery approach emphasizes (Section 5.1.3 (p. 180)), to get a good grip on the iterations and on the combination of different approaches may be of greater value. An important aspect of the Mn-approach is also that we advocate a continuous use of verification techniques to correct the ongoing system engineering.

5.3.5.4 Nesting trees

An SDL system is normally a very distributed system and its description is as distributed as the system it describes. This property makes it possible to have a large number of different designers work in parallel with an SDL description. This again creates challenges concerning the consistency between the elements, and there is often a need for liaison activities between different subdevelopments within a project.

That SDL has a nested conceptual structure makes the liaison efforts slightly easier. Liaison efforts can take place on a variety of concepts since the total system is a tree of concepts. The Mn-approach supports strongly this distributed property. It is our attitude that reducibility and refinement can be proved within any conceptual entity of the system.

Compositionality as described in Section 4.1 (p. 143) and Section 4.2.1 (p. 147) ensures that results of lower levels of the concept hierarchy can be composed to results on higher levels. This means that analysis work done in one area of the project is not performed again when higher level results are sought.

5.3.5.5 Inheritance structures

Inheritance structures using object orientation represent layering of concepts which may be orthogonal to the other layering concepts covered in this section. As we have seen (Section 5.3.4.3 (p. 205)) inheritance may coincide with refinement, but it has values in itself even when it does not coincide with refinement. As shown in Section 3.9 (p. 133) inheritance and the Mn-approach work well together for mutual benefits.

5.3.6 Mn supporting understanding and reuse

Earlier in this section on Mn methodology we have concentrated on how the Mn-approach can help such that the system descriptions are such that the system has a number of desirable properties.We have described how the Mn-approach helps the system descriptions in making the best future system.

In this subsection we shall have a look at how the Mn-approach can be used to improve the way the descriptions can be used for the best future system *development.* Future development and maintenance are dependent upon the proper understanding of the system and the possibility to retrieve the suitable parts to maintain. Normally a well structured system which is good for itself has the best chances of being good also for maintenance, but there may be ways to improve it in the direction of understanding and ease of retrieval.

5.3.6.1 Understanding

Let us take *understanding* first. The scenario is that a new project member should make himself familiar with a reasonably large part of the system. What would be the preferred strategy? The common strategy is to give him a few very informal and very high level descriptions of the whole system, and then – rather abruptly – leave him with the source code (i.e. SDL description) of the block in which he will be assigned to do maintenance. The new project member would like to be able to rely on the top-level information he gets. He wants to consider the description he gets as a correct specification of the system which is correct. Then he wants to have similar specifications on every level of the sys-

tem down to the level on which he shall work. The common approach to project member initialization at the best yields a *aha*-profile as defined in Section 5.1.4.4 (p. 189), but too frequently appears as *deceptive*- or *false aha*-profiles. What we want is a *steady*-profile where the education of the new project member is fairly predictable. To obtain confidence in the student, there is a need for descriptions on every level which is *reliable* and *transparent*. How can this be achieved?

The common approach to facilitating understanding of a piece of software is to supply it with either an informal description (comments), or a formal specification. The informal description has the major disadvantage that it cannot always be trusted. Furthermore it is not precise enough to give the answer to all those technical questions which may be asked on this level. The formal specification is normally written in another language than the system description itself. Often a "formal specification" is declarative while the system description is imperative (prescriptive). This requires that the newcomer must be able to handle also this supplementary language as well as the system description language. Furthermore to ensure that the formal specification is reliable it is necessary to prove that there is consistency between the formal specification and the system description. When this requires more than what can be done automatically we experience the same as with informal comments – that the specification is not reliable.

The general experience as discussed in Section 5.1.3.2 (p. 183) is that there is one main description which is reliable, while all other descriptions are less reliable when they are not automatically derived from the main description. This does not necessarily mean that all descriptions but the main one should be abandoned. Alternative descriptions play important roles during the system development as they form orthogonal views to the description of the system which should be used formally to correct the main description. After they have played this important role, their update during continued development is unfortunately often neglected and the description becomes unreliable.

Many practitioners will claim that even when the formal specifications are reliable and consistent with the system description, they are often not very transparent. Therefore the formal specification is often also abstracted so much that important system description details disappear.

All of this contributes to the confusion of the new project member.

The Mn-approach is that reductions are the best specifications. The advantages are:

- The system designer needs only knowledge of one language.
- The specification can be automatically deduced from the full system description.
- All relevant details are present, with the possible exception of data abstraction.

We illustrate the difference in approach in Figure 122 (p. 211).

5.3.6.2 Reuse

Ease of understanding corresponds well with the needs encountered in a reuse situation. Assume now that the engineer works with a problem and wants to know whether there are existing types which could help solve the problem. How could he most effectively retrieve and utilize such a set of components from the rest of the system or a library?

There are three interrelated questions involved here:

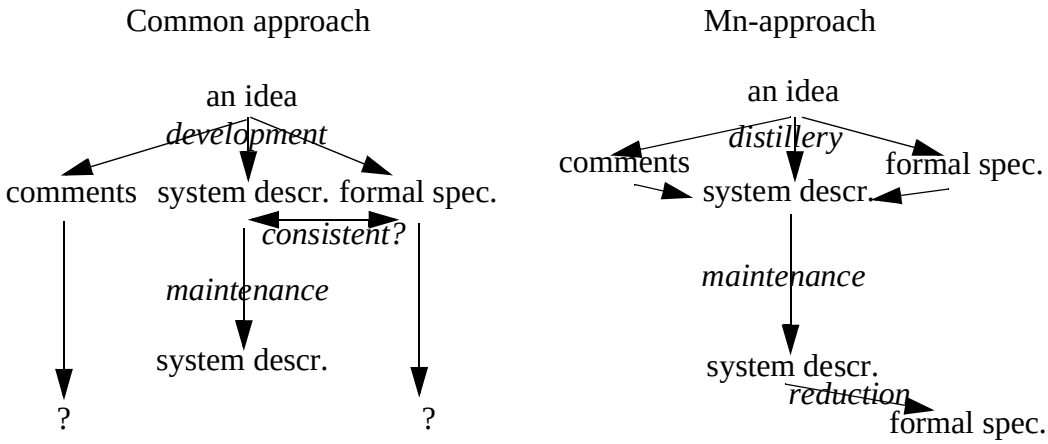


Figure 122: Reliable and transparent specifications

1. How can suitable candidates for reuse be made?
2. How can suitable candidates be found?
3. How can a suitable candidate be used?

Making candidates

To reap the most benefit from the Mn-approach a candidate for reuse should be proved reducible and the reduction should be generated. We may also consider reducibility an indicator of quality in itself.

When a component has been proved reducible and the reduction has been produced, what more is needed to make the component suitable for reuse?

In practice it is not sufficient to have a reduction in order to be reusable. Reusable components must be retrievable and transparent as well as having good quality. Here are some of the information that should be made available in the reuse repository.

1. *Name* of the component
2. *Necessary context* of the component. What is its encloser?
3. *Informal description*. Comments to the components.
4. *Functional description* as SDL process. Reduction of the full component.
5. *Confluence robustness*. Is it strongly confluent? (see Section 3.9.1 (p. 133))
6. *Structure*. If the component is a block, the first level block description.
7. *Complexity profile* as defined in Section 5.2.2.2 (p. 197). Complexity index.
8. *Quality assurance* figures such as test results and walkthrough minutes.
9. *Auxiliary* descriptions such as MSCs, test suites, invariants etc.
10. *Full description* of the component.
11. *Pointers to implementation* designs.

The name will be used for unique identification. The necessary context must be given since reusable components are not always self-contained. There may be need for other types available where it is supposed to reappear. The informal description is used for human recognition and for early screening in a large repository. The functional descrip-

tion is the formal specification of the component as a process. Confluence robustness is practical because it tells whether we may expect to find non-confluence patterns in this component when it is used in a larger context. The structure is an indication of how the component is distributed. The complexity profile is a metric of the component and should tell in a few words how difficult this component should be to maintain and to build upon. Other quality assurance figures may also be compiled. The auxiliary descriptions may have been used for test purposes, or during design. Pointers to these specifications help give a total picture of the component. Naturally there is a need for the full description in SDL of the component, and pointers to implementation designs are also helpful.

Finding candidates

Finding candidates for reuse is in itself a difficult question on which much literature has been and will be produced. It is the general question of finding something in a structure of potentially similar matters. There are two main strategies, either initiate a search or look in a structure. From a formal point of view the two strategies are not very different as searching also means to look in a structure. The difference is the use of human power.

Let us first discuss the search strategy. Firstly we have the problem of describing the search criterion, what am I looking for? Secondly there is the problem of defining when an item in the component database is close enough to be selected as a candidate. To describe behavior in a searchable way is a challenge in itself. Furthermore, when you have specified the behavior that well, there is a chance that the bulk of the work has been done anyway. In theory following the Mn-approach we have the following:

1. All items in the component base have a process description as its specification. This means that all items are comparable in form.
2. The search criterion is specified as a process.
3. The match criterion is that the search criterion is a refinement of the database component. The Mn-approach to establishing refinement presented in Section 4.2.2 (p. 149) leaves some room to define “closeness” by how much the item lacked for the establishment of refinement. A component which is “close”, but not perfect, could possibly be extended by specialization to a component which satisfies the requirement.

This application of the Mn-approach is very theoretical since we would never apply searching techniques if it were not for the fact that the amount of items in the database is fairly large. Then performing a refinement for each comparison should be prohibitive wrt. time even though it is in principle automatic. A hybrid approach is probably to be preferred. A coarse search is performed using structured comments as found in the list of attributes to a reusable component suggested above. Thereafter a finer filtering is done according to the suggested Mn-approach. We should also be aware that behavior functionality seldom is the only valid criterion which is sought by the designer. Other relevant criteria are distribution, robustness, testability, access to component designer, age, etc.

Automatic search in large libraries of suitable components is not the common situation at this point in history. Most libraries are rather small and provided that they are fairly well organized the designer himself acts as a search engine. He has not specified the search criterion in any great detail, it appears mainly as an idea in his head. From tra-

versing the structures in the item database which are supposedly logically established, the match criterion is that the designer recognizes something that resembles what he needs.

The question concerning this strategy is how the item base is best structured. Our simple answer is that the object-oriented structures should be well suited as item base traversal structures. As a people's library is divided according to a taxonomy, object orientation classifies concept the same way. The advantage of object orientation is that the classification scheme is reflected in the actual descriptions and not only in their documentation.

Using candidates

If the candidate is according to our requirements described above, the Mn-approach can use the reduction of the candidate for subsequent analysis according to the compositionality of reducibility proved in Section 4.1 (p. 143).

We should also be aware that reusing a component in object orientation means two slightly different things as pointed out in Section 3.9 (p. 133). The simplest way to reuse a component is to make an instance of it. Then the external definition given by the process specification (the reduction) should be about all what is needed.

The more advanced form of reuse is when the designer creates a new concept which is inherited from the reused component. To ascertain that the candidate is applicable will normally need a closer look at the real component and not only at the reduction. Still if the Mn-approach has been applied to all nested levels, we should need only to go one step down at the time and relate to the reduced version on that new level.

5.3.7 Mn-development

How can we achieve systems which are well structured according to the criteria?

We have discussed how the ideal Mn-friendly system specification should look, to give good structure to the system and to pave the ground for maintenance and further development. Now we want to sketch how the Mn-approach could be integrated into the development processes as a core principle.

5.3.7.1 Mn awareness

For each unit of the system consider whether the Mn assumptions "Confluent design" (Section 5.3.1 (p. 200)) are supposed to hold. If they are not meant to hold, the Mn-approach should not be applied. We repeat the assumptions of *confluent design* here:

1. Race conditions are considered harmful if they imply non-confluence.
2. SDL blocks should normally be reducible.
3. The complexity of the Mn-procedure applied to a unit reflects the complexity of the unit.
4. Data can be abstracted or they can be packaged in ways which isolate the data problems from the problems of concurrency and communication.
5. Time constraints can be abstracted or easily tested.

Even with a general positive attitude to the Mn-approach there may be units which do not fit with the Mn assumptions.

An Mn-oriented development strategy will make Mn awareness a part of the project plan and the result of the Mn awareness process part of the project requirements.

5.3.7.2 Top-down Mn

By “top-down” we mean that descriptions on high structure levels are made before descriptions on lower structure levels. The Mn-approach is essentially a bottom-up technique as it is described. Applying it top-down amounts mainly to perform the distillery approach as presented in Section 5.1.3.1 (p. 180) one step and then carefully check the step such that the distilled whole (the abstraction) refines to the precise and detailed system description. This careful checking is simply done using the bottom-up techniques.

The Mn-approach gives little assistance to the creation of the implementation. Since the process definition on purpose has eliminated all signs of internal communication, it is virtually impossible to produce a full system from the reduction. What is imaginable is that the total process definition *and* a structure definition which has come from other sources together could have enough information to suggest the process definitions of the components sketched in the structure definition. Something of this kind has been done for an finite state machine based model consistent with CCS in [111].

5.3.7.3 Bottom-up Mn

By “bottom-up” we mean that we use Mn to analyze blocks which have already been made. We may also build up a “profile” of the component which tells more about the component than whether it is reducible or not.

This is the most normal way to apply the Mn-approach, and we shall go through the Mn-strategy, which is basically a bottom-up strategy, in Section 5.3.8 (p. 214).

5.3.8 Mn-strategy

A *strategy* is more an “algorithm” which should be followed by the developers. The sequencing of the individual tasks is presented and countermeasures for non-conformance situations in the system description are described.

5.3.8.1 Progress

We consider progress first. Since termination may be impossible to assert, we may want to settle for less.

1. *Build a signal ordering*. We assume that the signal ordering criterion (c.f. Section 2.6.4.1 (p. 80)) is almost met. We will find loops and we will find needs to annotate the signals (by channel names).
2. *Consider every loop* in the signal ordering graph. Termination of the loops should be one of the following (see also Section 2.3 (p. 50)):
   - a data decision eventually exits
   - a fair decision eventually exits
   - a timer expires and the loop terminates

Each of the loops should be documented.

3. *Make sure that there is no data loop* inside a process. This may in principle be impossible to assert, but then an abstraction with a terminating loop should be applied.
4. *Consider possible abstractions* to simplify the progress establishment. Here typically abstraction of data is applicable (see also Section 4.3.3 (p. 157)).
5. *Strong progress* should then be considered. Normally we settle for weak progress, but there may be reasons to try and prove strong progress. Often weak progress and reducibility make it possible to assert strong progress from the reduction (Section 3.4.4 (p. 96)).

5.3.8.2 Confluence

Establishing confluence is the heart of "confluent design". By establishing confluence after having established (weak) progress, we can conclude reducibility. Even without having established progress, the search for confluence may be of value. Either potential problems may be found, or the reducibility can be made conditional to assuming progress.

We also produce complexity metrics as by-products of our confluence search.

1. *Categorize* the components according to a very rough scheme (see also Section 5.3.3.2 (p. 203)):
   - *One-input-channel* process (The process has only one input channel and therefore it cannot show any non-confluence.)
   - *Multi-lane process* (The process is actually a collection of "lanes" with one input and disjoint output. The clue is that the outputs are never merged.)
   - *Channel-state mapped* process (The process is such that for each basic state there is only one channel from which it accepts input.)
   - *Merge* process (The process have potential non-confluence patterns which must be considered more closely.)

The idea here is obviously that it is possible to perform this categorization very swiftly and manually. The three first categories are all trivially confluent, while the last category is the only one that requires additional analysis.

2. Make a *complexity profile* of each merge process (see Section 5.2.2.2 (p. 197))
3. Order the merge processes according to a *complexity index*.

The complexity index gives weights to the different classes of the complexity profile.

4. Take the most complex processes first and continue in the order of the complexity.
5. For each process proceed to analyze and possibly modify the critical points according to the following succession:
   - 5.1 Clarify the non-confluent situations
   - 5.2 Continue M_0 on the "state different" cases
   - 5.3 Perform generation change on the "sequence permuted" cases
   - 5.4 Try and see if external stuttering (Section 2.4.5.2 (p. 61)) could be used on the generation changed cases which turned into non-confluence

5.5 Analyze the auxiliary category situations

If confluence cannot be obtained this should be properly documented. A case which shows that there is actually an error should be produced.

5.3.8.3 Restructuring

When problems have been encountered, there should be redesign and correcting such that the problems disappear. Most often the problems and complexities originate from trying to do too much at the same time! And the cure is to restrict the behavior such that only one course of action takes place at any point in time. Said differently, “merge processes” should be made into one of the other, more confluence-friendly categories. Technically this means to apply **save** or internal errors such that the sequence of actions are forced into a more restrictive pattern.

The reason for trying to do more than one thing at the time is that forcing an order of actions will delay the action which came first, but which was unwanted. This is true, but chances are that the effect in practice is negligible. Let us assume strong progress such that saving does not imply any chance of semi-stable states (where internal signals reside in save-queue while there are no other internal signals in the system). This means that even though the save implies a delay, the signals which are to be consumed are somewhere in the system already, possibly only nanoseconds away. Since we assume this is a real-time system, it has to be configured to cope with this kind of delay anyway, or else it should have given the signals involved opposite priorities. In a real-time system it is usually not much to gain by performing a service sometimes faster.

This kind of restructuring was what we applied to the example in Section 4.2.3 (p. 151). The process V tries in Figure 100 (p. 153) to cope with any ordering of signals from the two bounds checking processes. This was proved to be non-confluent. In Figure 101 (p. 154) **save**s were introduced such that V would check upper bounds checker before lower bounds checker, and keep alternating. This does give a short delay every time the lower bounds checker finishes before the upper one, but it is reasonable to believe that the delay is minute. It is even possible to cope with accepting the lower bound checker signal first, but thereafter wait for the upper bounds signal. This third version of V would still involve saves, and the chances of delay would be even less since delay will only happen if the difference between the bounds checking is longer than a transition in V.

We should also note that action ordering through use of save does *not* transform the concurrent system into a sequential one. Take the example system D of Figure 97 (p. 151). Even when all four processes are action ordered, N,ub and lb are one-input-channel processes, and V is a channel-state-mapped process, all the processes will be basically active if the external input keep coming.

5.3.8.4 Iteration

When remedies have been applied, typically the block could change so much that there is a need for a total iteration of the Mn-approach applied to this block. Return to the progress step. The earlier conclusions on progress may have been upset by introduction of more channels and more saves.

5.3.8.5 More or less Mn

The Mn-approach works well with all other methods. The Mn-procedure can either serve as a “preprocessor” for other methods like Supertrace[73] or, the other way, use other methods as aids for its own success. These outwards and inwards uses of other techniques can also freely be mixed.

Outwards Use Mn as an aid to reduce larger systems such that common methods like Supertrace (or its commercial counterparts[40; 136]) can be used where otherwise they would suffer from state explosion. One problem is that the Mn-procedure should be used without abstraction since the reduction should be a precise reduction of the original component.

It is also feasible that reductions could play a constructive role when walkthroughs[50; 140; 51; 63] are used as the main source of validation. Using formal reduction as a part of walkthroughs could mean that the work could more easily be divided. One team could scrutinize the full structure where some reductions replace original subcomponents. Other teams may in parallel walk through the reduced components. Here it is important to realize that reducibility means neither that the description is correct nor that every significant aspect of the original component can also be found in the reduction. The correctness of a component can only be found in its interplay with other components. Non-functional characteristics like distribution, timing and implementation specific details is more easily discovered in a separate effort.

Inwards Inwards use from the Mn-approach means to utilize other techniques to solve the necessary problems encountered during Mn-procedure execution. These problems are normally:

1. Progress establishment
2. Reachability (or rather non-reachability) establishment.

The other techniques work to solve the “proof obligations” which the Mn-procedure has left in its conditional reduction (Section 4.5 (p. 171)).

5.4 Experience from an industrial case study

The Mn-approach has as its major aim to constitute a bridge between the practical world of system engineering and the theoretical world of program verification. In this thesis we have almost exclusively referred to experiences with the Mn-approach on examples taken from the world of theoretical computer science. Our references to the world of software engineering are based mainly on the experiences of the author as a system engineering consultant and as a researcher in engineering methodology.

The reason why strong empirical data cannot be presented in favor of the Mn-approach is due to the ever returning dilemma: the industry does not want to use a method without good evidence and a proper tool, real empirical data cannot be achieved unless the method is applied to real problems. Furthermore the Mn-approach has undergone improvements all along and will find its final shape only through practical application and the emergence of supporting tools.

5.4.1 The experimental tool

Still we are not entirely without empirical data. We have also programmed a tool. The experimental tool was programmed in C++ [41; 130] in 1995 and helped discover some of the general behavior of the Mn-procedure. The programming work necessary to maintain the tool was considered too resource demanding compared with the further development of the approach. The tool was not integrated with an existing SDL tool and therefore too much extra effort had to be placed in making basic software. See also Section 5.5 (p. 222) for more on Mn-tool building.

5.4.2 The practical case

Having declined the possibility to be supported by a tool in our practical experiments we had to lower our ambitions wrt. establishing reducibility of a proper system. Our goal became to see if applying the Mn-approach manually within a short period of time could give valuable feedback to the designers of the system.

We agreed with a department of Siemens AS., a branch of the Siemens corporation in Norway, to look into a part of a large piece of software which they had been produced recently. To find a system with reasonable chance of giving interesting results, the author had spoken with some of the designers to find a system where control and concurrency was more central than data management.

A system was chosen which was within a domain where the author had worked as a consultant for the company earlier. The author had not looked into the SDL descriptions of this subsystem before. The author does not have any specialist knowledge of the application domain other than through the work done for Siemens AS as a consultant on MSC methodology [59; 61; 60]. This background is of interest to this case study because it explains that the analysis, which was fragmentary and incomplete, was guided only by general knowledge of SDL systems and of the Mn-approach and not by knowledge of the application domain or the system itself.

The analysis took place on June 13. 1996 at Siemens AS in Oslo, Norway. The analysis was performed on paper by the author using pre-made schemes to facilitate the recording of the Mn-procedure. The analysis took about 5 hours, followed by 1 hour discussion with the designers of the system to present the tentative results of the study and to hear their reactions. Since the system description could not be removed from the premises, no proper reanalysis could be done after the discussion with the designers.

Since the size of the system was considerable relative to a short, manual analysis, we did not consider progress at all, but concentrated on finding problematic situations regarding confluence.

5.4.3 Main findings

The main findings of our experiment analysis were:

1. It was possible by manual analysis within very short time to find non-confluent situations which was considered harmful also by the designers themselves.
2. The system structure was fairly complicated, and was considered susceptible to confluence problems.

3. The complexity profiles of the processes were extremely biased towards the categories confluent, one-sided error and double-sided error.
4. We found several processes to be *multi-lane* processes and *channel-state mapped* processes. (See Section 5.3.3.2 (p. 203) for a definition of multi-lane processes and channel-state mapped processes.)
5. Data guards played a significant role, and data abstraction would have simplified the manual analysis.
6. The found problems were found with extremely shallow execution trees.

Even though we found non-confluent situations, we do not argue strongly that they constituted errors. Some of the non-confluent situations found were argued by the designers to be non-reachable even though this was not simple to prove. Other non-confluent situations was argued to be “in practice” non-reachable because they were dependent upon especially unfavorable timing. At least one situation was theoretically possible and considered harmful also by the designers.

The system structure was particularly complex concerning the merging channels, which implies implicit fair merge components in an Mn-approach. There were very many loops in the structure. How many of these structural loops that were also behavioral loops, we do not know since progress analysis was not undertaken.

The complexity profiles found were encouraging with respect to the expected applicability of the Mn-procedure on real systems. Hardly any complicated situations with sequence permutation and state difference were encountered. A reason for this was that the preferred simple process categories of multi-lane processes and channel-state mapped processes were found. In fact this empirical study triggered the definition of these concepts!

We found the problematic situations with extremely shallow execution trees. This meant in practice that already the initial state set Z_0 told almost the whole story which indicates that our complexity profile is very informative.

5.4.4 Some analysis details

The structure of the system analyzed is shown in Figure 123 (p. 220). There was no indication in the system definition that reducibility should be excluded for any of the subcomponents. Neither was there any indication that reducibility was desirable. As can be seen easily from Figure 123 (p. 220), the feedback possibilities are almost endless since there are two-way channels almost everywhere. The two-way channels are mainly used for protocols where the acknowledgment of the reception is used. This should indicate that even though the structure opens for very intricate loop structures, the behavioral structures are considerably simpler.

We found that very few procedures had states. Such procedures could then either be abstracted (i.e. eliminated for the sake of the Mn-analysis) or considered as expanded parts of a transition. The procedures were used mainly for rather trivial data initialization.

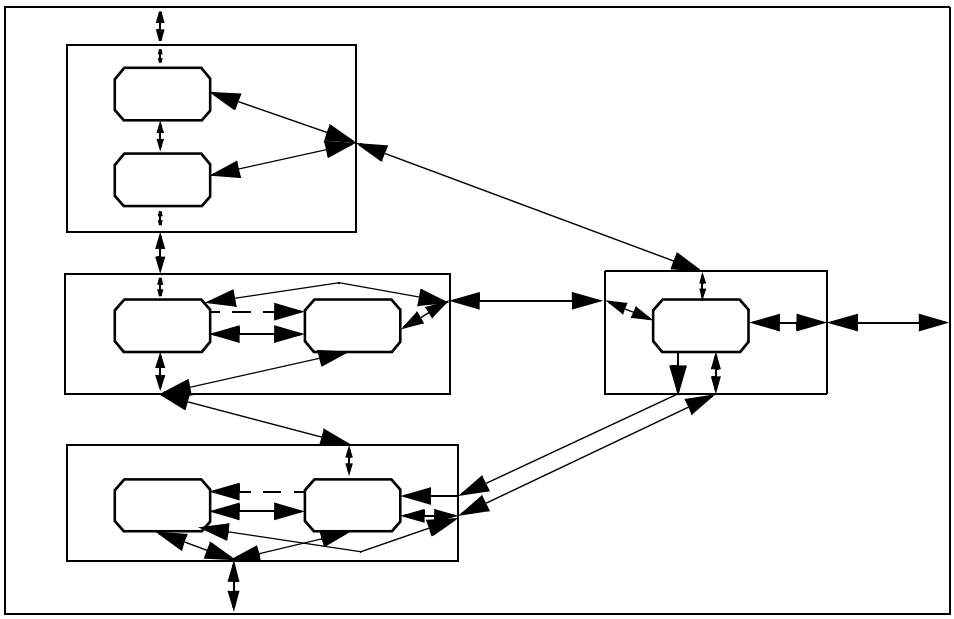


Figure 123: Structure of the system of the case

We found that very many transitions were omitted in the SDL description. The designers told us that the underlying run-time system (corresponds to our monitoring layer) would consider them internal errors.

We found genuine multi-lane processes, but their multi-lane nature was not made explicit by comments or structuring as SDL services.

There was a distinction between internal errors captured by the underlying support system and errors found and recovered within the system. Several transitions issued "SysWarning". The distinction between the two were not obvious, but a reasonable guess was that the omitted transitions were considered absolutely impossible while the SysWarning transitions were considered possible, somewhat harmful, but recoverable. This distinction is reflected in our complexity profile categories presented in Section 5.2.2.2 (p. 197).

We found that the non-determinism introduced by the data decisions was considerable. We had no time to consider whether the use of data could be decreased, or whether this was attractive to do. Neither did we consider data abstraction before we started the analysis. It is certain that data abstraction would have simplified the manual analysis, but we do not know how simple it would be to find a proper abstraction.

Our analysis was almost entirely on M_0 level only. And on the M_0 execution we very seldom tried more than finding initial state set Z_0, which means establishing the complexity profile. The state different situations we found turned out to be non-confluent by stabilization such that there was no reason to continue on M0. We tried an M1 execution once and the incomplete execution seemed to indicate that confluence would be found. During this execution we found that actual confluence would be dependent upon a large number of variants all being pairwise equal. The variants were due to arithmetic expressions which were not trivial to compare.

From the confluence search in the Mn-procedure we were able to detect similarities inside the processes which were not explicitly indicated through e.g. calling a common procedure. The similarities were needed for the benefit of confluence, but it was not very robust since the description was not structured to ensure the similarity.

5.4.5 The advice we offered

From our five hours analysis, we felt that we could offer some advice to the designers. They listened to the advice and accepted the arguments without feeling compelled to rush to change the already finished system which was tested to be all right.

We summarize the advice we gave to the designers:

1. Consider the system structure with the aim to eliminate channel merger.

This is just a technicality which can be done almost automatically, but which should be followed by some consideration of the communication patterns. This is according to our position in Section 5.3.3.1 (p. 203) where we argued that merge situations should always be explicit.

2. Review the system structure to see if the large numbers of channels could be decreased without really changing the goals of the communication.

Our worries were mainly concerned with some of the processes which received signals from very many channels. This is a source of complexity in our estimation model (Section 5.2.2.1 (p. 193)) and correspondingly it is our standpoint that it is also a source of complexity to the system. It would be a good exercise to assert whether this complexity is reflected in the complexity of the problem itself, or is added by accident.

3. Eliminate the found sources of non-confluence.

This of course is at the core of the “confluent design” presented in Section 5.3.8 (p. 214). There was especially one case where there was a possibility of problems with a retransmission in case of missing acknowledgment. Whether this was a real problem would be dependent upon the assumptions made about the lossy transmission and about timing constraints of the system. The implicit assumption of the existing description was that repeated retransmissions would not be interrupted by late acknowledgments.

4. Make explicit the nature of multi-lane processes and channel-state mapped processes.

This piece of advice was not expressed in these terms to the designers as the experience from the case study helped define these categories (see Section 5.3.3.2 (p. 203)). The experience showed that Mn-awareness of such “trivial” processes would help both for the design and the subsequent validation of the processes. We found processes which could have been divided into services, but this possibility was turned down due to expected tool problems.

5. Make more explicit the difference between omitted (erroneous) transitions and warning transitions.

The distinction between an erroneous transition and a warning transition is the distinction between a situation where the monitoring layer should take over and a situation where the internal recovery is normally considered sufficient. The Mn-approach appreciates a layered approach to execution surveillance as described in Section 5.3.5 (p. 206).

5.5 Mn tools

As indicated by the complexity estimates in Table 10 (p. 196), it is hard to cover even a moderately sized system without an automatic tool. The examples of this thesis are all extremely small (but not uninteresting).

The main problem is that there are so many cases and each case may have a reasonably large number of branches. The size of the endeavor and not the intricacy limits the systems which can be handled manually.

This is not to say that manual analysis according to the Mn-approach is without value. In situations which are small, like our examples in this thesis, or in cases where our primary goal is to find problematic spots in the system, like in our case study presented in Section 5.4 (p. 217), manual analysis can perfectly well do the job.

Still automatic assistance will always be a welcomed improvement. The more assistance the tool can give, the better, and we shall discuss what we may expect of an Mn-tool.

It is reasonable to take an existing SDL tool as base. It has the ability to simulate the SDL system provided proper input. Validators execute the system exhaustively or randomly to detect undesirable situations. There are also tools where the user may define the starting situation, and this is in fact what we need for an Mn-tool.

If we for one moment assume that there is no data and no non-determinism in the system, to simulate a system transition is identical to performing a step of the Mn-procedure. An initializing module will set up all potential non-confluence patterns and start executing. The two branches of the Mn-node resembles exhaustive simulation, but it needs not be absolutely exhaustive since our basic assumption about confluent subtrees makes it possible to choose the most suitable execution order of the internal signals. Following every new Mn-node produced there is the evaluation which must be programmed, but this is trivial. This constitutes the most basic Mn-tool.

When we add non-determinism, there is a need to include the more complex data structures into the tool. Since these structures are well specified (Figure 55 (p. 99)), it should be a fairly simple step.

The most important additional tool module should now be the symbolic execution of data. The challenges of this module are plentiful. While the simplest version could more or less just substitute the expression for the variable in an assignment and keep doing this without any simplification, more advanced tools would also try and perform some simplification. Simplifying arithmetic is a full research area in itself and not a part of this thesis. Even with a very rudimentary symbolic execution and the simplification left to the designer, the tool would get valuable added power.

When we add symbolic execution the necessary size of each node increases and compression techniques become more attractive. Smart data structures are not a part of this thesis, but there is definitely some speed gain in keeping Mn-nodes in a database during the Mn-procedure. If there is not enough room to keep it all in memory, virtual memory or a caching strategy should also give considerable effect. Random traversal or bitstate hashing like for Supertrace, find no use in the plain Mn-procedure.

It is also possible to add heuristics which could optimize the execution of the Mn-procedure. Since we may choose which internal signals to execute first, and which potential non-confluence patterns to try first, there should be room for some quick evaluation to guide the choices. We may also use heuristics to judge when a generation change should take place and when external stuttering could do the job.

Finally the Mn-tool should as a by-product produce the complexity profile for each component.

We conclude that an Mn-tool should be built on top of an existing SDL tool. Symbolic execution must be added as a special module. Heuristics and other “smartmanship” may be applied at will.

5.6 The Mn-method and the Nature of Real Reactive systems

Based on our conjectures of real, reactive systems in Section 5.1 (p. 178), how does the Mn method presented in Section 5.3 (p. 199) (possibly with the help of tools as suggested in Section 5.5 (p. 222)) correspond to the nature of real, reactive systems and their development and conceivability?

5.6.1 Mn-method applied to typical real, reactive systems?

We present here our opinions about how the Mn-method match the characteristics of real, reactive systems. We consider how the Mn-method might change the outlook of systems, and whether the Mn-approach could be successfully applied as validation technique to existing real, reactive systems.

5.6.1.1 Size

The Mn-method will probably have marginal effect on size. We do make advice to see if the communication structure could be made simpler in Section 5.3.3.1 (p. 203). The main problem is probably to decrease the number of possible loop situations as pointed out in Section 5.3.2.1 (p. 201). Sometimes multi-lane processes (see Section 5.3.3.2 (p. 203)) could be split into separate processes with their own communication structure. This would increase the number of processes, but still simplify the communication structure.

5.6.1.2 Independent components

Independence among the components of the system has two contrasting effects relative to the Mn-procedure. When components are very independent, confluence is easily established, but on the other hand, the perceived complexity of the reduction as defined in Section 5.2.2.3 (p. 198) is very poor. Dependent components imply greater problems in establishing reducibility, but the potential benefits from the reduction is better.

Therefore the Mn-method does not favor any extreme. The point is rather that independent behavior should be described in independent entities. There is not much to gain by collapsing independent behavior into fewer processes.

5.6.1.3 Nesting

Nesting is definitely encouraged by the Mn-method (see Section 5.3.5.4 (p. 209)) since the compositionality of reducibility (c.f. Section 4.1 (p. 143)) makes it possible to analyze a nested entity in steps. The inner ones are analyzed first, and then the enclosing ones based on the reductions of the earlier analysis.

Confluent design emphasizes that each entity (block type) should preferably be confluent, and this is most easily achieved if each block type corresponds to a clear cut concept in the problem domain.

5.6.1.4 Data

Real reactive systems do have data! The question is whether the data can be organized in a way which is manageable by the Mn-method.

The Mn-method does not have much to offer in the realm of data variables. The answer is simply symbolic execution and the subject is not covered much in this thesis. The important thing is to evaluate whether the data of the system can be handled by symbolic execution.

For our purposes we concentrate on systems where:

1. Data is often non-decisive or passive. The complexities of data is rarely encountered.
2. There are few complicated algorithms. These may be handled manually.
3. The data algorithms may be isolated in specific operators such that other aspects such as flow control and concurrency may be analyzed without the interference of data variable complexities.

In short, we concentrate on systems where symbolic execution of data does not pose unsurmountable obstacles.

If the system is not according to the above criteria, the developer is urged to consider a restructuring of the module in order to separate the data-intensive parts out into operators or subsystems.

5.6.1.5 Heterogeneous

Real reactive systems are typically heterogeneous as pointed out in Section 5.1.2.5 (p. 180). The Mn-method is a method mainly for the parts where the control structures and the communication is focused.

If the purpose of an analysis is more directed towards data or algorithms it may very well come in handy that the control parts are reducible such that their reductions can be used in the data-intensive analysis.

5.6.1.6 Real Time

The Mn-method does not offer much concerning real time. On the contrary we admit that confluence becomes a more complicated subject when duration has to be considered, too. The subject of timed confluence is for future research.

Please confer Section 3.7 (p. 119) for the discussion of timers.

5.6.2 The Mn-method in making real, reactive systems

Having discussed in Section 5.6.1 (p. 223) how real reactive systems correspond with systems made by or validated by the Mn-method, we shall in this section discuss how the Mn-method corresponds to how systems are actually made.

5.6.2.1 System analysis – the use of different descriptions

The Mn-method is not primarily a method for the early phases. The significance of the Mn-method is related to the formal (SDL) descriptions. Relating to the “distillery” strategy sketched in Section 5.1.3.1 (p. 180), the Mn-method is mainly concerned with the refinement relation between the precise whole and the precise and detailed system.

The refinement could be checked through our refinement technique pointed out in Section 5.3.5.3 (p. 208).

5.6.2.2 System design – the dynamics of system development

The Mn-method supports the dynamics of system development as it is described in Section 5.1.3.2 (p. 183) well, as the main feature with the Mn-method is that a system can be piecewise analyzed through reductions.

The Main Description

Many real, reactive systems use SDL and MSC in the design phase. More and more companies apply automatic code generation from SDL and the intermediate C or C++ code is not even kept. SDL appears as the main description and this is very much in accordance with the Mn-method.

Continuous development

With continuous development it is important that validation efforts can be:

1. done separately for different parts of the system,
2. can be reused when only minor changes have been made.

The Mn-method assists to achieve this. Since the Mn-method encourages that each subpart also should be reducible, this helps to set conceptual and technical boundaries which constitute natural areas for separate analysis. As we have argued in many places in this thesis, looking at the reduction of a component helps understand and clarify the component even without comparing it with explicit specifications.

The Mn-procedure is more robust towards reachability than the common reachability techniques. In principle it is possible to isolate the effect of a minor change to analysis which is centered around the changes. This is also the principle behind determining the "ripple effect" of a change as described in [139].

Even though the Mn-procedure is such that less than a process component in theory needs to be re-analyzed, a practical limitation is to reanalyze the whole process component when there are changes to it. This also means that all parts of Mn-procedures which have involved this process in some generation must be re-analyzed. This may still be less than the full analysis.

Concurrent development

The Mn-procedure is very distributed, and if the development of different parts of the system can be distributed, the analysis for reducibility can be distributed, too. Enclosing blocks may be analyzed once their components have been finished (and preferably reduced).

This makes it easier to apply validation techniques *during* the system development and not only afterwards.

Plans and reality

The Mn-method does not assure that the project plan is kept, but it makes a contribution to the effective fragmentation of the system in a way which supports fragmented validation.

This should increase the flexibility of the development of the total system and increase the reliability of progress reports.

5.6.2.3 Systems validation – how to believe they work

The Mn-method is not a way to skip testing or the scrutiny of walkthroughs, but the Mn-method reductions offer a way to experience the system with "new eyes".

The Mn-method increases the awareness of the purpose of each individual component. Furthermore the reduction may reveal complexities and effects that are thoroughly hidden in the original system. The positive effect of walkthroughs are often due to the fact that experienced engineers can "smell" trouble. There may not be any explicit specification to compare with. Reductions can be used as supporters of such "monolithic" (see Section 1.6.1.2 (p. 27)) walkthroughs.

Walkthroughs may also be used inside the Mn-procedure as informal means to assert progress, or unreachability of a non-confluence pattern. The confluence and reducibility will then be conditioned by these walkthroughs.

The Mn-approach has strong resemblance to systematic testing as what we are actually doing in the Mn-procedure is to test all potential non-confluence patterns. The complexity profile (c.f. Section 5.2.2.2 (p. 197)) can also be interpreted as a way to indicate where testing should be applied, and as such it constitutes a systematic approach to testing.

The Mn-approach aims at facilitating formal proofs through the use of reductions inside other techniques.

5.6.3 The Mn-approach to support description and understanding

If the Mn-method has been applied, can we assume that systems will be described in new ways, and that they are understood differently and possibly more effectively?

5.6.3.1 The language dimension

We have presented the Mn-approach as an approach for SDL, but the main ideas should be applicable also for other languages where asynchronous communication is the key issue. Still we should list some of the features of SDL which fit well with the Mn-method:

1. SDL is graphical, and the reductions can also be made *graphical*. This increases the structural overview of the behavior.
2. With a top-down application of the Mn-method as sketched in Section 5.3.7.2 (p. 214), we may say that *sketches* of the total behavior is compared with final design which is reduced. The comparison of the early sketches and the later reduced design should be done informally as there is little chance that the correspondence is 100%.
3. The Mn-approach is a validation techniques which *works locally.* This corresponds well with SDL where the reasoning is done locally as well.
4. The Mn-approach fits reasonably well with the *MAGIC* relations of SDL. The meta-relation is not present. The aggregate relation is represented by nesting which we have covered in Section 5.6.1.3 (p. 224). The generation relation representing dynamic process creation is not particularly well suited for the Mn-approach since reductions may be difficult to define. The identity relation is well taken care of by the handling of object orientation (Section 3.9 (p. 133)) which also covers the concept relation.
5. More than most validation techniques the Mn-approach offers an *imperative* style (SDL) also for what may be called specifications (namely the reductions).

5.6.3.2 The user dimension

In Section 5.1.4.2 (p. 188) we defined the user dimension categorizing users in four categories: programmer, specifier, team and observer. The Mn-method puts focus on the *programmer* as the important user. The Mn-method aims at making the programmer an eager validator, too. The *team* is also well supported by the Mn-method since distinct interfaces and well defined entities are emphasized. The *observer* may also find comfort in systems made through the Mn-method as the entities have reductions which can be studied in place of the original.

The *specifier* may not have much gain from the Mn-method, but on the other hand he probably does not lose much either. There will still be a need for alternative descriptions (i.e. specifications) which should be tested for consistency with the main model.

5.6.3.3 The problem dimension

The Mn-method is not sensitive to which class of problems it handles. In Section 5.1.4.3 (p. 189) we classified the problem in three classes: technical, explorative and vague. The Mn-method as such works on fairly formal descriptions, but the approach emphasizes

the aspect of understanding. The tension between high level sketches and low level implementation-like design can be informally resolved through reductions of the low level entities. This does require a certain degree of completeness of the low levels.

A monolithic approach like the Mn-approach may serve better than others on explorative problems since part of the game is to fumble around for improved understanding and not to compare formal descriptions. On the other hand when understanding is low, orthogonal approaches may cover the problem area better than only one monolithic approach.

5.6.3.4 Comprehension profiles

The Mn-method aims at contributing to a more smooth system development where validation is naturally integrated with the ongoing development. Concept awareness and constant surveillance of complexity are clues to place the Mn-method in the context of comprehension profiles as presented in Section 5.1.4.4 (p. 189).

The focus on concept awareness and the smooth application of validation techniques should prevent occurrence of the *deceptive profile*. The risk of deception lies in the designer concluding reducibility on false grounds. It is necessary to apply Mn tools for certain determination of confluence.

Since reductions may reveal hidden properties of the original system, this may appear as *aha-experiences* and lead to a (positive) aha-profile. The aha-experience should not be planned for, but it is definitely a positive experience when it happens.

The smooth application of validation techniques ranges from informal studies of progress and race conditions, through complexity profiles, to reducibility determined by Mn-tools. All together this should make it possible to assess the system components such that the *90% syndrome profile* should be avoided.

All in all we believe that the Mn-method should contribute to a *steady profile* in the system development. The idea is that no big surprises should happen or should be planned to happen. The understanding should grow with the system.

5.7 Concluding Practical Use of the Mn-approach

We have in this chapter discussed the match between the Mn-approach and real, reactive systems. We have found that the Mn-approach should fit well to support the improvement of quality in reactive systems.

We developed a reference model for real, reactive systems and compared this with an imaginary development using the Mn-method based upon the Mn-approach. The Mn-method “confluent design” was synthesized from experience and from the findings of the rudimentary industrial case study. We emphasized that the Mn-method had to be supported by an Mn-tool.

We presented simple estimates of complexity of the execution of the Mn-procedure, and argued that this complexity correlates with the complexity of the system itself.

6 The RPC-Memory Specification Problem

The Road to Wisdom

The road to wisdom? – Well it's plain
and simple to express:
Err
and err
and err again
but less
and less
and less.

6. The RPC-Memory Specification Problem

The RPC-Memory Specification Problem was posed by Manfred Broy and Leslie Lamport in 1994 and a conference was held in Dagstuhl, Germany in September 1994 which saw a number of different solutions to the specification problem.The problem specification and a number of solutions can be found in [125].

The problem appears to be a good test-bench for a validation technique. The specification problem reveals certain shortcomings of our SDL notation and triggers the suggestion for a few extensions to SDL in order to let the language handle a class of interesting cases which it did not quite handle before.

Our aim is mainly to use this problem as a test case for our Mn-approach. We present here also the development history towards the final description and not only the last versions of the descriptions.

6.1 Preliminary definitions

Components interact with one another using a procedure-calling interface. Actually we shall in this document use a signal interface. The *call* of an interaction procedure is modeled by the sending of a signal (with parameters), and the *return* of the procedure as another asynchronous signal. To represent the *raise of an exception* we simply use different types of return signals. That a procedure call paradigm is assumed means that a caller is inactive after issuing a call until receiving the corresponding return. This invariant cannot be enforced in the SDL specification, but it is used once in the analysis.

A component may contain multiple *processes* that can concurrently issue procedure calls. The return will contain the identity of the process which sent the corresponding procedure call.

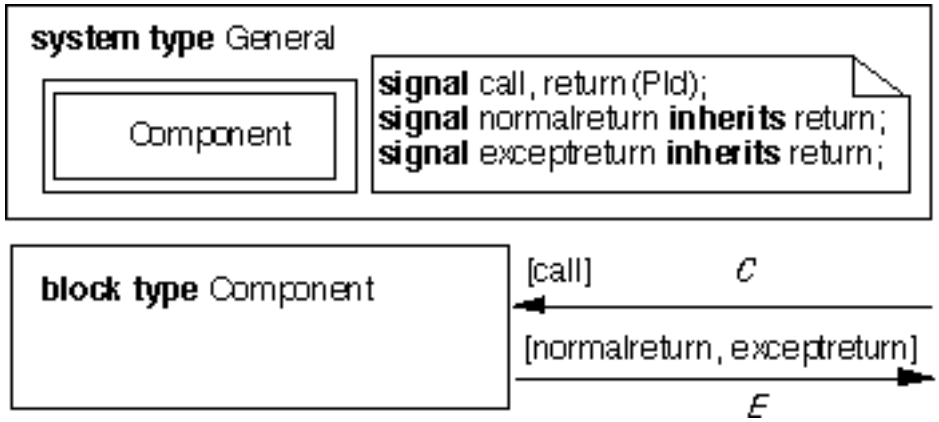


Figure 124: Component

In Figure 124 (p. 230) we can see a simple sketch of an SDL concept which describes a component of the RPC-Memory example. This component reacts to calls, and returns either normally or exceptionally. Components which are initiators (rather than receivers) of calls have a similar, but symmetrical interface.

For our purpose the component may also be seen as a process type when we want to give it a direct behavioral description.

6.2 The (unreliable) Memory and the Reliable Memory

The first problem is to specify a memory component. A memory component receives read or write requests which observe or update the individual entities of the memory. In its basic version the memory is not totally reliable meaning that whether a write or read operation is successful cannot be guaranteed. The write operation tries “on its own” an indeterministic number of times to write on the memory before it gives up and returns a MemFailure. These tries are independent and subject to interleaving with similar attempts onto the same memory location from other processes.

6.2.1 Problem 1a)

The problem is to specify the (unreliable) memory and the reliable memory.

6.2.1.1 Memory (unreliable)

We first define the structure of the (unreliable) Memory component in Figure 125 (p. 231). For readers unfamiliar with SDL, we note that the dashed arrow from MemCommHandler to wa denotes dynamic process creation. The dashed arrows outside the frame designates existing gates which were defined in Figure 124 (p. 230). In Figure 125 (p. 231) the gates get additional signaltypes.

The memory structure as shown in Figure 125 (p. 231) is derived due to the need to model the repeated tries to write onto the memory locations. There is one WriteAgent for each Write call and the WriteAgent then takes care of the repeated calls to Mem which is the real owner of the memory. The WriteAgent decides when to give up. The

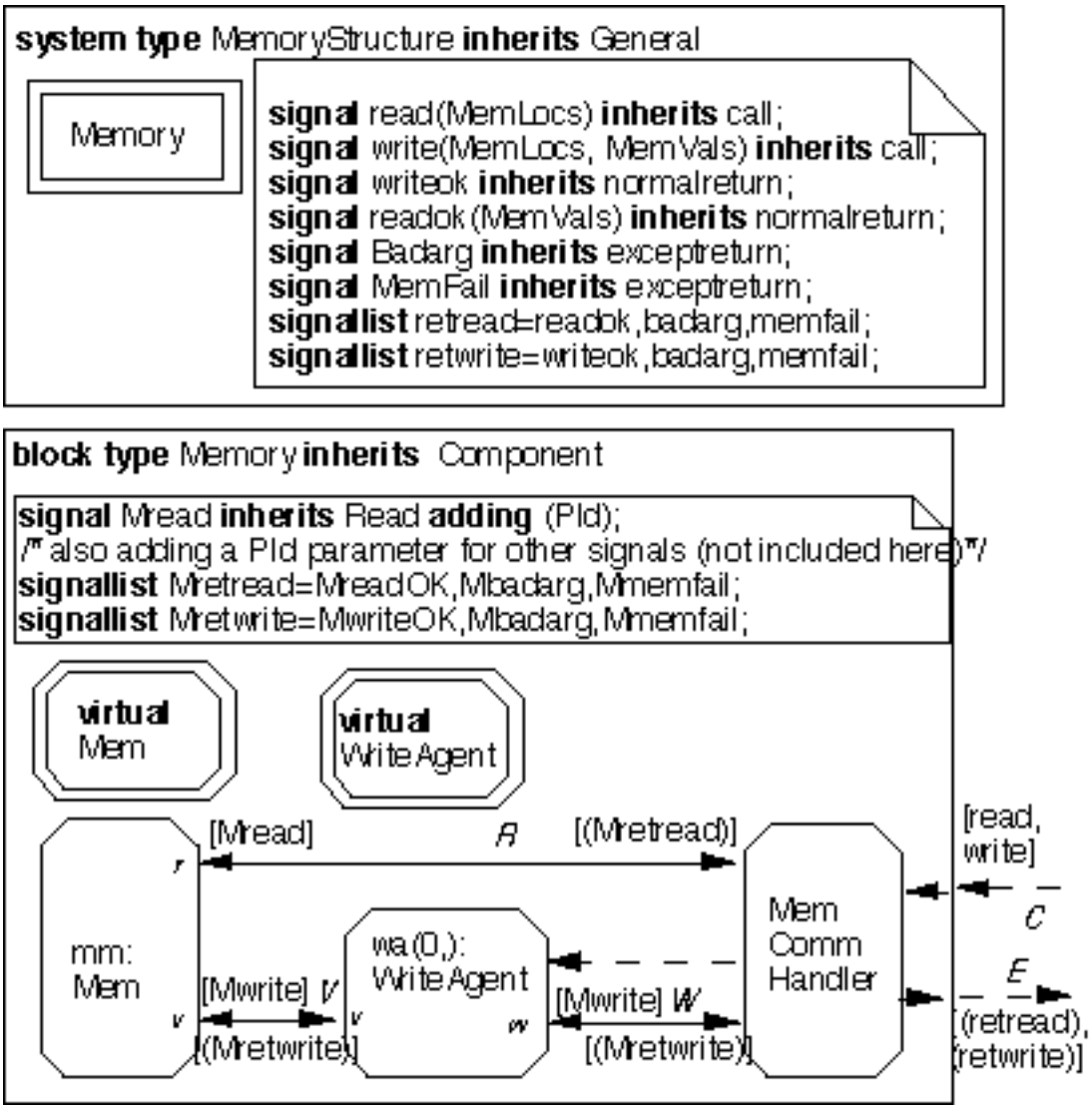


Figure 125: Memory structure

MemCommHandler handles all calls and returns. It determines whether the call has bad arguments and then raises BadArg exceptions. Remember that there may be multiple calls active in the Memory concurrently. Therefore the internal signals have added a parameter which is a pointer to the originator of the call. This is used by the MemCommHandler when it conveys return signals.

Having shown the memory structure we want to define the behavior of Memory by defining the processes. In Figure 126 (p. 232) we start by the MemCommHandler.

The MemCommHandler adds to the signal the extra Pid found as the SENDER which is predefined function in SDL, and sends it onto the interior of the Memory block. For read there is no problem as the request is merely transferred to the Mem, the owner of the memory array. With write we choose to have a special agent to keep track of the repeated attempts to alter the memory location. This agent is created for each call by the create symbol wa. The write request is then transferred to this agent. When the other

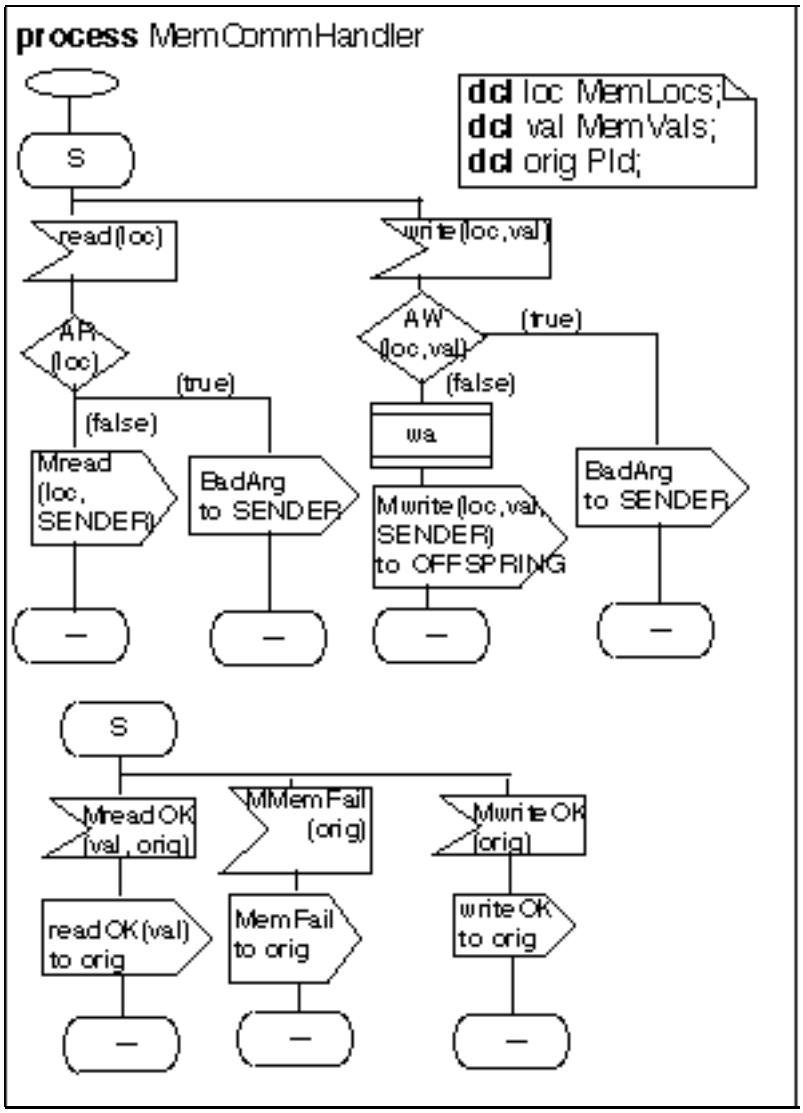


Figure 126: MemCommHandler

interior parts of the Memory have finished their work, the result is returned through the MemCommHandler. By the PId parameter of the returning signal, MemCommHandler knows where to relay the return signal.

Notice that bad arguments can be detected already by this process by the Boolean expressions AR(loc) and AW(loc,val) and the proper exception signal is sent immediately back to the caller, while for memory fail (MemFail) and for success we shall have to wait for the internal communication of Memory.

In Figure 127 (p. 233) we show the WriteAgent.

When the WriteAgent is created it will wait in state First. There it will receive the Mwrite signal which it relays onto Mem and then it waits in state Repeat. In state Repeat another Mwrite signal should be absolutely impossible, but the other returns should be handled. MwriteOK is simply relayed to the MemCommHandler.

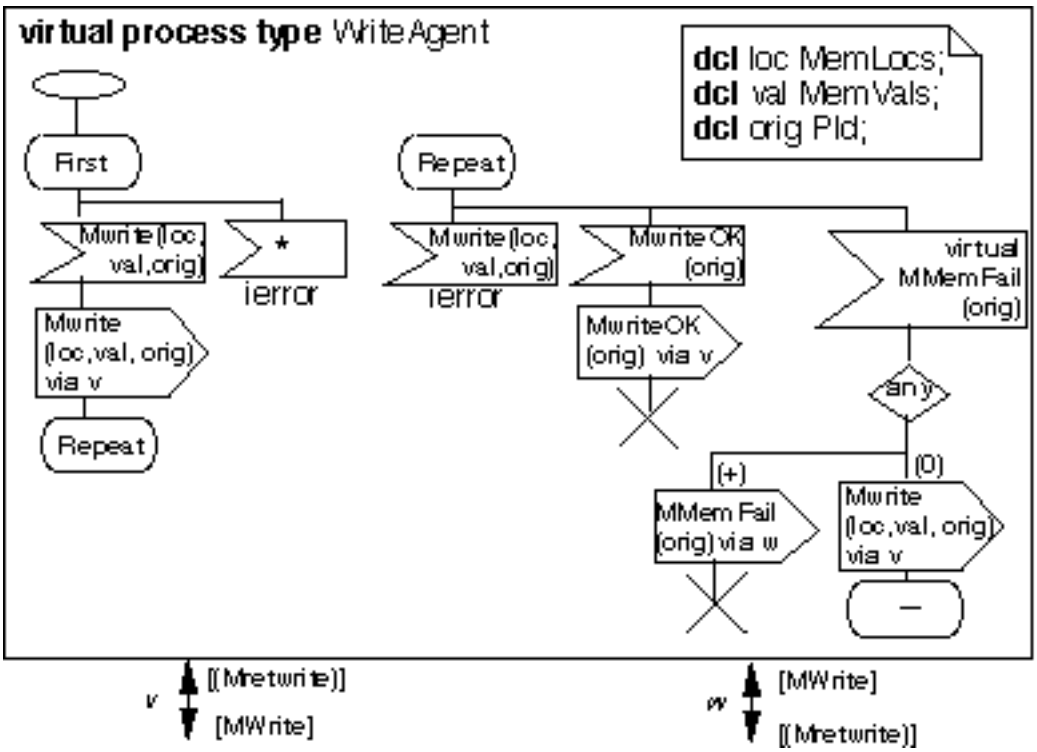


Figure 127: WriteAgent

The interesting feature is the handling of the memory failures. Whenever a MMemFail returns from the Mem, there is an indeterministic decision whether the unfortunate result should be relayed on to the MemCommHandler or whether another try should be attempted. The choice is indeterministic, but we define that the probability should be positive that the loop will terminate by issuing the MMemFail further to MemCommHandler. We designate this positive probability by a "(+)" on the branch. If we know nothing about the probability, i.e. the probability may also be 0, we may designate this by "(0)" like we have done on the branch that sends the Mwrite request back to Mem. If no indication is given on the answer branch, the default is "(0)". The reader should appreciate that this is an extension to the anyvalue decisions in SDL presented first in Figure 59 (p. 103).

The (Repeat,MMemFail)-transition is specified as **virtual** because we want to redefine the transition in specializations of WriteAgent (see Figure 130 (p. 235) and Figure 133 (p. 237)).

The branches which end in ierror are considered impossible or representing an internal error which we do not want to specify further. Our upcoming reductions will be conditioned by internal errors, meaning that either the system behaves as the reduction or an internal error will occur. The internal errors can be considered transitions which are actually not present in the transition system. It is sometimes possible to supply proofs that these transitions will not execute. In WriteAgent we could also have used **save** for the supposedly impossible transitions.

Finally in Figure 128 (p. 234) we present the owner of the memory itself, the Mem process.

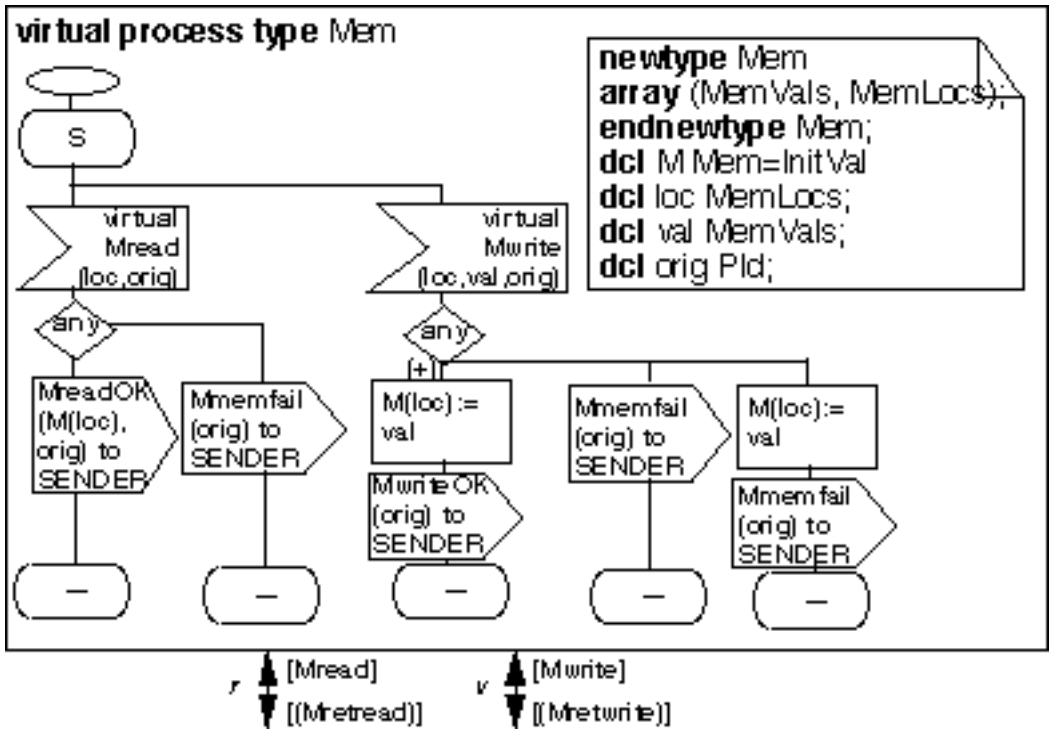


Figure 128: Mem

We see in Figure 128 (p. 234) that a memory return directly answers to a request, but the memory may not be successful. While reading is a one attempt effort, writing may involve looping between the WriteAgent and Mem and the memory will continue to try and complete the writing onto the location indicated. Notice that the memory may or may not have changed the memory when it returns a MemFail. Notice also that we have specified that the transition returning a success has a positive probability. This implies by itself that a loop of memory fails will eventually terminate by a positive MwriteOK if it does not terminate by a MemFail to MemCommHandler from the WriteAgent.

The reader should not necessarily pay any attention to the **virtual** specifiers only studying the Memory specification. These will be used in the sequel to specify the ReliableMemory in a compact manner.

6.2.1.2 Reliable Memory

The ReliableMemory is specified to do the same as (unreliable) Memory, but no MemFail exceptions will be raised. To get the most out of it, we still keep the possibility that the ReliableMemory has to try multiple times before it returns from writing with a success. Therefore it is reasonable to keep the structure of the Memory, but make some modifications in the form of specializations and redefinitions.

We express in Figure 129 (p. 235) that the structures of ReliableMemory and Memory are identical, but the used process types have been redefined.

The WriteAgent will always give the Mem another try when Mem has raised an internal MemFail.

Figure 129: ReliableMemory has the same structure as Memory

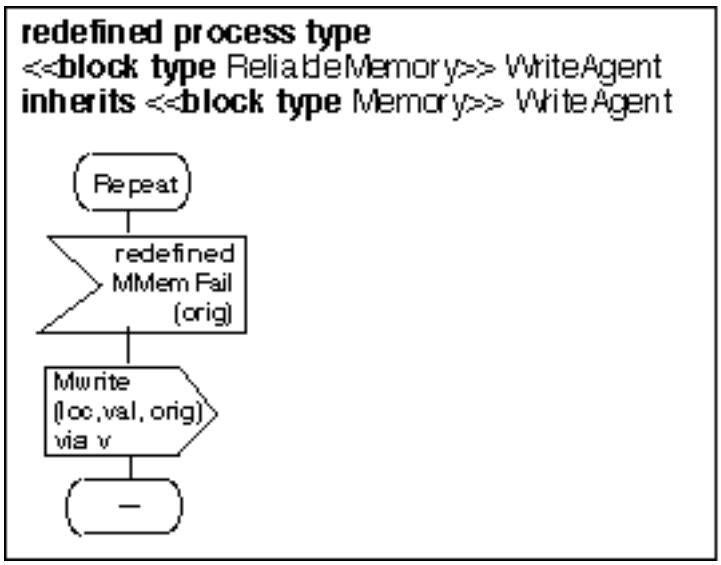


Figure 130: WriteAgent of the ReliableMemory

We see in Figure 130 (p. 235) that there is only a very minor modification of WriteAgent. Whenever the Mem has to report a MemFail, the WriteAgent just tries again.

The redefined Mem shown in Figure 131 (p. 236), the owner of the real memory, is also almost identical to the one in Memory, with the exception that reading cannot give any MemFail.

6.2.2 Problem 1b)

The problem is whether ReliableMemory is a valid implementation of Memory. By R implementing M, we will understand the same as R being a refinement of M as defined in Section 4.2 (p. 146).

We have in our description used object-oriented inheritance relations as suggested in Section 5.3.4.3 (p. 205) such that it is simple to see the difference between the Memory and the ReliableMemory. We shall go through the modifications in ReliableMemory to see that all behaviors of ReliableMemory is also possible in Memory.

Figure 129 (p. 235) shows that there is no structural difference between the (unreliable) Memory and the ReliableMemory.

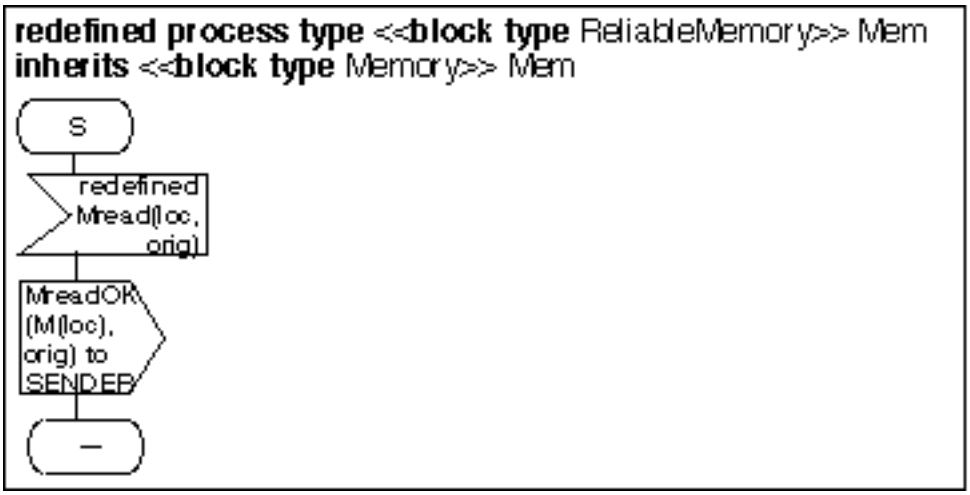


Figure 131: Mem of the ReliableMemory

We start by comparing versions of Mem. Figure 131 (p. 236) shows that the only difference is the ReliableMemory cannot return MemFail as a response to Read. Thus all behaviors of Mem in ReliableMemory can also happen in Mem of Memory provided the same external stimuli. From the rules of refinement stated in Section 4.2.2 (p. 149) we get that Mem in ReliableMemory is a refinement of Mem of Memory (Figure 128 (p. 234)).

Figure 130 (p. 235) shows that the WriteAgent cannot escape by issuing a MemFail in ReliableMemory like it can in Memory (Figure 127 (p. 233)). Again behavior which is possible in Memory is simply removed in ReliableMemory and thus all transitions of WriteAgent in ReliableMemory are also present in the corresponding WriteAgent in Memory. Is then refinement established between WriteAgent of ReliableMemory and WriteAgent of Memory according to rules of Section 4.2.2 (p. 149)? No, not quite, since WriteAgent of Memory has a branch with positive probability which is not present in WriteAgent of ReliableMemory, we cannot conclude refinement without some more reasoning.

Since for every behavioral branch, Memory has at least the same alternatives as ReliableMemory, we may conclude that for any *finite* behavior of ReliableMemory, the same behavior may happen in Memory.

Is it possible that there is an *infinite* behavior in ReliableMemory which cannot take place in Memory? We have in ReliableMemory removed the alternative which has positive probability in Memory, namely to return MemFail from WriteAgent to MemCommHandler. We must consider the behavior which infinitely visits this decision. Is it possible that in ReliableMemory there is an infinite loop where the WriteAgent tries again and again to get something different from MemFail, but Mem keeps returning MemFail? This cannot happen in Memory because there is the escape that the WriteAgent raises a MemFail and *this possibility has positive probability* meaning that the looping cannot continue infinitely. Now the situation is that not even the ReliableMemory can loop infinitely because in Mem there is a positive probability for a successful return.

We summarize:

1. Mem of ReliableMemory is a refinement of Mem of Memory following directly from our rules for comparing transitions in Section 4.2.2 (p. 149).
2. WriteAgent of ReliableMemory is a refinement of WriteAgent of Memory due to the rules of comparing transitions in Section 4.2.2 (p. 149) and supplementary reasoning about infinite behavior given above.
3. ReliableMemory is a refinement of Memory according to rules for refinement and inheritance presented in Section 5.3.4.3 (p. 205).

6.2.3 Problem 1c)

The problem is whether a process which only raises MemFailure exceptions can also be considered an implementation of Memory.

We define this kind of memory called FailMemory in a similar way to ReliableMemory. The structure is shown in Figure 132 (p. 237).

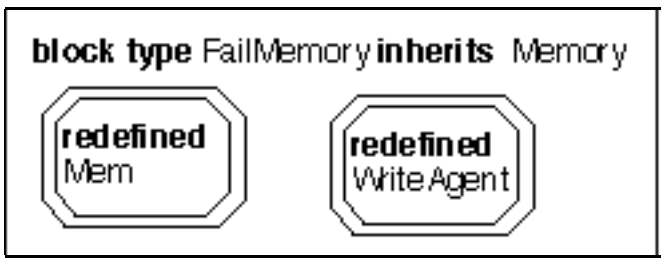


Figure 132: Structure of FailMemory

For the behavior we also follow the strategy used for ReliableMemory as shown in Figure 133 (p. 237).

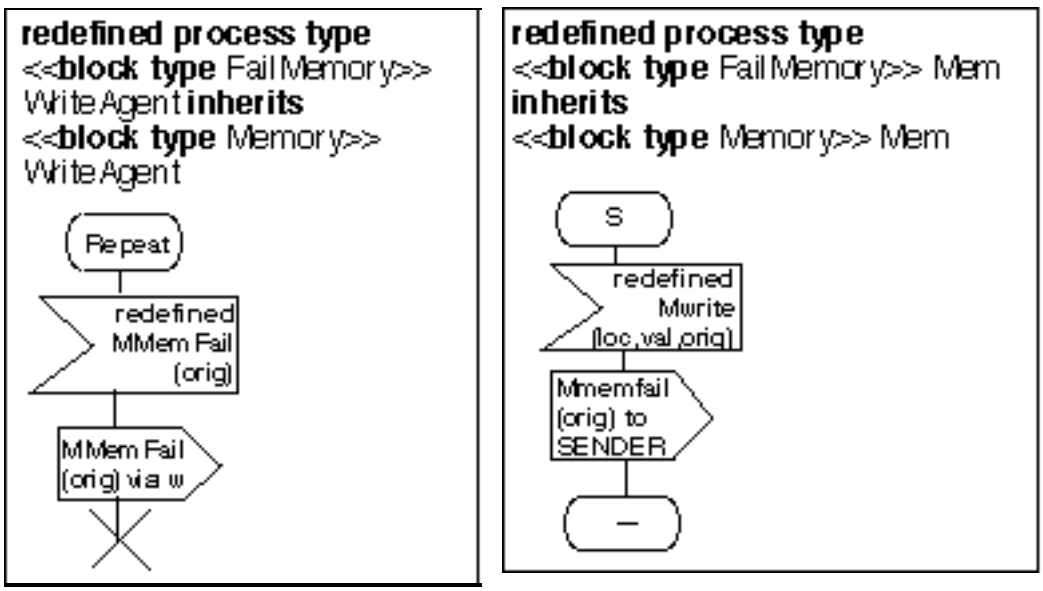


Figure 133: Behavior of FailMemory

Since the alternatives which return MemFailure are no different technically from those alternatives returning otherwise, the same argument as given in Section 6.2.2 (p. 235) can be applied to the continuously failing component, FailMemory.

However, our conclusion regarding FailMemory is not the same as with ReliableMemory.

Our specification of FailMemory as depicted in Figure 132 (p. 237) and Figure 133 (p. 237) is not quite the same as "does nothing but raise MemFailure" because the MemCommHandler which has *not* been redefined will in fact return BadArg if the arguments are out of range. This means that if the arguments are out of range, our Memory specification (and inherited FailMemory) will allow no other returns than BadArg. Refer to Figure 134 (p. 239) for an even simpler specification of a FailMemory process. It is clear that this simpler specification of FailMemory is not an implementation of Memory since it does not return BadArg when the arguments are out of range.

If we exclude BadArg, there is no doubt that whatever *finite* behavior FailMemory can show, Memory can also show. Still we cannot help thinking that there is not much help in FailMemory if you want an implementation of Memory. We also specified in Memory that there is a positive probability for success when the WriteAgent wants to write on Mem. This is enough to define that an implementation of Memory must have the ability to return a successful write (and a successful read)! The reason is that an infinite stream of write signals can in FailMemory return an infinite stream of MemFail, but in Memory there has to be a successful write.

The way this is described is hardly very transparent and not very explicit. If this positive probability alternative had been a part of an internal loop, it may have had other exits and the effect had not been the same.

SDL-92 can define that some behavior is *necessary* by the **virtuality**/**finalized** constructs combined with virtuality constraints. The problem is that there is no way to address the different *alternatives* of a non-deterministic decision. If such a notation for virtual non-deterministic alternatives within a transition existed, we could describe the alternative which returns a successful write as non-virtual (**finalized**) while other alternatives may be **virtual**. This would ensure that the successful writing had to be part of any specialization of Memory. Then it would not be possible to describe FailMemory as a specialization of Memory. As a practitioner this is more the kind of Memory concept which is practical when expressed as a requirement. We are not interested in implementations which perform any random subset of the desired behavior. There is normally some core behavior which all implementations should have the possibility to perform.

We conclude:

1. The straight forward FailMemory defined in Figure 132 (p. 237) is not a refinement of Memory since there is infinite behavior which can occur in FailMemory which cannot occur in Memory.
2. This infinite behavior of Memory is not very transparently described. Improved notation would be encouraged.

3. The super simple FailMemory defined in Figure 134 (p. 239) cannot be an implementation of Memory if we include BadArg in the way it is done in MemCommHandler since bad arguments would produce MemFail in FailMemory, but BadArg in Memory.

6.3 Reducing Memory *to a process description*

We have now specified Memory, ReliableMemory and FailMemory. They are all described as SDL block types. We were able to prove implementation relations between the types through inference rules based on syntactic similarity.

But how should we have decided whether FailMemory was an implementation of Memory if FailMemory was specified by an SDL process type as in Figure 134 (p. 239)?

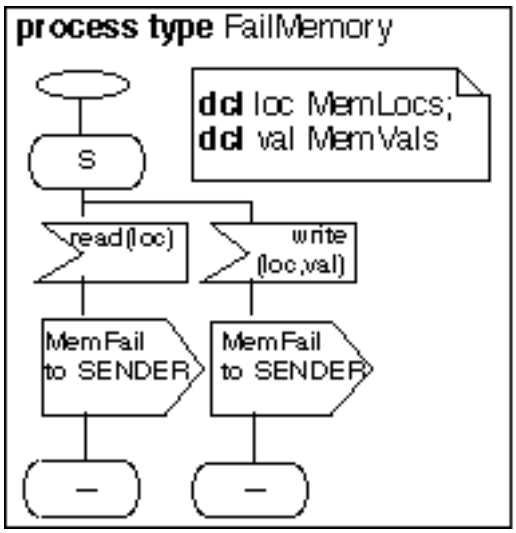


Figure 134: FailMemory as SDL process

Following the strategy for determining refinement presented in Section 4.2 (p. 146) we need to reduce Memory to a process description before we compare FailMemory and Memory transition by transition.

6.3.1 Why Memory *is not reducible as it is specified*

We recall that reducibility consists of two aspects. Firstly the system must be progressive, and secondly the system must be confluent. Memory is not confluent because Mem is not confluent wrt. different MWrite and MRead signals on the same location which may arrive concurrently. This is actually a fair merge component similar to what we described in the Brock-Ackerman example in Section 3.5.4.2 (p. 109). In order to obtain confluence we must use the **merge**-mechanism defined in Section 3.5.4.1 (p. 106).

It is actually a matter of explicitly defining the non-deterministic effect of the fair merge obtained at the input port of the Mem process.

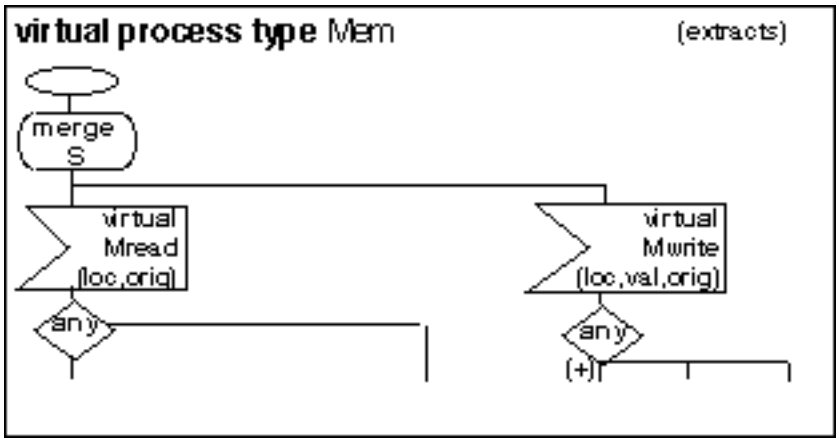


Figure 135: Extracts of Mem with merge state

The semantics of a **merge** is that whenever an Mwrite or Mread is received, there is a non-deterministic choice between consuming the signal or saving it. The probability of consuming it is positive at every scheduling point implying that a signal cannot infinitely be spontaneously saved. In effect the spontaneous save is a way to describe all possible permutations of signals and thus describe a *fair merge* situation by a finite notation.

6.3.2 Progress of Memory

The first requirement for reducibility is progress. Progress means that for all finite input streams the system will produce a finite output stream and subsequently execute no further transitions.

The simplest criterion for progress is that every transition produces less than it consumes. By ordering the signals partially such that every transition produces signals of less value than it consumes we know that the system will reach a waiting state for each external input as long as there are no spontaneous transitions. This is signal ordering criterion which was first mentioned in Section 2.6.4.1 (p. 80).

For our Memory system we have (almost) the following partial order:

1. read -> Mread -> (Mretread) -> (retread)
2. write -> Mwrite to WriteAgent -> Mwrite to Mem -> (Mretwrite) to WriteAgent -> (Mretwrite) to MemCommHandler -> (retwrite)

There are transitions, however, that do not produce less than they consume. There is an alternative in the WriteAgent which consumes MMemFail and produces Mwrite back to Mem. This violates the given partial order. This loop, however, cannot continue forever because there is positive probability on the alternative which returns success (MwriteOK) when consuming the Mwrite. Furthermore there is positive probability in WriteAgent to return MemFail.

All together we may conclude that Memory is progressive since the only loop is terminated by positive probability of exiting transition (fairness).

The strategy to determine progress used here was laid down in Section 5.3.8.1 (p. 214).

That ReliableMemory is also progressive follows along exactly the same lines as with Memory. Here we should note, however, that ReliableMemory contrary to Memory has the need for extreme fairness (see Section 3.5.3.3 (p. 104)). Mem is involved with a number of different WriteAgents and none of them should "starve". This means that for an infinite subsequence of Mem-decisions relating to one particular WriteAgent, the helpful directions will occur infinitely many times. In Memory it sufficed to have simple fairness in each non-deterministic decision in WriteAgent.

6.3.3 Confluence of Memory

We shall go through all the components of Memory and explain why they cannot contain a non-confluence pattern. It is quite obvious that confluence of Memory can be determined automatically. Furthermore the reasoning is valid also for ReliableMemory since no parts of the reasoning is upset by the difference between the two definitions.

6.3.3.1 Mem

All potential non-confluence patterns of Mem are resolved by the merge-mechanism which was introduced in Figure 135 (p. 240).

6.3.3.2 WriteAgent

WriteAgent is a *channel/state-mapped process* (defined in Section 5.3.3.2 (p. 203)) meaning that the channels divide the state space such that in a given state only signals of one channel are legally consumed. Other signals are considered internal errors.

Internal errors can be interpreted in three ways:

1. The internal errors are separately shown to be impossible.
2. The internal errors are interpreted as saves which means that confluence is simple. If the enclosing system *is* reducible, *and* the saves do represent impossible transitions, strong progress will follow from the reduction as shown in Section 3.4.4 (p. 96).
3. The internal errors are considered outside the scope of the proof. This means that the proof is partial. If a system is reducible, it means that we have proved that either the system acts like the reduced process or it performs an internal error.

In the case with the WriteAgent it is possible to prove that the internal error transitions cannot occur.

1. WriteAgent does not contain any transition with nextstate First. Therefore the only possibility to reach state First is when WriteAgent is created.
2. From analyzing MemCommHandler we find that the only place where WriteAgent is created is when Mwrite is sent to it just afterwards. In no other place is Mwrite sent to WriteAgent.
3. From analyzing Mem, we see that (Mretwrite) is only sent to WriteAgent in transitions triggered by Mwrite received from WriteAgent. WriteAgent sends Mwrite when entering state Repeat. Therefore in state First WriteAgent cannot receive anything else than Mwrite. We have shown that the internal error of First is impossible to reach.

4. Mwrite is sent to WriteAgent only from MemCommHandler just after the creation. Since no other signals can be received before it (from what we found above), this is the signal that will be consumed first. WriteAgent will then move to state Repeat. Since there are no other places that Mwrite is sent to WriteAgent, we must conclude that Mwrite cannot be consumed in Repeat and thus the internal error of Repeat has been shown to be impossible.

If we choose to interpret the internal errors as saves, the reasoning would be even simpler. Since the saves will still make the WriteAgent a channel/state-mapped process, confluence is established. This is all we need for now. The reduction will later show whether the saves are impossible. If the saves are possible, there will be semi-stable states in the reduction.

If we want to make an argument for strong progress independent of the reduction the following should hold:

1. Invariant for WriteAgent'First: There is an Mwrite signal on channel W.
2. The invariant is true after the creation of WriteAgent since it is followed immediately by the sending of Mwrite.
3. Since WriteAgent never returns to First, we need only consider this single case.
4. Invariant for WriteAgent'Repeat: There will eventually be a (Mretwrite) signal on channel V.
5. There are two transitions leading to Repeat. One comes from consuming Mwrite in First, and the second comes from consuming MMemFail from Mem.
6. Both these transitions ensure that Mwrite is relayed to Mem.
7. Eventually Mem will consume Mwrite and produce an (Mretwrite) signal on channel V.
8. WriteAgent will not leave Repeat without receiving an (Mretwrite) signal.
9. Thus we have shown the WriteAgent'Repeat invariant.

The reader should appreciate that there is no doubt that the simplest approach is to be content with the saves, and wait for the reduction. After all it is reducibility we are after anyway.

6.3.3.3 MemCommHandler

MemCommHandler is basically a *multi-lane process* (defined in Section 5.3.3.2 (p. 203)). A *lane* is a set of input and output channels such that there are no overlap between the channels of different lanes of the process. A process is a multi-lane process if all transitions can be placed in a lane meaning that its input is from the lane's input channel and its output merely onto the lane's output channels. Multi-lane processes are always confluent.

We consider every PId (process identifier) as an individual channel.

In MemCommHandler one set of lanes go along input on C and output on W, and the other set of lanes go along input on W and output on E. There is also a set of lanes (concerning BadArg exceptions) which has channels along C as input and channels along E as output. If we consider the PId of the external processes which sends a write or a read

as one channel each on C and E, we have that there is a unique PId for every WriteAgent. Thus there is a functional correspondence between singular bidirectional channels of W and PIds of C and E.

For this scheme to hold it is necessary to assume that one process in the environment has at most one pending call to the Memory at any point in time. This is to ensure that all PIds are actually distinct. If this assumption is violated it is a possibility that answers to different requests overtake each other with chaotic results in the external process. We feel that it is a reasonable interpretation of a procedure call scheme that an environment process has at most one pending call to one other component.

Since MemCommHandler can be interpreted as a multi-lane process, we may conclude that there cannot be any non-confluence patterns in MemCommHandler.

6.3.4 Reducing Memory

We have now shown that Memory (and ReliableMemory) is progressive and confluent. Therefore we may conclude that Memory (and ReliableMemory) is reducible. We shall perform the reduction through our reduction algorithm defined in Section 2.2.2 (p. 48).

6.3.4.1 Legend

We shall perform execution from a set of complete states. The execution tree from one complete state will have the syntax shown in Table 12 (p. 243).

Table 12: Execution table example

#[a]	Executor[b]	Guard[c]	State(s)[d]
1[e]			(**term**[f];Write<w>(l,v)[g]; ;)[h]
1.1[i]	MemCom.	AW	(**term**; ; ;w:BadArg[j])
1.2	MemCom	¬ AW	(First<a>[k]; ;W,a[l]:MWrite(l,v,w);)
1.2b[m]	WriteAgent	¬ AW	(Repeat<a>; ;V:MWrite<a>(l,v,w);)
1.2c	Mem	¬ AW	(Repeat<a>; ;V:**save**[n] MWrite<a>(l,v,w);)

a. State number. Every complete state in an execution has a unique number.
b. The executor is the name of the process which executes the transition
c. The guard is an expression which is an assumption for the transition. Special guards are (+) and (0) which represent alternatives in fair decisions. The guards are transformed back to decisions when the reduction is made into an SDL process.
d. The state is the complete state which is given in the syntax mostly used in this thesis.
e. This first line is the complete state from which this table represents the execution tree
f. **term** is here the name of the basic state. In this case it means “termination”. In general the name of the basic state is a tuple of basic state names of the component processes.
g. Write<w>(l,v) is an external signal. <w> designates the PId of its sender which is considered the name of a separate channel in this case. (l,v) are the symbolic parameters.
h. The general syntax of a complete state is: (basic state name; external signals; internal signals and variables; external signals).

i. When there are alternatives they are numbered by appending ".x" where the x is a natural number
j. w:BadArg means that BadArg is transmitted onto conceptual channel w (which is actually a PId)
k. The <a> is a PId of a process in a process set.
l. We also want to say both that the channel (set) has the name W and the desired process to be reached has PId a. This is denoted by both names preceding the colon (i.e. "W,a:") before the signal sequence.
m. If there are no alternative branches, the sequential execution states are numbered by appending letters starting with b.
n. **save** is described by a prefix to the signal

6.3.4.2 Executing Write

We start by the stable state reached from the initial transitions. To denote the basic state we shall use only the basic state of the appropriate WriteAgent since the other processes have only one state each. A WriteAgent which is not created or is terminated has the basic state **term**. A PId may also serve as a channel name.

Table 13: Executing Write from initial state

#	Executor	Guard	State(s)
1			(**term**;Write<w>(l,v); ;)
1.1	MemCom.	AW	(**term**; ; ;w:BadArg)
1.2	MemCom	¬ AW	(First<a>; ;W,a:MWrite(l,v,w);)
1.2b	WriteAgent	¬ AW	(Repeat<a>; ;V:MWrite<a>(l,v,w);)
1.2c	Mem	¬ AW	(Repeat<a>; ;V:**save** MWrite<a>(l,v,w);)

The complete state 1.1 is stable. The state 1.2c is semi-stable. Therefore we have finished the execution of Write.

The basic state Repeat<a> means that the WriteAgent denoted by a is in state Repeat. This WriteAgent is not involved in other communication. If another Write is input externally when the Memory is in Repeat<a> it will act exactly as shown in Table 13 (p. 244), and there will be another WriteAgent in Repeat-state. We have that the process set of WriteAgents are independent (of each other) and we can use practitioners' induction as described in Section 4.3.6.1 (p. 160) and perform a simple reduction with one WriteAgent as representative for the others. The state-vector is just present in an execution and does not enter the description as such.

We continue in Table 14 (p. 244) with executing the spontaneous consumption of the spontaneously saved MWrite signal shown in 1.2c.

Table 14: Spontaneous consumption of MWrite

#	Executor	Guard	State(s)
2		¬ AW	(Repeat<a>; ;V:**save** MWrite<a>(l,v,w);)
2.1	Mem	¬ AW, (+)	(Repeat<a>; ;V,a:MWriteOK(w), M(l)=v;)

Table 14: Spontaneous consumption of MWrite

#	Executor	Guard	State(s)
2.2	Mem	¬ AW, $(0)_1$	(Repeat<a>; ;V,a:MMemFail(w);)
2.3	Mem	¬ AW, $(0)_2$	(Repeat<a>; ;V,a:MMemFail(w), M(l)=v;)

We have no stable states and continue execution of each alternative state. We start by executing 2.1 in Table 15 (p. 245).

Table 15: Execution State 2.1

#	Executor	Guard	State(s)
2.1		¬ AW, (+)	(Repeat<a>; ;V,a:MWriteOK(w), M(l)=v;)
2.1b	WriteAgent	¬ AW, (+)	(**term**<a>; ;W:MWriteOK(w),M(l)=v;)
2.1c	MemCom.	¬ AW, (+)	(**term**<a>; ; M(l)=v; w:WriteOK)

State 2.1c is stable. We continue with 2.2 in Table 16 (p. 245).

Table 16: Execution State 2.2

#	Executor	Guard	State(s)
2.2		¬ AW, $(0)_1$	(Repeat<a>; ;V,a:MMemFail(w);)
2.2.1	WriteAgent	¬ AW,$(0)_1$, (+)	(**term**<a>; ;W:MMemFail(w);)
2.2.2	WriteAgent	¬ AW,$(0)_1$, (0)	(Repeat<a>; ;V:MWrite<a>(l,v,w);)

Here we have that 2.2.2 is equal to 1.2b and the result can be taken from Table 13 (p. 244) state 1.2c. We continue executing 2.2.1 in Table 17 (p. 245).

Table 17: Execution State 2.2.1

#	Executor	Guard	State(s)
2.2.1		¬ AW,$(0)_1$, (+)	(**term**<a>; ;W:MMemFail(w);)
2.2.1b	MemCom.	¬ AW,$(0)_1$, (+)	(**term**<a>; ; ; w:MemFail)

State 2.2.1b is stable. State 2.3 is exactly similar to 2.2 but the memory has been set.

Thus we have finished the execution of Write and the derived spontaneous save signal in all stable and semi-stable states. We summarize our execution in Table 18 (p. 245) where only the stable states are shown.

Table 18: Executing Write (summary)

#	Guard	State(s)
1		(**term**;Write<w>(l,v); ;)
1.1	AW	(**term**; ; ;w:BadArg)
1.2c	¬ AW	(Repeat<a>; ;V:**save** MWrite<a>(l,v,w);)
2	¬ AW	(Repeat<a>; ;V:**save** MWrite<a>(l,v,w);)
2.1c	¬ AW, (+)	(**term**<a>; ; M(l)=v; w:WriteOK)
2.2.1b	¬ AW,$(0)_1$, (+)	(**term**<a>; ; ; w:MemFail)
2.2.2b	¬ AW,$(0)_1$, (0)	(Repeat<a>; ;V:**save** MWrite<a>(l,v,w);)
2.3.1b	¬ AW,$(0)_2$, (+)	(**term**<a>; ; M(l)=v; w:MemFail)
2.3.2b	¬ AW,$(0)_2$, (0)	(Repeat<a>; ;V:**save** MWrite<a>(l,v,w), M(l)=v;)

It is quite simple now to bring the results of Table 18 (p. 245) back to an SDL process diagram page shown in Figure 136 (p. 247). We recall that the full reduction also includes a state vector with one entry for each WriteAgent according to Section 4.3.6.2 (p. 162). Since spontaneous consumption is impossible in (S,term) and write is impossible in (S,Repeat), it is possible to combine the two states into one and also eliminate the need for the WriteAgent state-vector.

We see in Figure 136 (p. 247) spontaneous save and spontaneous consumption. Here the spontaneous save is used with signals internal to the process which means that they actually are signals used for processes which are components of the block which has been reduced. This means that a spontaneous save not necessarily is the only construct in a transition. We see in Figure 136 (p. 247) that there is a transition where also a task is included before the spontaneous save.

Spontaneous consumption means that spontaneously saved signals are consumed. When such actions are lifted to a more global level, they appear as spontaneous while they on a more local level appear just as any other consumption of a signal.

6.3.4.3 Executing Read

Read does not have the same problems as Write since there is no "ReadAgent" which keeps trying to alter the memory over and over again. Still we have to remember the spontaneous save.

Table 19: Executing Read from initial state

#	Executor	Guard	State(s)
3			(S; Read<r>(l); ;)

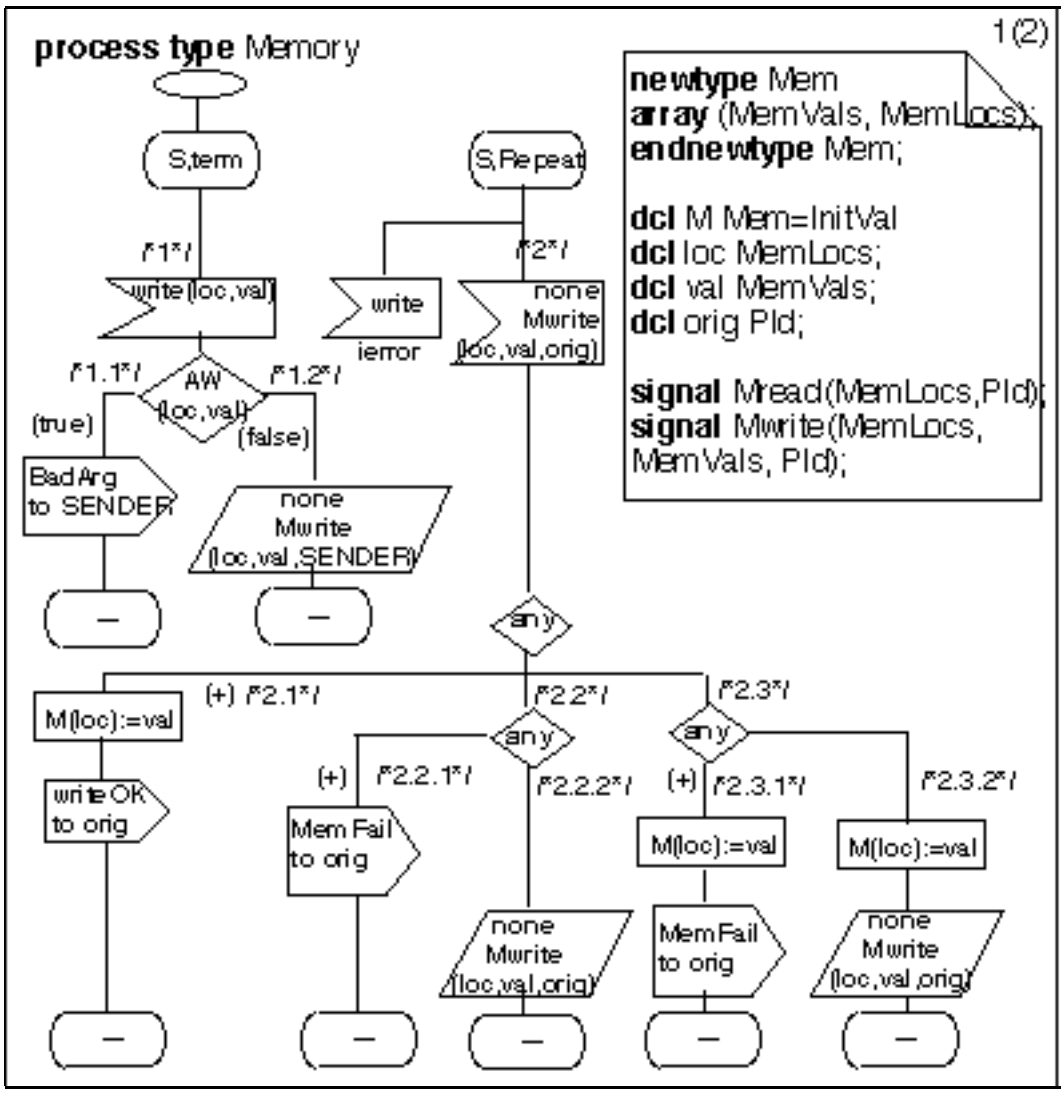


Figure 136: Memory (write) as process type

Table 19: Executing Read from initial state

#	Executor	Guard	State(s)
3.1	MemCom.	AR	(S; ; ; r:BadArg)
3.2	MemCom.	¬ AR	(S; ; R:Mread(l,r);)

State 3.1 is stable, but shall have to continue the execution of 3.2 in Table 20 (p. 247).

Table 20: Executing from 3.2

#	Executor	Guard	State(s)
3.2		¬ AR	(S; ; R:Mread(l,r);)
3.2b	Mem	¬ AR	(S; ; **save** R:Mread(l,r);)

State 3.2b is semi-stable. Now we continue to show the consumption of the spontaneously saved signal in 3.2b in Table 21 (p. 248).

Table 21: Executing Read from initial state

#	Executor	Guard	State(s)
4		¬ AR	(S; ; **save** R:Mread(l,r);)
4.1	Mem	¬ AR, $(0)_1$	(S; ; R:MreadOK(M(l),r) ;)
4.1b	MemCom.	¬ AR, $(0)_1$	(S; ; ; r:ReadOK(M(l)))
4.2	Mem	¬ AR, $(0)_2$	(S; ; R:MMemFail(r);)
4.2b	MemCom.	¬ AR, $(0)_2$	(S; ; ;r:MemFail)

Neither state 4.1 nor state 4.2 are stable, and we continued one step for each. We summarize our reduction in Table 22 (p. 248).

Table 22: Reduction of Read in Memory

#	Guard	State(s)
3		(S; Read<r>(l); ;)
3.1	AR	(S; ; ; r:BadArg)
3.2b	¬ AR	(S; ; **save** R:Mread(l,r);)
4	¬ AR	(S; ; **save** R:Mread(l,r);)
4.1b	¬ AR, $(0)_1$	(S; ; ; r:ReadOK(M(l)))
4.2b	¬ AR, $(0)_2$	(S; ; ;r:MemFail)

The results are then brought back into an SDL process diagram page in Figure 137 (p. 249).

6.3.5 Reducing ReliableMemory

The reduction of ReliableMemory will of course result in a corresponding process which does not have the alternatives to return MemFail exceptions. We show in Figure 138 (p. 250) the resulting write part of ReliableMemory.

Similar to Memory the ReliableMemory reduction includes also a state-vector for WriteAgents. We may also combine the states as indicated in Section 6.3.4 (p. 243).

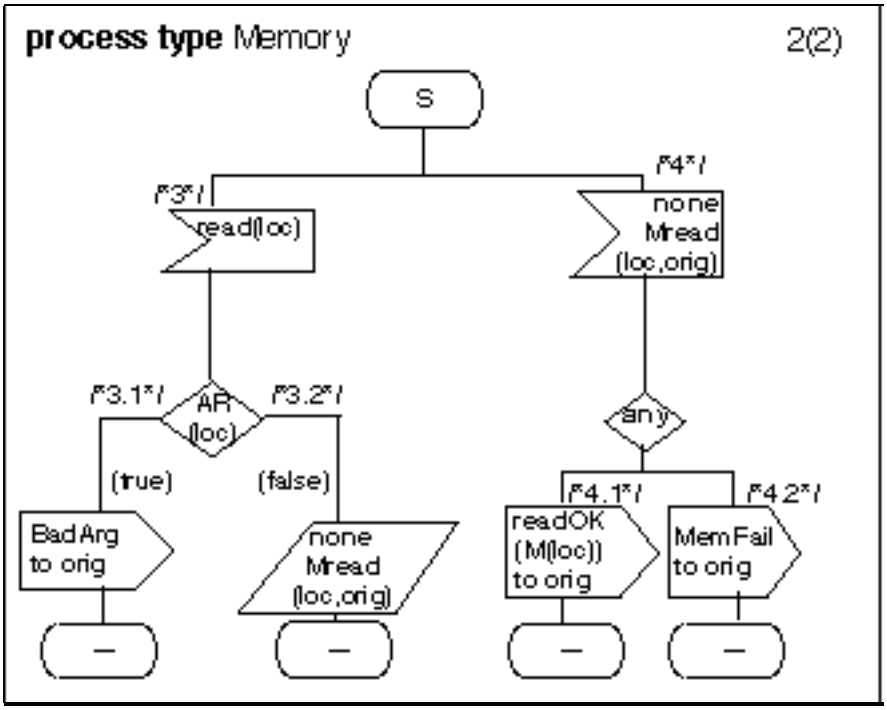


Figure 137: Memory (read) as process type

6.3.6 Comparing FailMemory *with* Memory

When we now have reached a process description of Memory in Figure 136 (p. 247) and Figure 137 (p. 249),and we can more easily compare with the direct process description of FailMemory in Figure 134 (p. 239). We must distinguish between cases relative to the values of conditions AW and AR.

Assume AW true. This means that the write signal has bad arguments and Memory will always return BadArg. FailMemory will return MemFail. Thus we have a behavior of FailMemory which cannot occur in Memory. The same holds for AR true.

We must conclude that FailMemory as depicted in Figure 134 (p. 239) is *not* a valid implementation of Memory.

6.3.7 What have we gained by reducing Memory *(*ReliableMemory*)?*

Our strategy was to describe Memory first in a way which was optimal wrt. transparency for the reader and simplicity for the designer. The design was selected by using standard engineering techniques. There were three major concerns in the problem:

1. *Concurrency problem*. The memory itself could be addressed by a number of concurrent processes. These requests should be merged fairly.
2. *Repetition problem*. The writing of a location in memory could result in an indefinite (but not infinite) number of tries.
3. *Bad arguments*. Requesting processes may simply provide requests with illegal arguments.

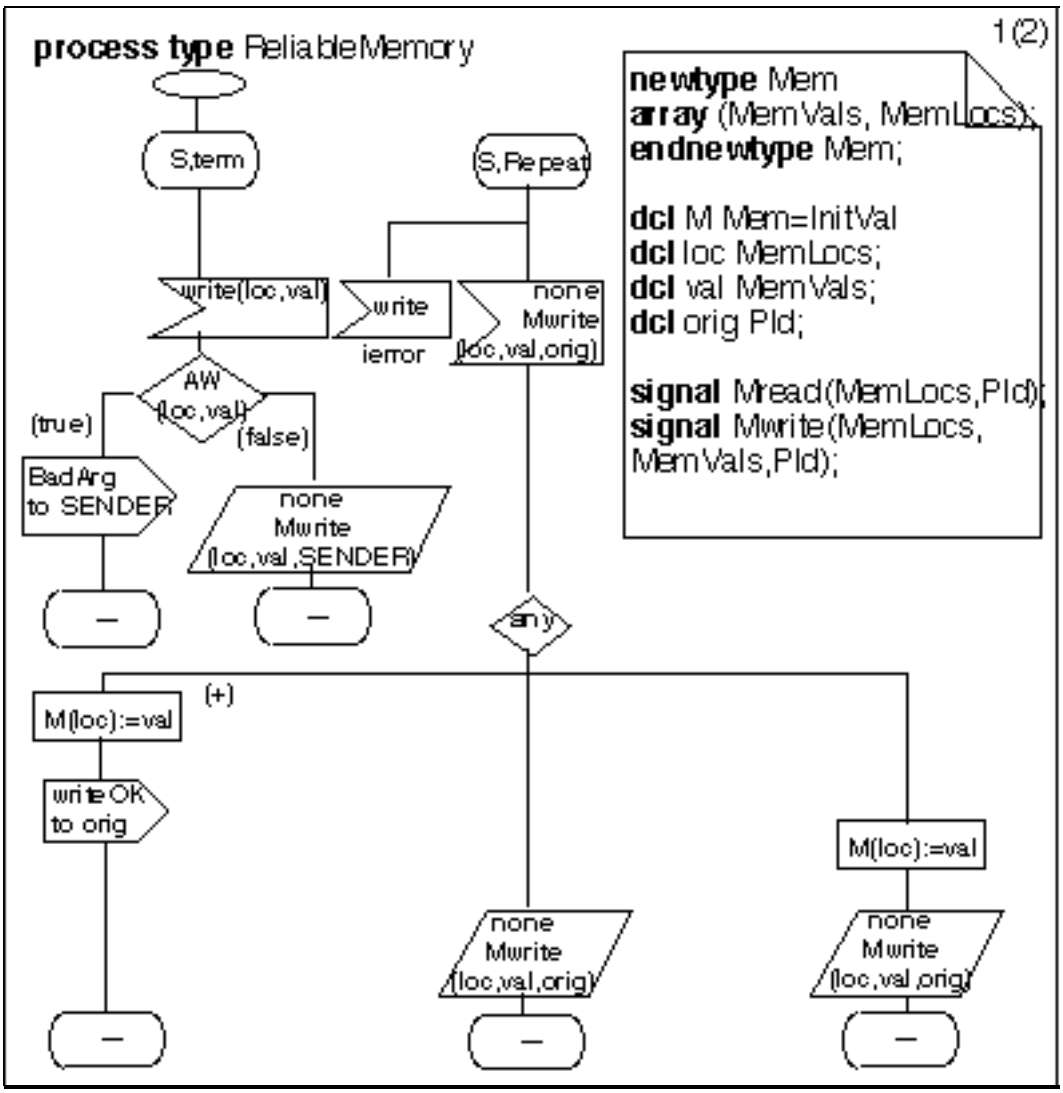


Figure 138: ReliableMemory (write) as process

Through our division of the Memory specification in three processes, each of which took care of each of the above mentioned problems, we achieved *separation of concern*. The Mem process took care of concurrency, the WriteAgents took care of repetition and MemCommHandler took care of checking for bad arguments.

From this transparent, but slightly voluminous description, we wanted a more compact and "canonical" version where the Memory block was described as an SDL process. We had to introduce a few non-SDL extensions to be able to describe the Memory as one process which externally is faithful to the block Memory. Let us summarize our tricks:

1. *Merge state and Spontaneous save.* By using a spontaneous save construction we could make Mem describe non-deterministic (fair) merge of signals in a way which could be interpreted as confluent which was a prerequisite for reducibility. Spontaneous saves of internal signals could appear inside transitions which also contained other SDL constructs like tasks.

2. *Alternatives of non-deterministic decisions with positive probability.* Fairness was assured by specifying some alternatives of decisions to have positive probabilities which means that in the long run (of a loop) these alternatives will occur and break an infinite loop.
3. *Spontaneous consumption.* As a counterpart to spontaneous save, we had to introduce spontaneous consumption which meant that seen from the outside, transitions (which consume internal signals) would start spontaneously. Locally these transitions appear as normal transitions of those signals which had been saved.
4. *Practitioners' induction*. To reduce the variability introduced by multiple instances of block (and process) sets where the instances are mutually independent, we use the "practitioners' induction" which boils down to taking only one representative for the set and also including a state-vector. We also showed that for Memory and ReliableMemory we could ad hoc eliminate the need for the state-vector as well.

After having shown progress and confluence, the reduction resulted in a description which in itself was fairly readable. Especially the resulting description of the ReliableMemory was very compact and readable. This description will be used in the sequel when ReliableMemory is to be connected to other components.

We managed to achieve a compact notation which could easily be used to show that ReliableMemory was an implementation of Memory while FailMemory was not.

As a curiosity we also note that the reduction shows that WriteAgent (defined in Figure 127 (p. 233)) is strongly progressive if the presumably impossible transitions were interpreted as saves (see Section 6.3.3.2 (p. 241)) since no signs of normal saves are present in the reduction.

6.3.8 Modifying Memory *to make* FailMemory *an implementation*

How should Memory be changed in order to make FailMemory an implementation? Also for the continuation of this exercise we may want to have a specification of Memory which contains FailMemory.

Our specification of Memory was motivated by a layered error-detecting strategy. Bad arguments would be detected first, and then storage failure resulting in MemFail.

It seems, however, that the designers of the problem have been using MemFail as a general "there is an error in my system" type of error while BadArg gives more specific information. Along these lines we can easily modify Memory by letting MemCommHandler have the non-deterministic option to return MemFail instantly after consuming write or read.

Figure 139 (p. 252) would then be the reduced Memory specification when loops and redundancies are removed. That FailMemory is a refinement of Memory follows directly from the rules for transition comparison in Section 4.2.2 (p. 149) since it is obvious that Memory can execute the transition(s) of FailMemory.

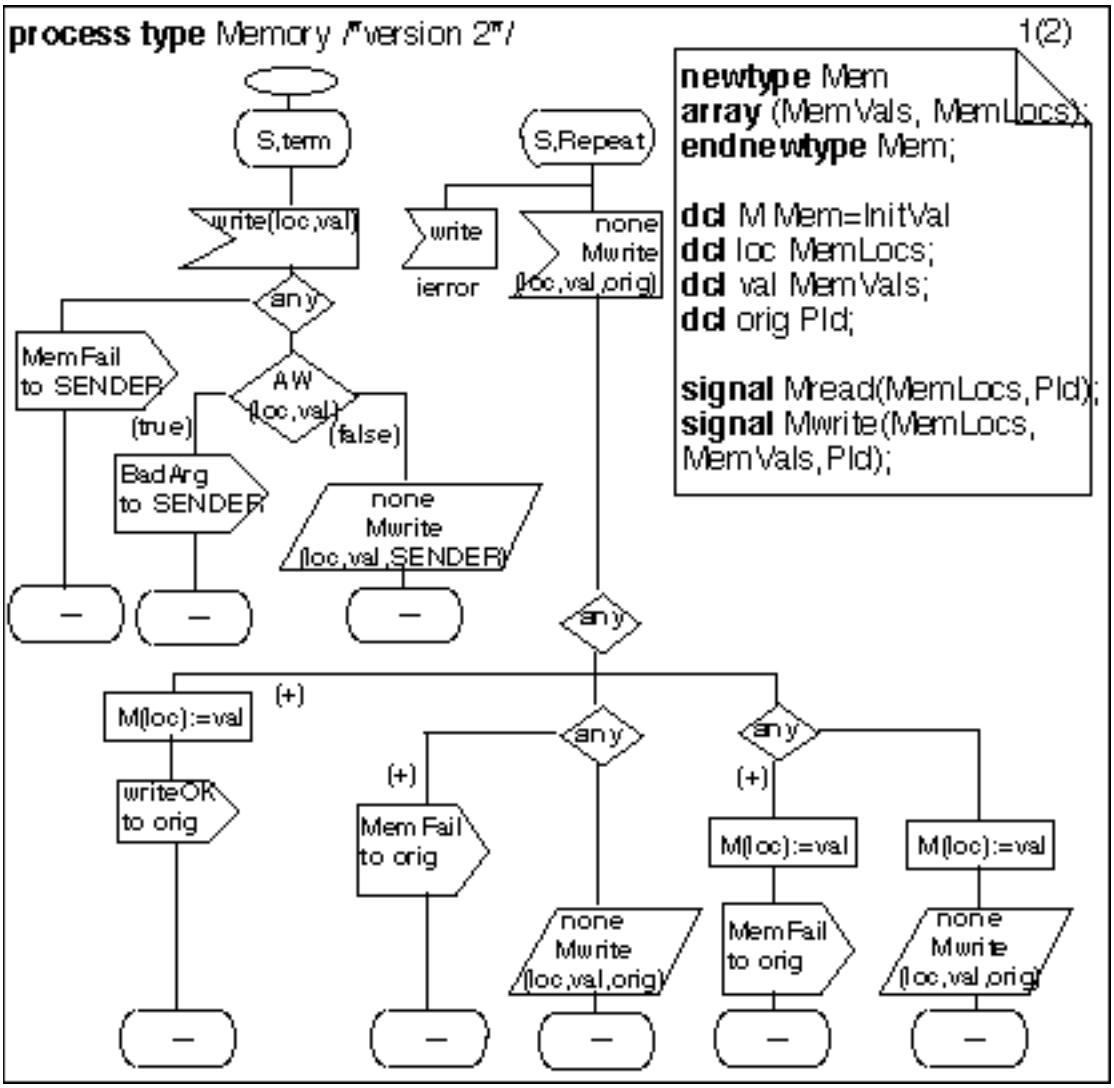


Figure 139: Memory (write) modified to accommodate immediate MemFail

6.4 The RPC *component*

We are now going to specify a component which in general relays remote procedure calls from a sender to a receiver. SDL is not well suited to specify processes of this kind of abstraction, but by allowing more extensions related to signals, we get a fairly compact description.

6.4.1 The SDL extensions

Our SDL extensions are related to the need to handle signals as data. We want to be able to handle signal objects fully as data objects meaning that they shall be possible to store in variables.

1. *Signal data type*. We define a new predefined data type which is the set of all signals. It is designated SIGNAL, such that a declaration of a signal variable will look like: "**dcl** my_sig SIGNAL;". The signal data type may then also appear as parameter to another signal. It is also possible to define variables of specific signal types by extended the syntax: "**dcl** my_sig SIGNAL mysignaltype;"
2. *Output*. It is possible to output a stored signal by outputting the variable: "**output** my_sig;" The SENDER attribute of the signal is modified to SELF of this process[1].
3. *Dash signal*. We define a predefined function (called DASHSIGNAL) which returns a SIGNAL value which is equal to the most recently consumed signal of the process.
4. *Check signal type*. We define a Boolean function which can be used to check the signal type of a signal variable: "my_sig **is** return;". A similar construct can then be used to interpret parameters: "my_sig **qua** exceptreturn(param);".
5. *Atleast input*. We may consume signals of different signal types in one transition by specifying a supertype of the signal types by "**input atleast** return;". Combination with DASHSIGNAL and signal type check makes it possible to use this effectively. In general we can use "**atleast** signaltype" also in signallists to indicate that any signal type which is inherited from signal type is allowed.

The notations for checking signal type, interpreting parameters and the atleast input are inspired by features of Simula [7].

6.4.2 Problem 2. The RPC *component environment*

The problem is to specify an RPC component. RPC is a component which "translates" a procedure call which arrives as a parameter of a higher order procedure call from the sender (RemoteCall) into a basic procedure call to the receiver. In our terms the procedure calls and their returns are modeled as asynchronous signals.

Procedure calls as parameters to a higher order procedure call will in our setting mean signals as parameters to other signals using the extended SDL notation of Section 6.4.1 (p. 252).

In Figure 140 (p. 254) we have shown the environment of the RPC component. We shall in the sequel describe RPC as a process type. The RPC process can be understood as the only instance in the RPC block type.

6.4.3 The RPC process

We describe the RPC process in Figure 141 (p. 254).

We have checked for a bad call by the Boolean expression AC. This check takes place before any other handling of the RemoteCall. Improper syntax of the RemoteCall will *always* raise a BadCall exception.

We have assumed that there is one basic RPC for each remote call communication. Concurrent remote calls must then be handled by several basic RPCs.

1. There may also be a need to keep the original SENDER of the signal. This can always be done manually as parameter to the signal, but we could also make a construct to circumvent the modification of SENDER.

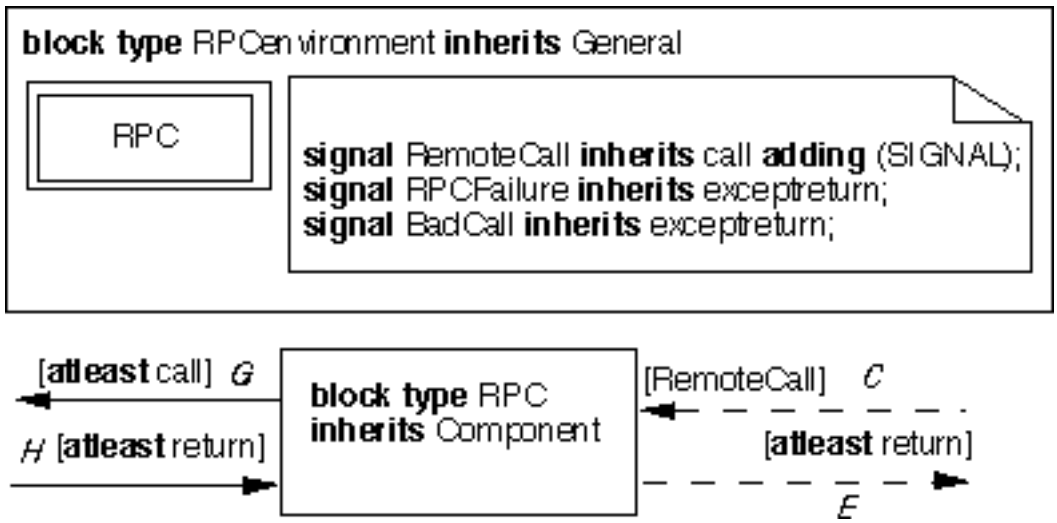


Figure 140: The RPC environment

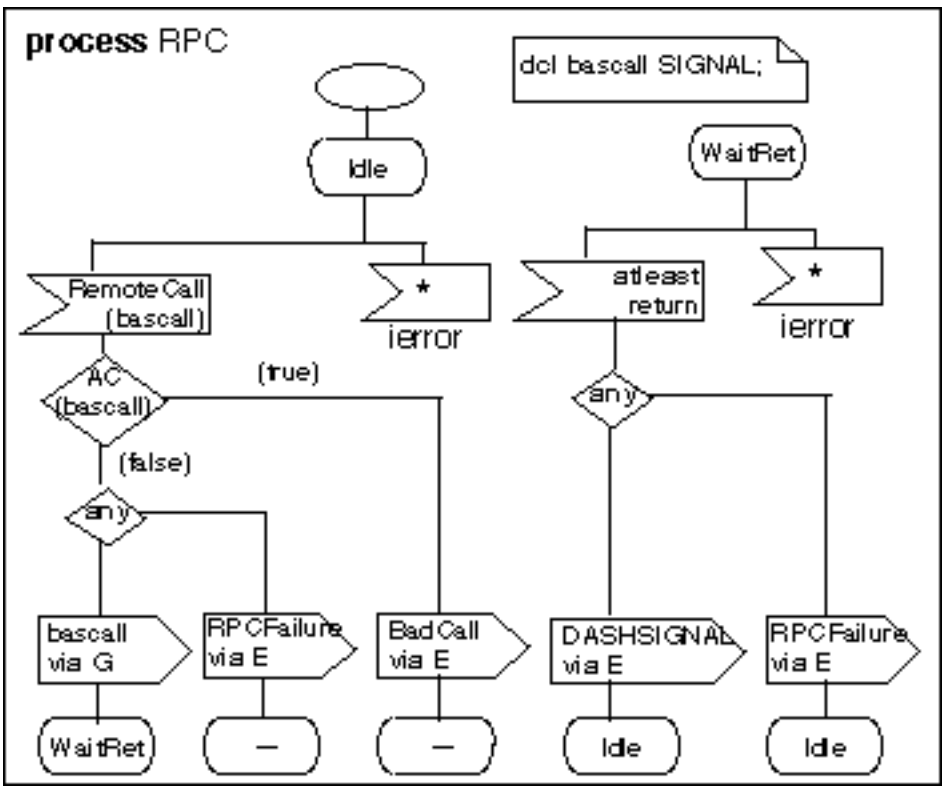


Figure 141: The RPC process

6.5 Implementing the Memory *by* RPC

We will now specify an implementation of Memory by using RPC in connection with a ReliableMemory.

6.5.1 Problem 3. Implementing Memory *by* RPC

The problem is to specify an implementation of Memory by combining the RPC component with ReliableMemory, and to prove that the resulting implementation actually *is* an implementation of Memory.

The structure of the implementation is given by Figure 142 (p. 255).

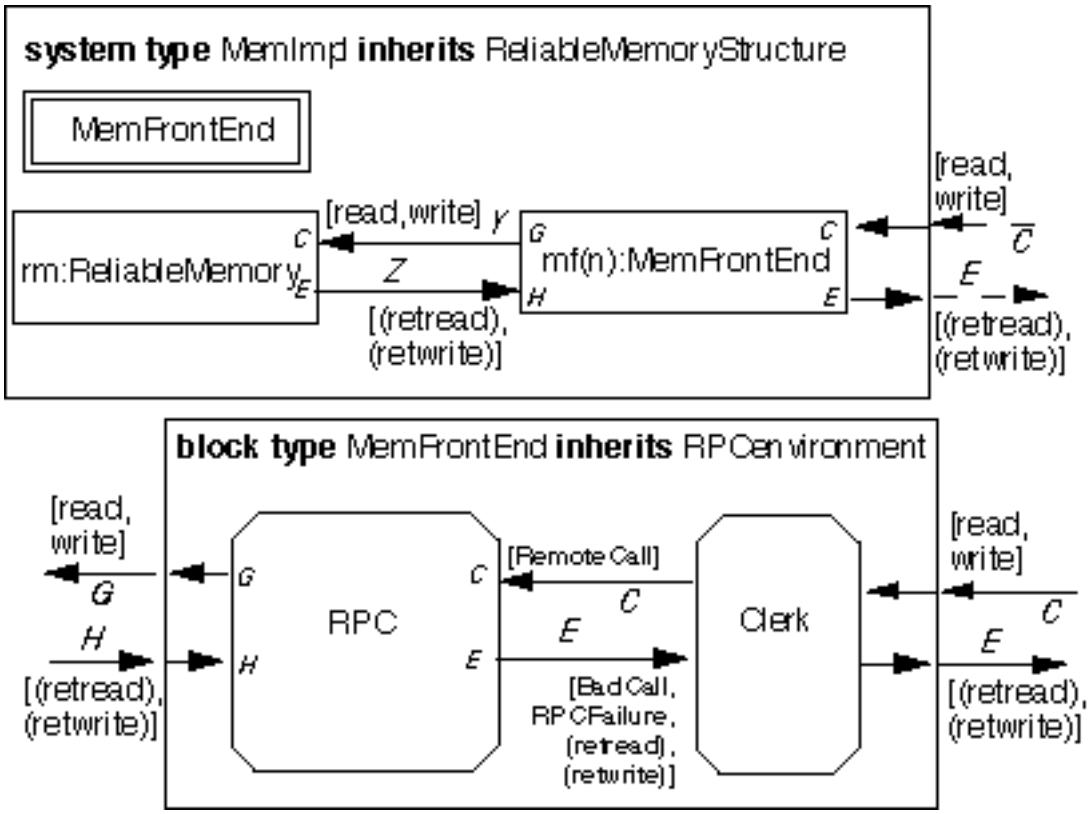


Figure 142: The Memory implementation structure

We notice that we use object-oriented inheritance on several levels to achieve very compact and transparent descriptions. We should again note, however, that the RPC type is defined as a block type, but here it is used as a process directly. This level mismatch is really no problem since the RPC process can be seen as the only entity of the block type RPC. Strictly speaking we should then had blocks in MemFrontEnd, but this extra level serves really no purpose.

To prove that MemImpl is an implementation (refinement) of Memory, we apply our general strategy which means to show that MemImpl is reducible and then compare the process version of MemImpl with that of Memory.

When we analyze MemImpl we shall first show that MemFrontEnd is reducible and perform the reduction. In the analysis of MemImpl itself we shall use the reductions of ReliableMemory and of MemFrontEnd. We recall from Section 6.3.5 (p. 248) that the reduction of ReliableMemory is composed of a simple reduction and a state-vector. We define that the multi-gates of ReliableMemory is connected to mf(n):MemFrontEnd by one MemFrontEnd to each individual gate since the different MemFrontEnd instances represent different communication initiatives on multi-gate C of MemImpl.

6.5.2 The Clerk

The Clerk is an interface between the RPC and its Memory environment, defined by Figure 143 (p. 256).

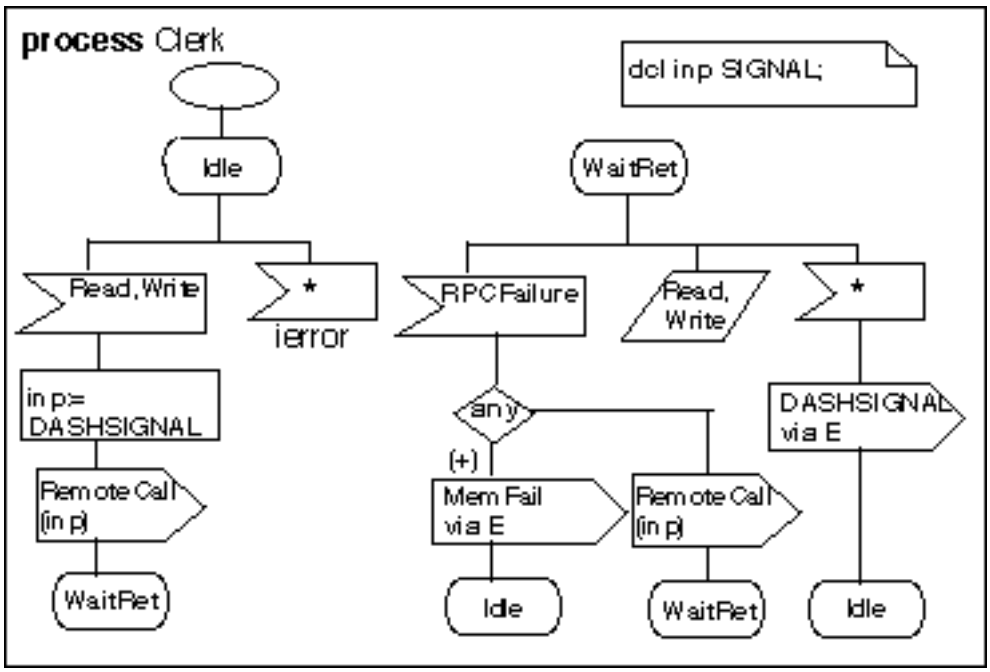


Figure 143: Process Clerk

In one direction, towards the RPC component, the basic Read and Write signals are translated and relayed by the RemoteCall. In the other direction, towards the environment, the return from the RPC is either just relayed, or if RPCFailure is received a nondeterministic decision is made to return MemFail or to try again. Sooner or later a MemFail will serve as a helpful direction and exit from the retransmit loop.

6.5.3 Progress of MemImpl

We start by trying to find a partial order of signal types such that every transition produces only signals of lower value.

In Figure 144 (p. 257) we have shown parts of the partial order which can be derived from the transitions of MemImpl. We see that the only possible cycles concern the situation when RPC returns RPCFailure and the Clerk returns the original call.

The progress through this RPCFailure cycle is simply assured by the positive probability of the alternative which returns MemFail from Clerk shown in Figure 143 (p. 256).

We conclude that MemImpl is at least weakly progressive. Strong progress may follow from the reduction if MemImpl is shown to be reducible.

6.5.4 Confluence of MemImpl

We continue to show that MemImpl is confluent by showing that none of the components can provide a non-confluence pattern.

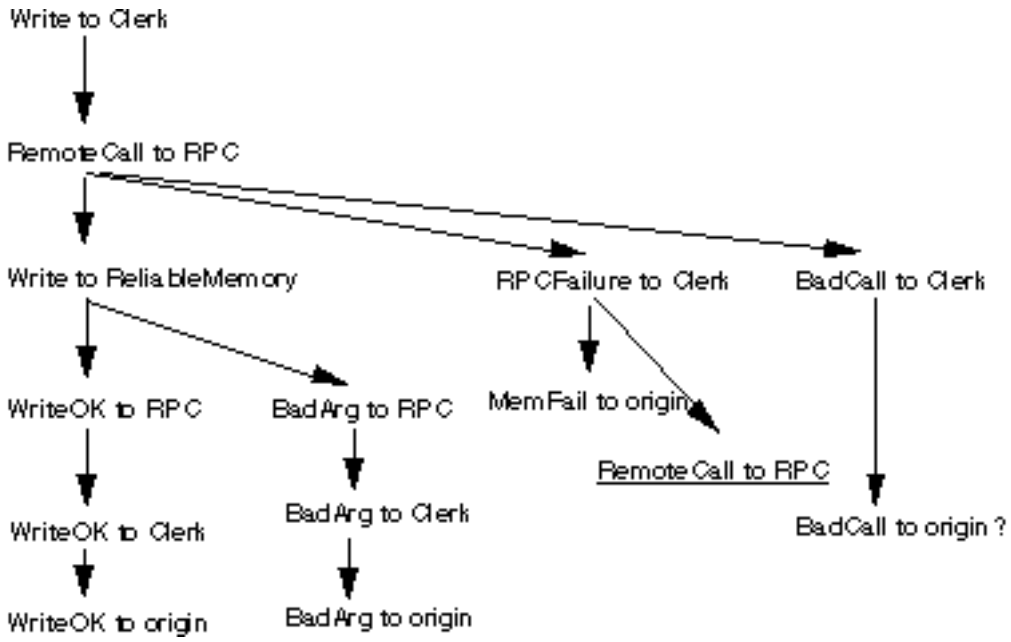


Figure 144: Parts of the partial order of signals in MemImpl

6.5.4.1 MemFrontEnd

The block type MemFrontEnd defined in Figure 142 (p. 255) is reducible on its own as we shall see that both components are channel/state-mapped.

RPC

RPC (Figure 141 (p. 254)) is a channel/state-mapped process where the state space is divided by the channels. In Idle, RPC accepts only input from gate C (i.e. from the Clerk). In WaitRet, RPC accepts only input from H (i.e. from ReliableMemory). All other possibilities are ruled illegal.

Accepting the partial reducibility where reduction is modulo internal errors, RPC is obviously confluent.

It should be possible also to prove that the signals which are internal errors in RPC are actually impossible, but we can also interpret them as saves.

Clerk

Clerk (Figure 143 (p. 256)) is also a channel/state-mapped process. In state Idle only input from C is allowed (i.e. Read and Write), while in WaitRet, preferred signals are the return signals from the ReliableMemory and RPCFailure all coming from H.

The reduced MemFrontEnd process type is given in Figure 145 (p. 258) and Figure 146 (p. 259).

The reduction is easily achieved in the same way as shown in Section 6.3.4 (p. 243). The reduction can also be seen as concatenating all the transitions involved in the execution of an external input. Internal output is not shown, and neither are states with internal signals and the consumption of internal signals.

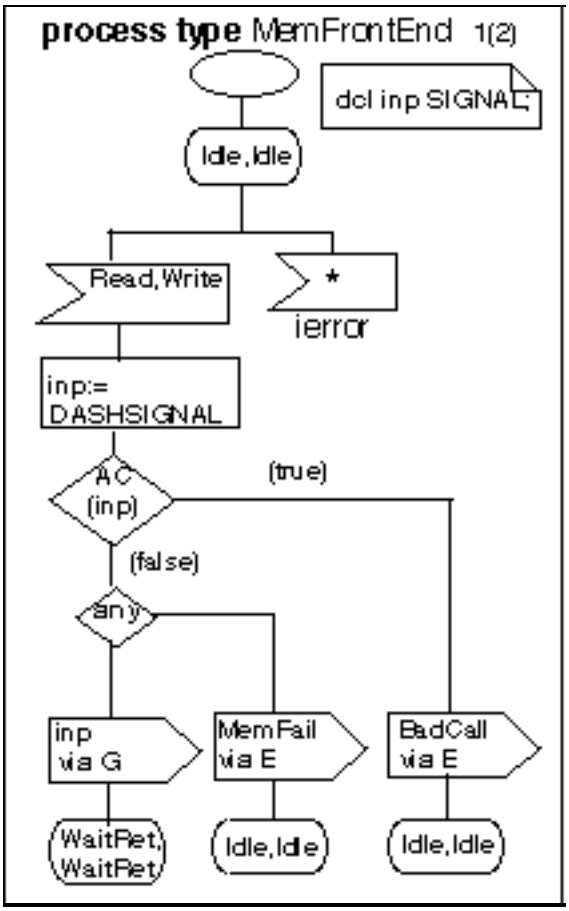


Figure 145: MemFrontEnd reduced 1(2)

6.5.4.2 ReliableMemory

Taking the definition of ReliableMemory as given in Figure 138 (p. 250) we can easily see that the spontaneous saves take good care of any potential collisions of Read and Write signals from various sources. The spontaneous save has the effect that all permutations of colliding signals are equally possible.

6.5.5 Reduction of MemImpl

We shall now execute the reduction of MemImpl by executing all external signals in all stable states. In Table 23 (p. 259) we start with the most interesting case, Write. From the same kind of argument as we had for the multiple WriteAgents in Section 6.3.4.2 (p. 244) we can conclude that also in MemImpl we can apply “practitioners’ induction” and use only one representative of the block set.

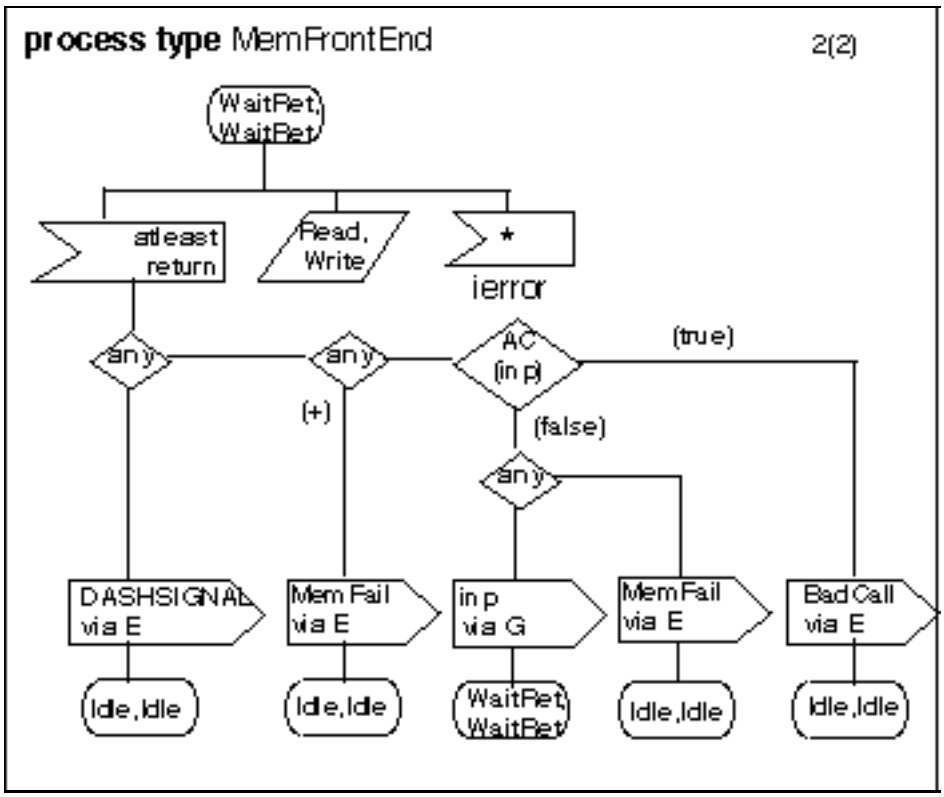


Figure 146: MemFrontEnd reduced 2(2)

6.5.5.1 Reducing MemImpl wrt. Write from Initial state

We start from the initial state (Idle,Idle,S) with a Write signal.

Table 23: Executing Write from initial state

#	Executor	Guard	State(s)
1			(Idle,Idle,S;Write(l,v); ;)
1.1	MemFront.	$\neg AC,(0)_1$.	(WaitRet,WaitRet,S; ;Y:Write(l,v), inp='Write(l,v)';)
1.2	MemFront	$\neg AC,(0)_2$.	(Idle,Idle,S; ; inp='Write(l,v)'; MemFail)
1.3	MemFront	AC	(Idle,Idle,S; ; inp='Write(l,v)'; BadCall)

While states 1.2 and 1.3 are stable, we continue the execution of 1.1 in Table 24 (p. 259).

Table 24: Executing 1.1

#	Executor	Guard	State(s)
1.1		$\neg AC,(0)_1$.	(WaitRet,WaitRet,S; ;Y:Write(l,v), inp='Write(l,v)';)

Table 24: Executing 1.1

#	Executor	Guard	State(s)
1.1.1	ReliableM.	¬AC,$(0)_1$, AW	(WaitRet,WaitRet,S; ; Z:BadArg, inp='Write(l,v)';)
1.1.2	ReliableM.	¬AC,$(0)_1$, ¬AW	(WaitRet,WaitRet,S; ; **save** MWrite(l,v,m[a]), inp='Write(l,v)';)

a. m designates the MemFrontEnd which issued the Write-call

State 1.1.2 is semi-stable. We continue to execute 1.1.1 in Table 25 (p. 260). We notice

Table 25: Executing 1.1.1

#	Executor	Guard	State(s)
1.1.1		¬AC,$(0)_1$, AW	(WaitRet,WaitRet,S; ; Z:BadArg, inp='Write(l,v)';)
1.1.1.1	MemFr.	¬AC,$(0)_1$, AW,$(0)_1$.	(Idle,Idle,S; ;inp='Write(l,v)'; BadArg)
1.1.1.2.1	MemFr.	¬AC,$(0)_1$, AW,$(0)_2$,(+)	(Idle,Idle,S; ;inp='Write(l,v)'; MemFail)
1.1.1.2.2	MemFr.	¬AC,$(0)_1$, AW,$(0)_2$,(0)	(WaitRet,WaitRet,S; ; Y:Write(l,v), inp='Write(l,v)';)

some points in this execution:

1. State 1.1.1.1 refers to normal return of the BadArg signal
2. We know that we have ¬AC from the guard, such that we know the outcome of the execution of the decision on the second branch of the decision 1.1.1.2.
3. On the branch of 1.1.1.2 which must be chosen, there is another decision between MemFail and retrying inp. We consider the MemFail option the same as state 1.1.1.2.1.
4. State 1.1.1.2.2 is the same as 1.1. This branch is therefore pruned since there is a helpful direction in 1.1.1.2.1.

We may now summarize execution of Write from Idle, Idle, S. in Table 26 (p. 260).

Table 26: Executing Write from initial state (Summary)

#	Guard	State(s)
1		(Idle,Idle,S;Write(l,v); ;)
1.1.1.1	¬AC,$(0)_1$, AW,$(0)_1$.	(Idle,Idle,S; ;inp='Write(l,v)'; BadArg)

Table 26: Executing Write from initial state (Summary)

#	Guard	State(s)
1.1.1.2.1	$\neg AC,(0)_1$, $AW,(0)_2,(+)$	(Idle,Idle,S; ;inp='Write(l,v)'; MemFail)
1.1.2	$\neg AC,(0)_1$, $\neg AW$	(WaitRet,WaitRet,S; ; **save** MWrite(l,v,m[a]), inp='Write(l,v)';)
1.2	$\neg AC,(0)_2$.	(Idle,Idle,S; ; inp='Write(l,v)'; MemFail)
1.3	AC	(Idle,Idle,S; ; inp='Write(l,v)'; BadCall)

a. m designates the MemFrontEnd which issued the Write-call. The MemFrontEnd is one-one-related to the external PId of the SENDER relative to a MemImpl process.

If we bring this back to part of an SDL process, this is shown in Figure 147 (p. 261).

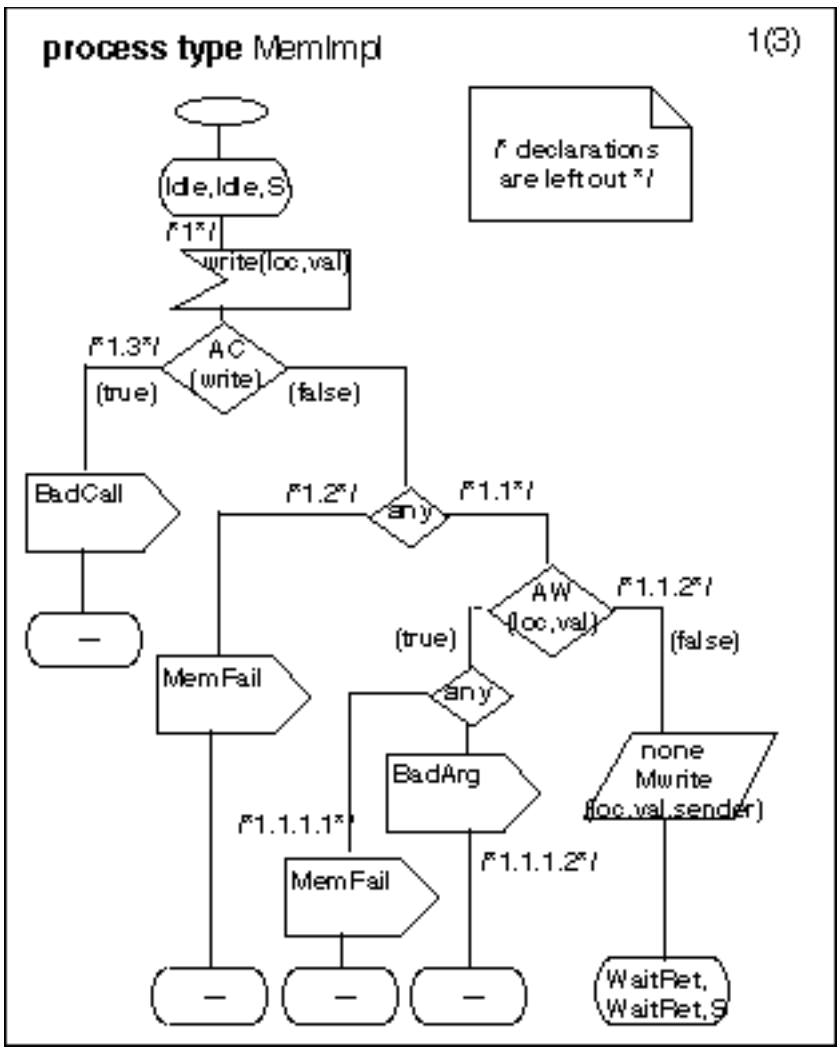


Figure 147: Process type MemImpl (reduced) Write, first part

6.5.5.2 Reducing MemImpl wrt. spontaneous consumption

We continue with the semi-stable state (WaitRet, WaitRet, S) and the consumption of the spontaneous saved MWrite in Table 27 (p. 262).

Table 27: Executing spontaneous consumption

#	Executor	Guard	State(s)
2		¬AC, ¬AW[a]	(WaitRet,WaitRet,S; ; **save** MWrite(l,v,m[b]), inp='Write(l,v)';)
2.1	ReliableM.	¬AC, ¬AW, (+)	(WaitRet,WaitRet,S; ; inp='Write(l,v)', Z: writeOK, M(l)=v;)
2.2	ReliableM.	¬AC, ¬AW, $(0)_1$.	(WaitRet,WaitRet,S; ; **save** MWrite(l,v,m), inp='Write(l,v)';)
2.3	ReliableM.	¬AC, ¬AW, $(0)_2$.	(WaitRet,WaitRet,S; ; **save** MWrite(l,v,m), inp='Write(l,v)', M(l)=v;)

a. We notice that guard which shows what condition the spontaneous save occurred.
b. m designates the MemFrontEnd which issued the Write-call. The MemFrontEnd is one-one-related to the external PId of the SENDER relative to a MemImpl process.

We see that 2.2 and 2.3 are semi-stable and very similar to state 2. The only significant difference is that 2.3 has managed to assign the memory location. The reader should not think that we should prune 2.2. (and possibly 2.3). The case is simply that the process performs a loop back to a (semi-)stable state which is quite normal in the life of an SDL process. This is different from when the reduction execution returns back to an *instable* state in the same execution. Such loops are pruned.

We continue with the instable state 2.1 in Table 28 (p. 262).

Table 28: Executing 2.1

#	Executor	Guard	State(s)
2.1		¬AC, ¬AW, (+)	(WaitRet,WaitRet,S; ; inp='Write(l,v)', Z: writeOK, M(l)=v;)
2.1.1	MemFr.	¬AC, ¬AW, (+), $(0)_1$.	(Idle,Idle,S; ;inp='Write(l,v)', M(l)=v; WriteOK)
2.1.2.1	MemFr.	¬AC, ¬AW, (+), $(0)_2$,(+)	(Idle,Idle,S; ;inp='Write(l,v)', M(l)=v; MemFail)
2.1.2.2	MemFr.	¬AC, ¬AW, (+), $(0)_2$,(0)	(WaitRet,WaitRet,S; ;Y:Write(l,v), inp='Write(l,v)', M(l)=v;)

This execution is very similar to the one in Table 25 (p. 260), but the state 2.1.2.2 cannot be pruned right away since there are no instable state in the execution of state 2 which equals it. We execute 2.1.2.2, and consider the guards, and get the execution of Table 29 (p. 263).

Table 29: Executing 2.1.2.2

#	Executor	Guard	State(s)
2.1.2.2		¬AC, ¬AW, (+), $(0)_2$,(0)	(WaitRet,WaitRet,S; ;Y:Write(l,v), inp='Write(l,v)', M(l)=v;)
2.1.2.2.b	ReliableM.	¬AC, ¬AW, (+), $(0)_2$,(0)	(WaitRet,WaitRet,S; ; **save** MWrite(l,v,m), inp='Write(l,v)', M(l)=v;)

We have now reached stable states on all execution branches in Table 30 (p. 263).

Table 30: Executing spontaneous consumption (Summary)

#	Guard	State(s)
2	¬AC, ¬AW[a]	(WaitRet,WaitRet,S; ; **save** MWrite(l,v,m[b]), inp='Write(l,v)';)
2.1.1	¬AC, ¬AW, (+), $(0)_1$.	(Idle,Idle,S; ;inp='Write(l,v)', M(l)=v; WriteOK)
2.1.2.1	¬AC, ¬AW, (+), $(0)_2$,(+)	(Idle,Idle,S; ;inp='Write(l,v)', M(l)=v; MemFail)
2.1.2.2.b	¬AC, ¬AW, (+), $(0)_2$,(0)	(WaitRet,WaitRet,S; ; **save** MWrite(l,v,m), inp='Write(l,v)', M(l)=v;)
2.2	¬AC, ¬AW, $(0)_1$.	(WaitRet,WaitRet,S; ; **save** MWrite(l,v,m), inp='Write(l,v)';)
2.3	¬AC, ¬AW, $(0)_2$.	(WaitRet,WaitRet,S; ; **save** MWrite(l,v,m), inp='Write(l,v)', M(l)=v;)

a. We notice that guard which shows under what condition the spontaneous save occurred.
b. m designates the MemFrontEnd which issued the Write-call. The MemFrontEnd is one-one-related to the external PId of the SENDER relative to a MemImpl process.

Graphically we can see this program segment in Figure 148 (p. 264).

6.5.6 Comparing MemImpl *and* Memory

Our task is to prove that MemImpl is a valid implementation of Memory meaning that any behavior of MemImpl is a behavior of Memory. We follow the principles of Section 4.2.2 (p. 149).

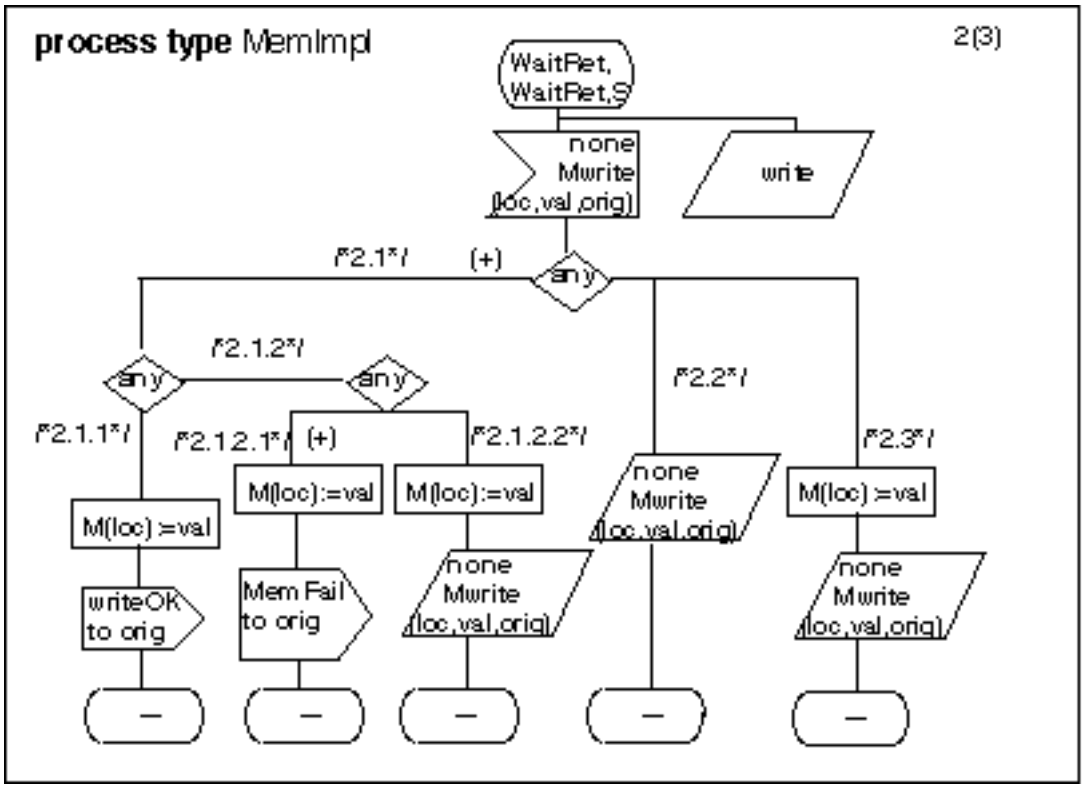


Figure 148: Process type MemImpl (reduced) Write, second part

We have the reduced Memory (the write part) in Figure 139 (p. 252) and the corresponding write part of MemImpl in Figure 147 (p. 261) and Figure 148 (p. 264). They look similar, but not absolutely identical.

1. MemImpl checks for AC, the syntax of the signal as such. This means that it checks whether the signal can be interpreted as a write or read. Memory has no such outermost syntax check.
2. When the decision on AW is reached and AW is true, MemImpl may output MemFail while Memory always outputs BadArg.

These discrepancies indicate that MemImpl may not be a valid implementation of Memory. The first point may not be serious since Memory takes for granted that only write or read signals are admitted anyway. The second discrepancy is not important since MemImpl may return MemFail already before the test on AW. Testing AW to true does not really add any new external behavior since MemFail may be returned when AW is true or false already by the first non-deterministic decision.

The comparison wrt. the spontaneous consumption of the MWrite signal shows some differences. We see that the decision structures are different, but the set of branches all together seems comparable. We do not want to make up a set of transformation rules for (fair) non-deterministic decisions, and therefore we divide the analysis here in two, considering only finite behavior and then infinite behavior.

6.5.6.1 Finite behavior

During the analysis of finite behavior we flatten the decisions. By flattening the alternatives, we mean that we oversee the fact that some alternatives may have a specified positive probability. This simplifies the decision structure of the non-deterministic decisions.

Spontaneous consumption has the option to return merely MemFail in Memory, but this option is absent in MemImpl. This discrepancy refers to the slightly different position the spontaneous save has in the two architectures. Since MemImpl does have the option to return MemFail directly when consuming the original Write signal, we consider this discrepancy insignificant.

We conclude that for finite behavior, MemImpl has the same alternatives as Memory.

6.5.6.2 Infinite behavior

Finally we should consider infinite behavior. In decisions with alternatives with positive probability we know that all those alternatives must appear when the decision occurs infinitely.

Consider the following scenario in MemImpl. If we assume an infinite input stream of write signals where AC and AW are false we may still have a return stream where "`M(loc):=val;`**`output`** `MemFail`" happens every time. This is because there is a non-deterministic decision inside the positive probability alternative. With pure non-deterministic decisions any single alternative may occur every time. This is not possible in Memory where whenever an infinite input stream of write signals where AC and AW are false occurs, the return stream must contain at least one WriteOK.

We must conclude that MemImpl is *not* an implementation of Memory.

6.5.6.3 Modifying Memory *such that* MemImpl *is an implementation*

In order to modify our original Memory such that MemImpl is a valid implementation of it, we must look into the positive probability of returning WriteOK from Mem. In Section 6.3.2 (p. 240) we argued that this ensures that any implementation of Mem must have some chance of success i.e. that WriteOK should have some positive probability to be returned. Our MemImpl may be implemented such that the memory is actually changed, but MemFail is raised anyway. It seems contrary to the general purpose of the Memory that we should have an implementation which always exhibits only half way success.

If we modify Mem such that there is no positive probability for success, Memory is still progressive since there is a positive probability of relaying MemFail in WriteAgent. In ReliableMemory we must still have the positive probability to return success in Mem since WriteAgent cannot relay the MemFail.

These changes will result in the reduced Memory shown in Figure 149 (p. 266).

We can now find no infinite behavior which is possible in Memory, but not in MemImpl.

We conclude that MemImpl is a valid implementations of Memory.

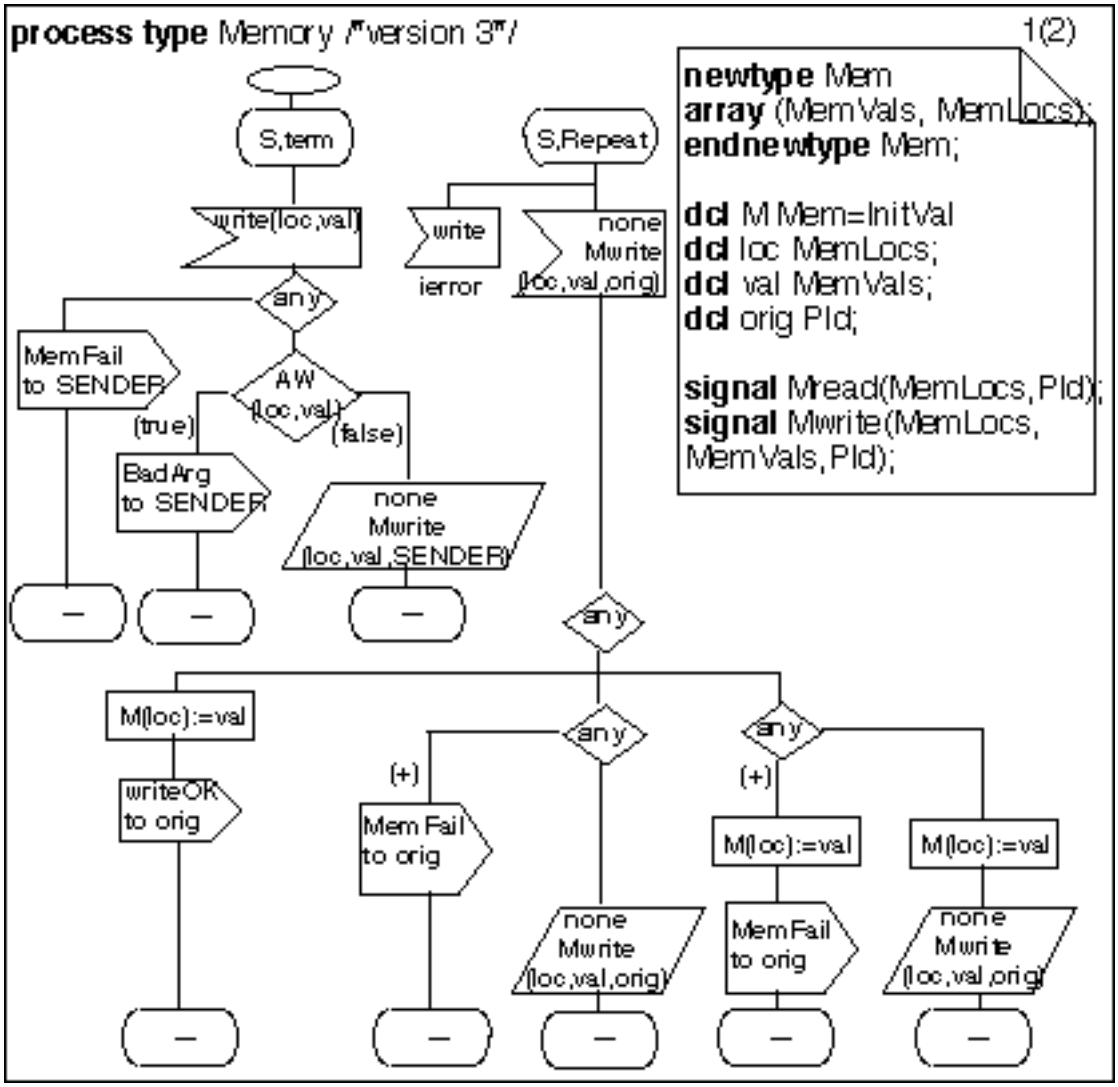


Figure 149: Final specification of Memory **(write)**

6.6 Implementing RPC

The last part of the RPC-Memory specification problem is about the implementation of RPC via using a lossy RPC component which sometimes just does not return, but when it returns it does so within a given duration limit.

We have already recognized that SDL is not suitable to specify time and duration constraints as pointed out in Section 1.6.3 (p. 35), and our thesis has not made any serious attempt to improve these aspects of SDL.

Therefore we have found no reason to give the solution to the last part of the RPC-Memory specification problem in this thesis.

6.7 Conclusions

The RPC-Memory specification problem has shown more facets of specification and verification than one would expect at first glance. The problem itself is fairly simple, but complexities are hidden in the fairness requirements and the implementations.

The example problem has shown that SDL is fairly well suited for such a problem of real time engineering, but that there are specific aspects which need notation extensions and more elaborated verification techniques.

SDL has problems when it comes to:

1. Describing fairness.
2. Generalizing wrt. signal types. Signals cannot be considered data in standard SDL.

With our extensions, these areas were also possible to handle.

In our solution to the problems we have shown that a one-way top-down approach from the most general specification to the implementations is not adequate. We show that our engineering approach turns out to conclude that the proposed implementations are not actual refinements. When we decide to make them refinements, the original general specifications had to be iterated and changed. By doing so we had to go through arguments for the desirability of the design chosen.

We also found that the exercise was very illustrative for using our practitioners' verification approach on a practical/theoretical problem. We feel that our descriptions and transformations had two very desirable properties:

1. The descriptions were readable.
2. The transformations were transparent and/or automatic. The proofs could probably have been done by any engineer.

6 The RPC-Memory Specification Problem

Conclusions

7 Conclusions and further work

Omniscience

Knowing what
thou knowest not
is in a sense
omniscience

7. Conclusions and further work

We have in this thesis presented an approach to systems engineering based on a simple validation strategy. What are the strongholds of the Mn-approach and what are its shortcomings? How should further research and development improve the approach?

7.1 Recapitulation

The Mn-approach is based on a procedure (the Mn-procedure) for determining confluence of a concurrent system of communicating extended finite state machines. From a very simple idea and a very simple basic strategy – to check all possible race conditions – we were able to prove interesting features of interesting, theoretical systems.

We let the Mn-approach inspire a method – confluent design – which should be possible to use favorably on real systems. We reported from a rudimentary industrial case study. We provided arguments for the applicability of the Mn-approach from our own experience as a long time engineer of reactive systems.

The Mn-approach does not cover everything. There is no simple way to determine exactly in which situations the Mn-approach adds valuable insight. In Figure 150 (p. 270) we have given an overview of the domains of the Mn-approach.

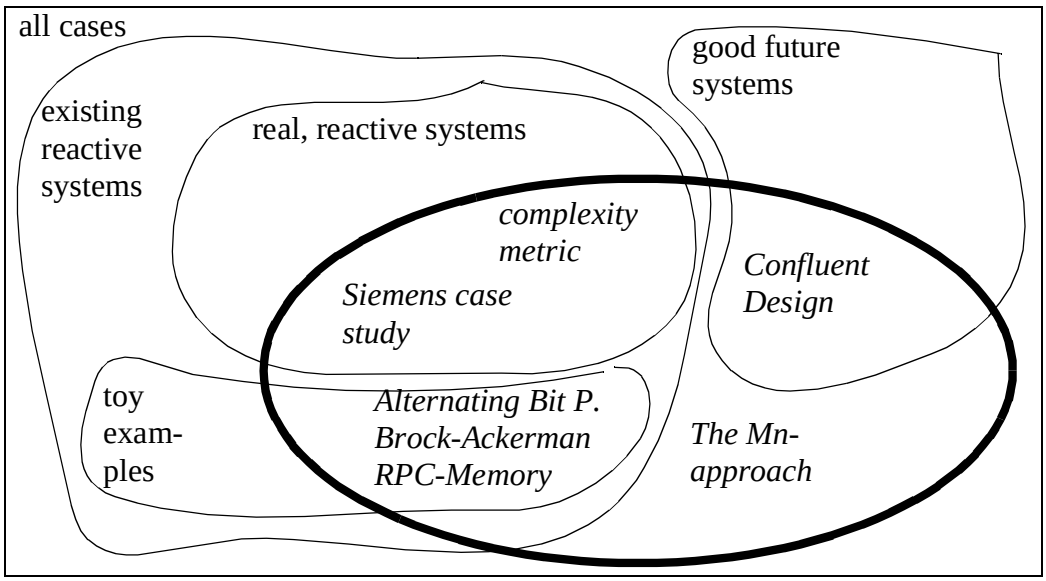


Figure 150: The Scope of the Mn-approach

7.2 The strongholds of the Mn-approach

The strong points of the Mn-approach seem to be its smooth transition from the practical engineering to the theoretical world of automata, and its friendliness towards other approaches. It is also reasonable to believe that proofs by the Mn-approach are more readable and comprehensible than formal proofs, at least for practitioners.

7.2.1 A bridge from theory to practice

On one side of the spectrum there is the practitioner who has great skepticism towards theory. On the other end there is the theorist who could not care less about the practical application of his ideas, but he wants a well founded way to visualize certain aspects of a concurrent system. The Mn-approach may have something for both these two extremes and also for persons in between these extremes. We have tried to illustrate the different aspects of the Mn-approach on such a spectrum in Figure 151 (p. 270). The the-

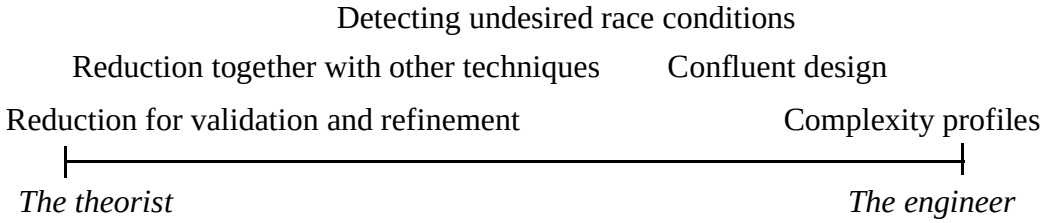


Figure 151: The Mn-bridge between theorists and engineers

orist may want to use the Mn-procedure to make a reduction of a system to explore its external properties. We have also seen that reductions can be used in connection with refinement verification.

Slightly less ambitious is to select certain parts of the system which are specially suited for reduction and reduce these. These reductions are then used together with other methods like reachability algorithms.

Even less ambitious is to use the Mn-procedure just to indicate where there are race conditions which may be problematic. We may not even go all the way to ascertain that there is a real problem before we present the indication to the engineer. He may then get an “aha-experience” and admit that there may be a problem.

To keep to the confluent design guidelines is reasonable even if the Mn-approach will not be used formally to validate the system. And finally to apply the complexity profiles based on the Mn-approach may indicate problematic areas without any direct infringement on the system development as such.

7.2.2 The Mn-approach is the friendly approach

Since the Mn-approach is a monolithic approach to validation which only reduces the complexity of a subsystem by eliminating the internal communication, it can be easily combined with other methods. After having applied the reduction, the resulting system is still a system of communicating finite state machines, and theoretically eligible to the same approaches as the original. This means that any technique which could have been used on the original, can also be used when a part has been reduced via the Mn-procedure.

The question is what the analysis wants to conclude. It is obvious that the aim of the analysis must be expressible in terms of the transformed system. That the analysis aim is expressible in terms of the transformed system must also mean that all examples or counterexamples of the aim must be expressible in terms of the transformed system. If this holds for some property, we claim that the property holds also for the original system.

7.2.3 The Mn-proofs are transparent

Each step of the Mn-procedure is an execution which is fairly straight forward, and it may often be performed automatically as a symbolic simulation. The reduction algorithm is a series of symbolic executions, too. The result of the reduction is a process graph which is in (slightly extended) SDL.

There is no notation which is not known to the engineer and there are no inference rules (verification steps) which are not executions.

The Mn-approach to validation is an imperative approach using the concepts and operations which are already known by the engineers.

7.3 Points on which the Mn-approach could be improved

The Mn-approach is not the answer to everything. There are areas where the Mn-approach is not well suited, and there are areas where more research should improve the technique.

The major issue where the Mn-approach has little to offer is in handling data. We have also decided not to spend effort on proving progress.

Furthermore the Mn-approach is, similar to SDL itself, not perfectly suited to reason about real time. We also believe that some people will argue that our approach to fairness is not fully adequate.

Finally we have in the thesis used other techniques to support the Mn-approach, such as backwards execution. There should be advantageous to look into a more systematic use of other techniques in connection with the Mn-approach.

7.3.1 Where the Mn-approach has little to offer

There are two important subjects which we have treated only ad hoc in this thesis. These subjects are data and progress.

7.3.1.1 Data

Concerning data we have said that our major aim is to eliminate internal communication and data is not so important then. This is of course only partly true. If the aim is to make complete reduction which can be used as integral part in other methods, the data must be included.

We have done little or nothing to evaluate whether the data used in real systems are actually suitable for symbolic execution. We know theoretically that data may exhibit all the complexity of the world, but our conjecture has been that reactive systems seldom contain unmanageable data complexity. From experience we are convinced that certain parts of systems have very simple data, while others are complicated.

7.3.1.2 Progress

Progress is a prerequisite for the Mn-procedure, but we have treated this subject in a very ad hoc manner. Still we believe that in reactive systems, progress is often not the worst thing to establish. On the other hand in a real system the structure of feedback loops with retransmissions and acknowledgments etc. may be quite difficult to follow manually. This is definitely an area for further study both empirically and theoretically.

7.3.2 Where the Mn-approach may not be perfectly suited

We have reason to believe that theorists are not perfectly happy with our treatment of real time and fairness.

7.3.2.1 Real Time

We can handle timers with the Mn-procedure. We are aware that confluence is a much more intricate concept when time and duration have to be taken into account and not only the order of events.

It is definitely a subject for further research to determine how and whether the reduction technique of the Mn-approach can be extended to cope with time and duration in a more general way than we have shown in this thesis.

7.3.2.2 Fairness

We have included in the Mn-approach a notation for fairness in connection with decisions. Our notion of fairness is described as extreme fairness or a variant of probabilistic fairness. This notion seems to fit well in an imperative setting where each language construct is considered to execute independent of all other imperatives. Thus when a nondeterministic decision executes, it has no “knowledge” of earlier or later executions of this or other decisions. With our extreme fairness, we assume that certain outcomes of the decision (helpful directions) have positive probability to appear. The positive probability is present *every* time the decision is executed.

There are many other notions of fairness which have proved to be practical. We do not eliminate the possibility that other fairness concepts than our extreme fairness may be suitable to include in communicating finite state machines, and in the language SDL. This is a matter of future research.

7.3.3 Where the Mn-approach could be helped by other techniques

The main risk when applying the Mn-procedure is that unreachable non-confluence patterns are found and absolute confluence cannot be proven. We have used simple backwards execution as an ad hoc way to prove that a given state is not reachable.

We are certain that improved coverage of the Mn-approach could be obtained if we applied in a systematic way supporting techniques to eliminate non-confluence patterns which are unreachable. Typically proven state invariants could serve as a filter to define a subset of the full set of complete states.

This is a matter for future research.

7.4 Empirical data and tools

Our rudimentary industrial case study can hardly be considered a thorough empirical study of the applicability of the Mn-approach. It would have been feasible to perform more studies along the same line, but it became evident that automatic support of the Mn-procedure is almost a prerequisite.

Our prototype tool could not do the job as merely to make the system available to the tool would take many hours and be fairly error prone. Furthermore the prototype tool could not handle all the features needed.

The rise in effort from indicating problematic race conditions to proving reducibility is substantial, and we decided to spend our available time on smaller toy examples where we could reveal theoretical problems of the approach.

More empirical studies are needed to support the conjecture that the Mn-approach is actually applicable in the real life on real systems. In order to perform such studies in a reliable way tools are needed. It should be fairly simple to modify the SDL tools available to accommodate the Mn-approach, and we hope that the tool vendors will find the Mn-approach interesting and consider investing some resources into making an Mn-module to their tools.

Pilot studies where aspects of the “confluent design” approach is used should supplement studying the effect of using Mn-approach as validation approach to already existing systems.

The reader should remember that the engineer is the most valuable resource of validation. He has the power to make things simple and verifiable, or – to make them complex and impossible to verify.

8 References

8. References

1. Apt, K. R., Pnueli, A. and Stavi, J. (1984). “Fair termination reivisited with delay.” *TCS* **33** 65-84.
2. Barringer, H. (1985). *A Survey of Verification Techniques for Parallel Programs*. Berlin, Springer-Verlag LNCS 191 3-540-15239-3.
3. Bartlett, K. A., Scatlebury, R. A. and Wilkinson, P. T. (1969). “A Note on Reliable Full-Duplex Transmission over Half-Duplex Links.” *Communications of the ACM* **12**(5) 260-265.
4. Basin, D. A. (1996). Verification Based on Monadic Logic. BRICS, Univ. of Aarhus BRICS Notes Series NS-96-3, Aarhus
5. Belina, F., Hogrefe, D. and Sarma, A. (1991). *SDL with Applications from Protocol Specification*. Hemel Hempstead:, Prentice Hall
6. Bengtsson, J., Griffioen, D., Dristoffersen, K., Larsen, K. G., Larsson, F., Pettersson, P., et al. (1996). *Verification of an Audio Protocol with Bus Collision Using UPPALL*. 8th International Conference on Computer Aided Verfication, Pages: 244-256, Springer-Verlag LNCS 1102
7. Birtwistle, G. M., Dahl, O.-J., Myhrhaug, B. and Nygaard, K. (1975). *SIMULA BEGIN*. New York, Petrocelli/Charter
8. Boehm, B. W. (1981). *Software Engineering Economics*. Englewood Cliffs:, Prentice Hall
9. Bolognesi, T. and Brinksma, E. (1987). “Introduction to the ISO Specification Language LOTOS.” *computer Networks and ISDN Systems* **14** 25-59.
10. Bowen, J. and Hinchey, M. G. (1995). *Applications of Formal Methods*. Prentice Hall 0-13-366949-1.
11. Bræk, R. and Haugen, Ø. (1993). *Engineering Real Time Systems*. Hemel Hempstead, Prentice Hall International 0-13-034448-6.

8 References

12. Bræk, R., Haugen, O., Melby, G., Møller-Pedersen, B., Sanders, R. and Stålhane, T. (1996). Integrated Methodology. Oslo, SISU.

13. Brand, D. and Zafiropulo, P. (1983). "On communicating Finite-State Machines." *Journal of the ACM* **30**(2) 323-342.

14. Brock, J. D. and Ackerman, W. B. (1981). Scenarios: a model of non-determinate computation. *Formalization of Programming Concepts* Eds. Diaz and Ramos. Berlin, Springer-Verlag. LNCS 107 252-259.

15. Broy, Dederichs, Dendorfer, Fuchs, Gritzner and Weber (1993). The design of Distributed Systems. An Introduction to FOCUS. Technische Universität München München

16. Broy, M. (1987). "Semantics of Finite and Infinite Networks of Concurrent Communicating Agents." *Distributed Computing* **2**(1) 13-31.

17. Broy, M. (1991). "Towards a Formal Foundation of the Specification and Description Language SDL." *Formal Aspects of Computing* **3**(1) 21-57.

18. Broy, M. (1993). (Inter-)Action Refinement: The Easy Way. *Program Design Calculi* Ed. M. Broy. Springer. NATO ASI Series, Series F: Computer and System Sciences 118

19. Broy, M. and Stølen, K. (1994). *Specification and Refinement of Finite Dataflow Networks - a Relational Approach.* FTRTFT'94, Pages: 247-267, Springer-Verlag 863

20. Broy, M. and Stølen, K. (1996). *FOCUS on System Development*. Munich, (manuscript)

21. Bruns, G. (1993). *A Practical Technique for Process Abstraction*. CONCUR'93, 4th International Conference on Concurrentcy Theory, Hildesheim, Germany August 1993, Pages: 37-49, Springer-Verlag LNCS 715 3-540-57208-2.

22. Burstall, R. M. and Darlington, J. (1977). "A Transformation System for Developing Recursive Programs." *Journal of the ACM* **24**(1) 44-67.

23. Carroll, J. and Long, D. (1989). *Theory of finite automata with an introduction to formal languages*. London, Prentice-Hall International, Inc. 0-13-913815-3.

24. Cavalli, A. R., Chin, B.-M. and Chon, K. (1996). "Testing methods for SDL systems." *CN&ISDN* (June 1996) 1669-1684.

25. CCITT (1988) Z.100 CCITT Specification and Description Language, ITU,

26. CCITT (1988) Z.100 Annex D SDL User Guidelines, ITU,

27. Clarke, E. M., Emerson, E. A. and Sistla, A. P. (1983). *Automatic verification of finite state concurrent systems using temporal logic specifications*. 10th ACM Symposium on Principles of Programming Languages, Austin, Texas January 24-26, 1983, Pages: 117-126, ACM

28. Clarke, E. M. J. (1996). Symbolic Model Checking. BRICS, Univ. of Aarhus BRICS Notes Series NS-96-4, Aarhus

29. Coad, P., Yourdon, E. (1990). *Object-Oriented Analysis*. Englewood Cliffs, Prentice Hall

30. Dahl, O.-J. (1992). *Verifiable Programming*. Hemel Hempstead, UK, Prentice Hall International 0-13-951062-1.

31. Dahl, O.-J., Dijkstra, E. W. and Hoare, C. A. R. (1972). *Structured Programming*. London and New York, Academic Press

32. Dahl, O.-J. and Hoare, C. A. R. (1972). Hierarchical Program Structures. *Structured Programming* Eds. O.-J. Dahl, E. W. Dijkstra and C. A. R. Hoare. London and New York, Academic Press.

33. Darlington, J. and Burstall, R. M. (1976). "A System which Automatically Improves Programs." *Acta informatica* **6** 41-60.

34. de Roever, W.-P. (1992). Why Formal Methods is a Must for Real-Time System Specification. Oslo, University of Oslo.

35. de Roever, W.-P., Coenen, J., Buth, K.-H., Engelhardt, K., Lakhneche, Y. and Stomp, F. (1993). State-based Formalisms for Data Refinement.

36. DeMarco, T. (1979). *Structured Analysis and Systems Specification.* Englewood Cliffs, Prentice Hall

37. Dershowitz, N. and Jouannaud, J.-P. (1990). Rewrite Systems. *Handbook of Theoretical Computer Science. Formal Models and Semantics* Ed. J. van Leeuwen. Amsterdam, Elsevier. Vol. B 243-320.

38. Dietz, C. (1994). Duration Calculus Specifications of Shared Registers. University of Oldenburg Draft, ProCos, copy by O-J. Dahl Oldenburg

39. Dyer, M. (1992). *The Cleanroom Approach to Quality Software Engineering.* John Wiley & Sons

40. Ek, A. (1993). *Verifying Message Sequence Charts with the SDT Validator.* SDL '93 Using Objects. Proceedings of the Sixth SDL Forum, Darmstadt, Germany October 12th – 16th 1993, Pages: 237-249, North Holland 0-444-81486-8.

41. Ellis and Stroustrup, B. (1990). *The Annotated C++ Reference Manual*. Addison-Wesley

42. Ellsberger, J., Hogrefe, D. and Sarma, A. (1997). *SDL. Formal Object-oriented Language for Communicating Systems.* London, Prentice Hall Europe 0-13-621384-7.

43. Emerson, E. A. (1996). Automated Temporal reasoning about Reactive Systems. *Logics for Concurrency. Structure versus Automata.* Eds. F. Moller and G. Birtwistle. Berlin, Springer-Verlag. Lecture Notes in Computer Science LNCS 1043 41-101.

44. ETSI (1994), "Methods for Testing and Specification (MTS); Methodologies for Standards Engineering – Specification of Protocols and Service", WD 1 in Q6 on Methodology at ITU-TS Geneva Oct. 19.-27. 1994

45. Færgemand, O. and M.M., M., Eds. (1989). *SDL´89 The Language at Work. Proceedings of the Fourth SDL Forum, Lisbon, October 1989.* Lisbon, North Holland: Elsevier

46. Færgemand, O. and Reed, R., Eds. (1991). *SDL´91 Evolving Methods. Proceedings of the Fifth SDL Forum, Glasgow, October 1991*. Glasgow, North Holland: Elsevier

47. Færgemand, O. and Sarma, A., Eds. (1993). *SDL '93. Using Objects. SDL Forum 1993*. Darmstadt, North Holland. Elsevier 0-444-81486-8.

48. Finkel, A. (1988). *A new class of analyzable CFSMs with unbounded FIFO channels*. Protocol Specification Testing and Verification VIII, Pages: 283-294, Elsevier Science Publishers B.V., North Holland VIII

49. Francez, N. (1986). *Fairness*. New York, Springer-Verlag

8 References

50. Freedman, D. P. and Weinberg, G. M. (1982). *Handbook of Walkthroughs, Inspections, and Technical Reviews. Evaluating Program, Projects and Products*. Boston, Little, Brown and Company

51. Gilb, T. and Graham, D. (1993). *Software Inspection*. Wokingham, Addison-Wesley 0-201-63181-4.

52. Grabowski, J. (1994). *Test Case Generation and Test Case Specification with Message Sequence Charts*. Inauguraldissertation, Institut für Informatik und angewandte Mathematik, Universität Bern

53. Grossman, R. L., Nerode, A., Ravn, A. P. and Rischel, H., Eds. (1993). *Hybrid Systems*. Springer Verlag LNCS 736

54. Harel, D. (1987). “Statecharts: A visual formalism for complex systems.” *Scientific Computing Programming* **8**(3) 231 – 274.

55. Hauge, T. and Haugen, Ø. (1989). *OST – An Object-oriented SDL Tool*. Forth SDL Forum, Lisbon, Portugal 9. - 13. October 1989,

56. Haugen, O. (1980). *Hierarchies in Programming and System Description*. Master Thesis, University of Oslo

57. Haugen, O. (1996). “Special issue on SDL and MSC.” *CN&ISDN* (June 1996) 1581-1717.

58. Haugen, O. (1997). *The MSC-96 Distillery.* to be published at SDL Forum 97, Paris, France Sept. 1997, North-Holland

59. Haugen, Ø. (1994). MSC Methodology. SISU Report L-1313-7, Oslo

60. Haugen, Ø. (1995). On the advanced use of MSC. SISU/Siemens Notat L-2103-HAU1, Oslo

61. Haugen, Ø. (1995). *Using MSC-92 Effectively.* SDL'95 with MSC in CASE. Proceedings of the Seventh SDL Forum, Oslo, Norway 26.-29. Sept. 1995, North-Holland, Elsevier

62. Haugen, Ø., Bræk, R. and Melby, G. (1993). *The SISU project*. SDL '93 Using Objects. Proceedings of the Sixth SDL Forum, Darmstadt, Germany October 12th – 16th 1993, Pages: 479-489, North Holland 0-444-81486-8.

63. Haugen, Ø., Stenhaug, U., Trettenes, N., Åsen, H. K. and Jarfonn, D. (1994). Walkthrough as a means for V&V. SISU Report L-1313-5, Oslo

64. Hayes, I. e. (1987). *Specification Case Studies*. London, Prentice-Hall International 0-13-826579-8.

65. He, J., Hoare, C. A. R., Fränzle, M., Müller-Olm, M., Olderog, E.-R., Schenke, M., et al. (1995). Provable Correct Systems. Oxford report by FTP

66. Hein, P. (1966). *Grooks*. Gylling, Borgen Pocketbooks 85

67. Henzinger, T. A. (1996). Automatic Verification of Real-Time and Hybrid Systems. BRICS, Univ. of Aarhus BRICS Notes Series NS-96-5, Aarhus

68. Hinkel, U. (1996). A Formal Semantics for the Time Concept in SDL. manuscript.

69. Hoare, C. A. R. (1969). “An axiomatic Approach to Computer Programming.” *Communication of the ACM* **12** 576-580.

70. Hoare, C. A. R. (1978). “Communicating Sequential Processes.” *Communications of the ACM* **21**(8)

71. Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Hemel Hempstead, Prentice Hall International 0-13-153271-5.

72. Holzmann, G. J. (1988). “An Improved Protocol Reachability Analysis Technique.” *Software, Practice and Experience* 137-161.

73. Holzmann, G. J. (1991). *Design and Validation of Computer Protocols*. Englewood Cliffs, Prentice Hall International 0-13-539925-4.

74. Holzmann, G. J. (1996). *Early Fault Detection Tools*. TACAS96, Passau, Germany March 1996, Pages: 1-13, Springer-Verlag LNCS 1055

75. Holzmann, G. J. (1996). On-the-Fly Model Checking Tutorial. BRICS, Univ. of Aarhus BRICS Notes Series NS-96-6, Aarhus

76. Holzmann, G. J. and Patti, J. (1989). *Validating SDL Specifications: an Experiment*. 9th Int. Workshop on Protocol Specification, Testing and Verification, Twente, The Netherlands June 1989, North Holland

77. ISO (1989) ISO 8807 Information processing systems - Open System Interconnection – LOTOS– A Formal Description Technique Based on the Temporal Ordering of Observational Behaviour,

78. ITU (1993) Z.100 ITU Specification and Description Language (SDL), ITU-T, June 1994, 237 p

79. ITU (1993) Z.100 Annex F Specification and Description Language (SDL) Annex F. SDL Formal Definition, ITU, April 1994, (33+437+183) p

80. ITU (1993) Z.100 Appendix I SDL Methodology Guidelines, ITU-T, July 1994, 129 p

81. ITU (1994) Z.105 SDL combined with ASN.1, ITU-TS, Oct. 19.-27. 1994, 69 p

82. ITU (1994) Z.120 Annex B Algebraic Semantics of Message Sequence Charts, ITU-T, October 1994, 50 p

83. ITU (1996) Z.100 Addendum to Recommendation Z.100: CCITT Specification and Description Language, ITU, October 1996, 31 p

84. ITU (1996) Z.100 Supplement 1 SDL+ Methodology - Use of MSC and SDL (with ASN.1), ITU, October 1996,

85. ITU (1996) Z.106 Common Interchange Format, ITU-TS, Oct. 18. 1996, ? p

86. ITU (1996) Z.120 Message Sequence Charts (MSC), ITU-T, Oct. 1996, 78 p

87. Jones, C. B. (1986). *Systematic Software Development Using VDM*. Hemel Hempstead, Prentice-Hall International 0-13-880717-5.

88. Jonsson, B. (1987). *Compositional Verification of Distributed Systems*. Ph. D., Department of Computer Systems, Uppsala University

89. Jonsson, B. (1994). “Compositional Specification and Verification of Distributed Systems.” *ACM Transactions on Programming Languages and Systems* **16**(2) 259-303.

90. Kaasbøll, J. J. (1996). Aspects of object-oriented modelling. Concepts for analysis and guidelines for design. University of Oslo Dr. Philos Thesis Research Report No. 214, Oslo

91. Kahn, G. (1974). *The semantics of a simple language for parallel programming*. IFIP Congress '74, Stockholm, Sweden Pages: 471-475, North-Holland

92. Keller, R. M. (1975). *A fundamental theorem of asynchronous parallel computation*. Parallel Processing: Proceedings of the Sagamore Computer Conference, August 20.-23. 1974, Pages: 102-112, Springer LNCS 24

8 References

93. Kim, H. C., Choi, H., Yim, C. H. and Hong, J. P. (1991). *The Automated Verification of SDL Specification Using Numerical Petri-nets*. SDL´91 Evolving Methods. Proceedings of the Fifth SDL Forum, Glasgow, October 1991, Glasgow October 1991, North Holland: Elsevier

94. Kwong, Y. S. (1977). “On reduction of asynchronous systems.” *Theoretical Computer Science* **5** 25-50.

95. Lam, S. S. and Shankar, A. U. (1984). “Protocol Verification via Projections.” *IEEE Transactions on Software Engineering* **SE10**(4) 325-342.

96. Lewerentz, C. and Lindner, T. (1994). Case Study "Production Cell": A Comparative Study in Formal Specification and Verification. Forschungszentrum Informatik an der Universität Karlsruhe FZI-Publication 1/94, Karlsruhe

97. Madsen, O. L., Møller-Pedersen, B. and K., N. (1993). *Object-Oriented Programming in the BETA Programming Language*. Wokingham, England, Addison-Wesley 0-201-62430-3.

98. Mauw, S. (1993), "An algebraic semantics for Message Sequence Charts", TD 40 at ITU SG 10 WG MSC Geneva October 19. 1993

99. Mazurkiewicz, A. (1986). *Trace theory.* Petri Nets: Application and Relation to other Models of Concurrency, Bad Honef September 8.-19. 1986, Pages: 279-324, Springer-Verlag LNCS 254-255

100. Melham, T. F. (1996). Some research Issues in Higher Order Logic Theorem Proving. BRICS, Univ. of Aarhus BRICS Notes Series NS-96-7, Aarhus

101. Meyer, B. (1988). *Object-Oriented Software Construction.* Prentice Hall

102. Milner, R. (1980). *A Calculus of Communicating Systems*. Springer Verlag

103. Milner, R. (1989). *Communication and Concurrency.* Hemel Hempstead, Prentice Hall 0-13-115007-3.

104. Müller, O. and Nipkow, T. (1995). *Combining Model Checking and Deduction for I/O-Automata*. Tools and Algorithms for the Construction and Analysis of Systems, Pages: 1-16,

105. Nahm, R. E. M. (1994). *Conformance Testing Based on Formal Description Techniques and Message Sequence Charts.* Inauguraldissertation, Universität Bern

106. Olsen, A., Faergemand, O., Møller-Pedersen, B., Smith, J. R. W. and Reed, R. (1994). *Systems Engineering Using SDL-92*. North Holland 0 444 89872 7.

107. Pollack, R. (1996). What we Learn from Formal Checking. Part I: How to Believe a Machine-Checked Proof. BRICS, Univ. of Aarhus BRICS Notes Series NS-96-8, Aarhus

108. Pollack, R. (1996). What we Learn from Formal Checking. Part III: Formalization is Not Just Filling in Details. BRICS, Univ. of Aarhus BRICS Notes Series NS-96-10, Aarhus

109. Prinz, A. (1994), "BSDL The Language", TD PL/10-65 at ITU-TS Geneva Oct. 19.-27. 1994

110. Q-Labs Cleanroom Competency Centre (QCCC) (1992). *Cleanroom Software Engineering Applied to Telecommunications.* NSDCS'92,

111. Qin, H. and Lewis, P. (1990). *Factorization of Finite State Machines under Observational Equivalence*. CONCUR '90. Theories of Concurrencey: Unification and Extension, Amsterdam Pages: 427-441, Springer-Verlag LNCS 458

112. Rapporteur, I. S. Q. (1996), "Draft Recommendation Z.100 Addendum", COM 10-17 at ITU SG 10 Geneva 10-18 April 1996

113. Rational (1997). Unified Modeling Language. Rational Software Corporation WWW Manuals Version 1.0, Santa Clara

114. Ravn, A. P. e. (1991). Embedded, Real-time Computing Systems. ProCoS project Working Draft Denmark, England

115. Reed, R. (1996). “Methodology for real time systems.” *CN&ISDN* (June 1996) 1685-1702.

116. Reisig, W. (1985). *Petri Nets. An Introduction*. Berlin, Springer-Verlag 4 3-540-13723-8.

117. Reniers, M. (1995). *Syntax requirements of Message Sequence Charts*. SDL'95 with MSC in CASE. Proceedings of the Seventh SDL Forum, Oslo, Norway Sept. 26.-29. 1995, North-Holland

118. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. (1991). *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey, Prentice Hall 0-13-629841-9.

119. Rumbaugh, J. e. a. (1997). Unified Modeling Language Version 1.0. Rational Software Corporation

120. Sarma, A. and Bræk, R. (1995). *SDL'95 with MSC in CASE. Proceedings of the Seventh SDL Forum*. SDL'95 with MSC in CASE., Oslo, Norway 26.-29. Sept. 1995, North-Holland, Elsevier

121. Seltveit, A. H. (1994). Complexity Reduction in Information Systems Modelling. NTH, Norwegian Technical Highschool Dr. Ing. Thesis IDT 1994:8, Trondheim

122. Sethi, R. (1974). “Testing for the Church-Rosser Property.” *Journal of the ACM* **21**(4) 671-679.

123. Shen, Y.-N., Lombardi, F. and Dahbura, A. T. (1989). *Protocol Conformance Testing Using Multiple UIO Sequences*. 9th Int. Workshop on Protocol Specification, Testing and Verification, Twente, The Netherlands June 1989, Pages: 131-143, North Holland

124. Souissi, Y. (1991). *A Modular Approach for the Validation of Communication Protocols using FIFO nets*. Protocol Specification Testing and Verification XI, Pages: 143 – 158, Elsevier Science Publishers B.V., North Holland XI

125. Spies, K., Broy, M. and Merz, S. (1996). *Formal Systems Specification: The RPC-Memory Specification Case Study*. Berlin, Springer-Verlag

126. Srivas, M. K. (1996). A Combined Approach to Hardware Verification: Proof-Checking, Rewriting with Decision Procedures and Model-Checking. Part I: Slides. BRICS, Univ. of Aarhus BRICS Notes Series NS-96-11, Aarhus

127. Srivas, M. K. (1996). A Combined Approach to Hardware Verification: Proof-Checking, Rewriting with Decision Procedures and Model-Checking. Part II: Articles. BRICS, Univ. of Aarhus BRICS Notes Series NS-96-12, Aarhus

128. Stølen, K. (1995). *Development of SDL Specifications in Focus*. SDL'95 with MSC in CASE. Proceedings of the Seventh SDL Forum, Oslo, Norway 26.-29. Sept. 1995, Pages: 269-278, North-Holland, Elsevier

129. Stølen, K., Dederichs, F. and Weber, R. (1993). “Assumption/Commitment Rules for Networks of Asynchronously Communicating Agents.” *Draft copy*

8 References

130. Stroustrup, B. (1992). *The C++ Programming Language. Second Edition*. Addison-Wesley

131. Telelogic (1996). Methodology Guidelines. The SOMT Method. Telelogic Manual SDT 3.1 SOMT 0.2, Malmø

132. Uselton, A. C. and Smolka, S. A. (1994). *A Compositional Semantics for Statecharts using Labeled Transition Systems*. CONCUR '94, Proc. 5th Int. Conf. on Concurrency Theory, Uppsala, Sweden Pages: 2-17, Springer-Verlag LNCS 836

133. Vardi, M. (1996). An Automata-Theoretic Approach to Linear Temporal Logic. *Logics for Concurrency. Structure versus Automata*. Eds. F. Moller and G. Birtwistle. Berlin, Springer-Verlag. Lecture Notes in Computer Science LNCS 1043 238-266.

134. Vergauwen, B. and Lewi, J. (1993). *A Linear Local Model Checking Algorithm for CTL*. CONCUR'93, 4th International Conference on Concurrency Theory, Hildesheim, Germany August 1993, Pages: 447-461, Springer-Verlag LNCS 715 3-540-57208-2.

135. Verhaard, L. (1996). “An introduction to Z.105.” *CN&ISDN* (June 1996) 1617-1628.

136. Verilog (1994). GEODE Simulator. Basic Concepts. *Reference Manual - GEODE Simulator* . Toulouse, France, Verilog. 1-36.

137. Weissman, C. (1967). *LISP 1.5 Primer*. Dickenson Publishing Company, Inc.

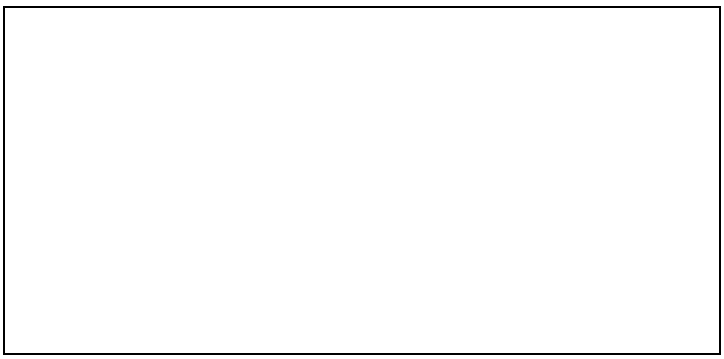
138. Wezeman, C. D. (1990). *Protocol conformance testing using multiple UIO-sequences*. Workshop on Protocol Specification, Verification and Testing, volume 9 of Proceedings of the IFIP WG 6.1 9th International Symposium on Protocol Specification, Testing and Verification, Pages: 131-143, North-Holland

139. Yang, F. and Chen, J. (1991). *REAS – A ripple effect analysis system*. SDL´91 Evolving Methods. Proceedings of the Fifth SDL Forum, Glasgow, October 1991, Glasgow October 1991, North Holland: Elsevier

140. Yourdon, E. (1989). *Structured walkthroughs*. Englewood Cliffs, N.J., Yourdon Press 0-13-855289-4.

141. Zhao, Z. and Bochmann, G. v. (1986). *Reduced reachability analysis of communication protocols: a new approach*. 6th International Workshop on Protocol Specification, Testing and Verification, Montreal, Quebec June 1986, North-Holland

9 Summary of new SDL constructs



9. Summary of new SDL constructs

We have found the need to introduce a few new SDL constructs to support the Mn approach and to be able to describe the systems which we find interesting in this context. Here we summarize these constructs.

There are four new constructs and they serve different purposes.

1. *Fair Non-deterministic Decisions* serve to facilitate specification of systems such that progress can be determined.
2. *Merge, spontaneous save* and *spontaneous consumption* are mechanisms which make it possible for the Mn-approach to express the non-determinism introduced by fair merge of signals from different channels.
3. *Signals as variables* help to generalize certain behavior patterns such that they become independent of exactly which signal arrives. This is used in the RPC-Memory example.

9.1 Fair Non-deterministic decision

Standard SDL-92 has anyvalue expressions which can be applied in decisions. Then the decisions represent a non-deterministic choice where we have no knowledge whatsoever about the chances of one alternative against the others. This is not practical when fairness is the issue. By introducing a simple notation (+) on a branch from a non-deterministic decision, we describe that this branch has a positive probability.

This means that if this decision is executed an infinite number of times, the “positive” alternative will be chosen an infinite number of times. We define that for every infinite subsequence of decision answers, any (+) alternative should appear infinitely many times. This represents extreme fairness.

The suggested extended notation for extreme fairness in SDL is shown in Figure 152 (p. 284). An imperative definition of our fairness construct is given in Figure 153 (p. 284).

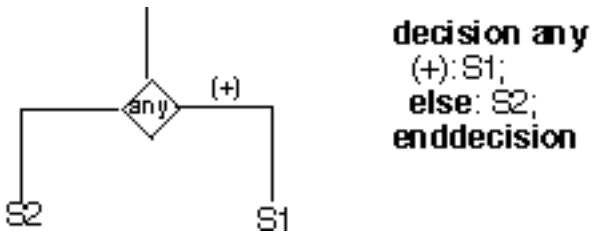


Figure 152: Fair Decision

```
dcl z1 Integer:=any(Natural);
dcl z2 Natural:=any(Natural);
...
z2:=any(Natural);
decision (z1<=z2)
  (true):  S1; z1:=any(Natural);
  (false): S2: z1:=z1-1;
enddecision;
```

Figure 153: Imperative definition of fairness

To obtain extreme fairness the any-expression must be random such that all Natural numbers have the possibility to be chosen.

9.2 Merge, spontaneous save and spontaneous consumption

In our Mn-approach it is very important that any non-determinism is described explicitly. In systems where non-determinism based on race conditions between signals is acceptable we need a way to describe the race condition explicitly. Our approach is the imperative “spontaneous save”.

There are three elements to this feature: the merge state, the spontaneous save and the spontaneous consumption.

Merge state

The **merge** state is the user’s way to express that there is an explicit race condition which is considered acceptable. Normally either no basic states or all basic states of a process are merge states.

Spontaneous save

The spontaneous save is the clue to the mechanism. The spontaneous save is not used by specifiers, but comes as a result of reduction and in the Mn-procedure. The idea is that when a signal is about to be consumed in a merge state, it is instead spontaneously saved. The complete state will normally become semi-stable. When the process is in a

merge state, all normal signals will be spontaneously saved. In reductions the spontaneous saves may occur also inside transitions since they are often operations on inner processes.

Spontaneous consumption

The spontaneously saved signals are then spontaneously consumed. At any point in time, when the process is in a merge state, the first spontaneously saved signals of any channel may or may not be consumed. The spontaneous consumption is considered globally triggered and in reductions they appear as external signals. Only the first signals may be spontaneously consumed. This means that the order of the signals within a channel is not changed, but the signals on different channels are merged.

All in all these constructs serve to describe in a finite way the infinite set of signal mergers. The graphical notation is shown in Figure 154 (p. 285).

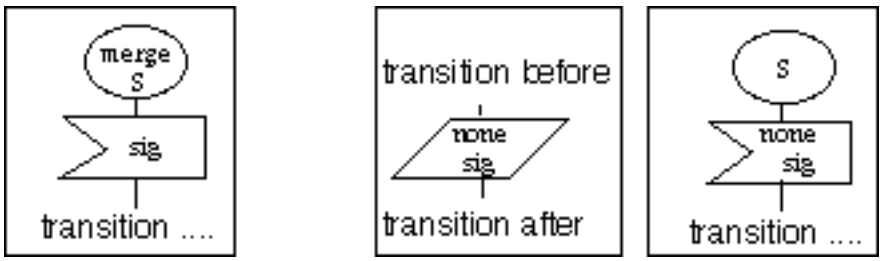


Figure 154: Merge state, spontaneous save and spontaneous consumption

9.3 Signal objects as data objects

As any SDL implementation has discovered, there is definitely some gain in harmonizing signals and variables. We have done this by the following scheme:

1. *Signal data type*. We define a new predefined data type which is the set of all signals. It is designated SIGNAL, such that a declaration of a signal variable will look like: “**dcl** my_sig SIGNAL;”. The signal data type may then also appear as parameter to another signal. It is also possible to define variables of specific signal types by extending the syntax: “**dcl** my_sig SIGNAL mysignaltype;”
2. *Output*. It is possible to output a stored signal by outputting the variable: “**output** my_sig;” The SENDER attribute of the signal is modified to SELF of this process[1].
3. *Dash signal*. We define a predefined function (called DASHSIGNAL) which returns a SIGNAL value which is equal to the most recently consumed signal of the process.
4. *Check signal type*. We define a Boolean function which can be used to check the signal type of a signal variable: “my_sig **is** return;”. A similar construct can then be used to interpret parameters: “my_sig **qua** exceptreturn(param);”.
5. *Atleast input*. We may consume signals of different signal types in one transition by specifying a supertype of the signal types by “**input atleast** return;”. Combination with DASHSIGNAL and signal type check makes it possible to use this effectively. In general we can use “**atleast** signaltype” also in signallists to indicate that any signal type which is inherited from signal type is allowed.

1. There may also be a need to keep the original SENDER of the signal. This can always be done manually as parameter to the signal, but we could also make a construct to circumvent the modification of SENDER.

All together these features make it possible to generalize SDL communication more wrt. signals. We have given an example in Figure 155 (p. 286).

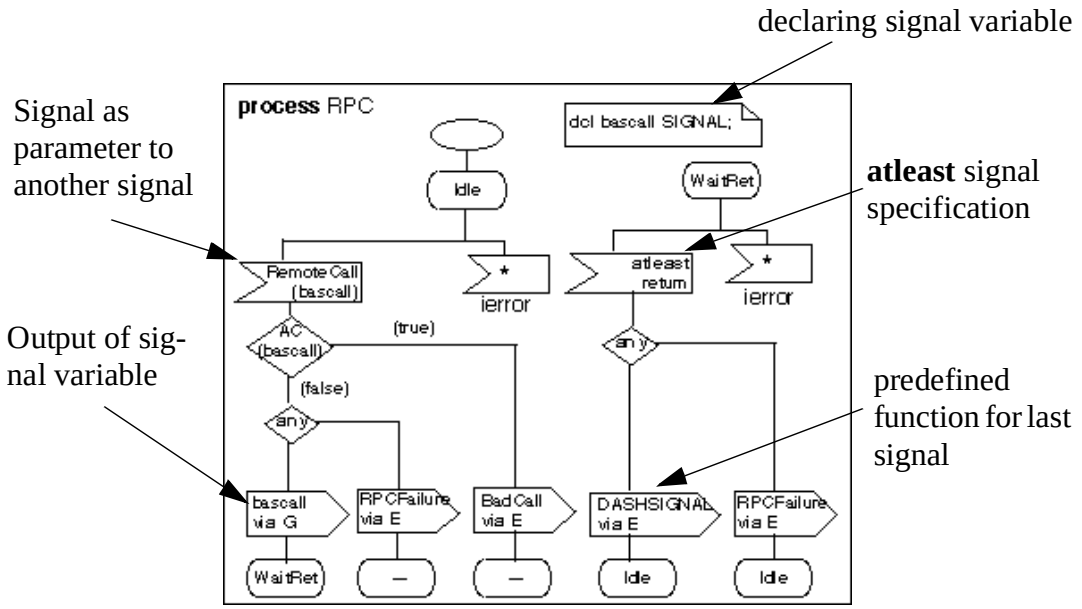


Figure 155: Example of signal as data

10 List of Figures

10. List of Figures

10 List of Figures

10 List of Figures