University of Oslo
Department of Informatics

Validation of Contract Decomposition by Testing

Mass Soldal Lund

Cand. Scient. Thesis

February 2002

Abstract

In this cand.scient. thesis we propose a strategy for testing validity of decomposition of contract oriented specifications. The strategy is based on Abadi and Lamport's Composition Theorem for the *Temporal Logic of Actions* and test case generation from executable specifications.

A composition rule, inspired by the Compositon Theorem, is formulated in a semantics based on timed streams. A subset of the *Specification and Decription Language* (SDL) is defined and the SDL subset is formalized in the semantics.

A simplification of the testing strategy was realized in an experimental prototype tool for testing of contract decompositions in SDL. In addition another prototype tool based on a conventional strategy was built as a reference tool.

Testing of the two tools showed that both validated valid contract decompositions and falsified invalid contract decompositions. Testing also showed that the tool based on the composition rule in some interesting situations was considerably more efficient than the tool based on the conventional strategy.

Foreword

This is a thesis for the cand.scient. degree at the Department of Informatics, Faculty of Mathematics and Natural Sciences, University of Oslo. The work of this thesis has been supervised by Ketil Stølen. While writing this thesis I have been a student at SINTEF Telecom and Informatics.

I would like to use the opportunity to thank Ketil Stølen and for good and inspiring supervision, and for an interesting time as his student, SINTEF Telecom and Informatics for taking good care of their students, Telelogic for providing me with a license for Telelogic Tau, and Folker den Braber for useful and interesting discussions. Thanks to Bendik, Linda, Eirik and Geir for being good friends.

Mass Soldal Lund

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this cand.scient.[1] thesis we develop a strategy for testing the validity of decomposition of contract oriented specifications. The strategy is inspired by Abadi and Lamport's *Composition Theorem* [Abadi and Lamport, 1995], and is in this thesis refered to as the *Abadi/Lamport testing method*.

A conventional strategy for testing of contract decomposition, which we refer to as the *Direct testing method*, is also described.

Both testing methods are formalized the stream-based semantics of the the FOCUS method [Broy and Stølen, 2001], and are based on principles from functional testing.

We explain how both testing methods were simulated with the *Description and Specification Language* (SDL) [ITU-T, 2000b] and the commercial tool Telelogic Tau SDL Suite [Telelogic, 2001]. Emperical testing was conducted in order to validate the correctness of the Abadi/Lamport testing method and in order to compare the two testing strategies.

Section 1.1 gives motivation for this work and section 1.2 offers an overview of the thesis.

1.1 Motivation

Today's computerized systems tend to be large and complex. They are often distributed, which means different parts of the system process in parallel, and open in the sense that they run in environments which potentially can give the systems any input. While computerized systems become a more integrated part of infrastructure and society, there is also an increasing need

[1] Cand.scient. is short for candidatus scientarum, a degree comparable to Master of Science.

for ensuring the reliability of such systems. Cost and effectiveness in a development process are not insignificant.

We believe that a computerized tool based on the Abadi/Lamport testing method could ease the development and maintenance of distributed systems, make the systems more reliable and make the development more efficient.

1.1.1 Decomposition

All distributed systems consist of several components. A well known approach to specification of component based systems is to start with a specification of the system as a whole and then identify and specify the components of the system. This decomposition can be done in several iteration, so that a specified component of the system is decomposed into sub-components.

This is not a new principle. In 1966 Börje Langefors proposed *the fundamental principle of systems work*[2]:

> "Partition the system works into seperate tasks, a through d,
>
> a. *Definition of the system as a set of parts.*
> List all parts from which the system is regarded as built-up.
>
> b. *Definition of system structure.*
> Define all interconnections which make up the system by joining its parts together.
>
> c. *Definition of the system parts.*
> For each singel part (or group of similare parts) seperately define its properties as required by the system work at hand and do this in a format as specified by the way the systems structure is defined (in task b).
>
> d. *Determination of the properties of the system.*
> Use the definitions as produced by the tasks a, b, and all seperate tasks c, all taken together. Compare with specifications wanted for the system and repeat a, b, c, and d until satisfied."[3]

With this approach to system development there is a need for methods for validating that a composition of specified components refines a higher level

[2]We understand *systems work* as Langefors' expression for what we call *system development*.

[3]The quote is from [Langefors, 1973], but the fist edition from 1966 also contains *the fundamental principle*

specification. We refer to this as validation of decomposition. In the development of distributed systems such methods for validation must be able to handle concurrancy.

Our work is inspired by Abadi and Lamport's Composition Theorem. The Composition Theorem is used for performing this validation by the means of logical proofs, and can be said to belong to the field of *formal methods*.

1.1.2 Formal Methods

Formal methods can be seen as a branch of theoretical computer science where the objectives are to formalize, and make abstract models for reasoning about, matters such as computer programs, programming languages and system development techniques.

The history of theoretical computer science can be traced back to at least the 1930s. Lambda-calculus orignates from a paper by Alonzo Church published in 1933 [Church, 1933] and in 1936 Alan M. Turing published his first paper on what was later called Turing-machines [Turing, 1936]. Both lambda-calculus and Turing-machines can be seen as mathematical models of computation, which at the time was performed by humans and mechanical devices. In the 1960s groups of computer scientists began formulating models, based on mathematics and logic, for reasoning about computer programs and programming languages. Most famous is probably the Hoare-logic [Hoare, 1969] (see section 2.3.1). In the 1970s systems with parallelism came in focus [Jones, 1992], and in the 1980s and -90s a considerably amount of research in this field has been on such systems. Approaches refered to in this thesis are [Jones, 1981], [Misra and Chandy, 1981], [Lamport and Schneider, 1984], [Hoare, 1985], [Abadi and Lamport, 1990], [Abadi and Lamport, 1995], [Cau and Collette, 1996], [Stølen, 1996] and [Broy and Stølen, 2001].

Most of the work within formal methods has been formalization, but there have also been made attempts to apply formal methods directly in system development. There are however few examples of this being successful [Finney and Fenton, 1996, Fenton and Pfleeger, 1997]. Some of the reason may be that formal techniques are tedious and time-consuming, and people without thorough training often find them hard to understand [Finney, 1996]. As a result, formal methods are little used in real-life system development, especially in development of commercial software [Finney and Fenton, 1996].

As we explain in chapter 3, we still believe that the results from formal methods could be useful in system development. Our work is based on two prinsiples from formal methods; *contract oriented specification* and the *composition principle*.

1.1.3 Contract Oriented Specifications

Contract orientated specifications orginate from [Jones, 1981] and [Misra and Chandy, 1981]. As indicated by the name, contract oriented specifications are specifications formed as contracts. The contract oriented specification of a component consists of an *assumption* and a *guarantee*. The contractual relationship is that the component should behave as specified in the guarantee as long as the environment of the component behave as specified in the assumption.

Contract oriented specifications evidently have some advantages. With this specification format we are able to distinguish, in a logical way, the functionality of a component from the conditions that must be fulfilled for the component to work. A perhaps more importent advantage is that we get a specification format which emphasize what is demanded from the environment of a specified component. We can decrease the chance of unpleasant surprises when integrating components because the requirements on the context of the components are explicitly specified. We believe for example outsourcing of implementation, use of standard components and re-use of components could be made easier with contract oriented specifications.

Contract oriented specifications are presented in more detail in section 2.3.

1.1.4 The Composition Principle

Abadi and Lamport's Composition Theorem is a rule for validating decomposition of contract oriented specifications by logical proofs, and may in certain respects be seen as a generalization of similar rules in [Jones, 1981] and [Misra and Chandy, 1981]. We believe the Composition Theorem can be generalized to a universal principle for validation of contract decomposition. We refere to this principle as the *composition principle*. Results from [Cau and Collette, 1996], which show that the principle is language independent, support this belief. In this respect, both the Composition Theorem and the composition rule we propose in chapter 6 are instances of the composition principle. Other instances of the principle can be found in [Abadi and Lamport, 1990] and [Stølen, 1996].

1.1.5 Functional Testing at Specification Level

Development of distributed computerized system will often include large groups of people. Different parts of a system are developed by different

persons, and the development process involve persons that not are experts on system development.

Methodologies for system development must be able to handle this. We believe that good methods for specification, and validation at specification level, are essential parts of such methodologies. It is not uncommon practice to do testing of integration after the components are implemented. If it then should become evident that the components are not able to communicate or that the integration does not give the expected result, it may be necessary to make changes in the implementation or even make new implementations of some of the componenets. If testing of integration is done at specification level, the risk for implementing components that cannot be integrated is reduced.

Validation is usually done by automated or manual testing, but testing techniques are often informal or semi-formal and much of the testing usually occur at the end of a system development process. We believe that both integration testing during the specification phase of a system development process and testing techniques with a formal fundament can make system development more efficient and make systems more reliable.

1.1.6 Graphical Specification Techniques

The last decade graphical, diagram based specification languages have been dominating the system development industry. Experience have proved that these specification languages are useful, especially because they are easy to learn and understand. Because of this we focus on graphical specification techniques.

As specification language in our work we use the *Specification and Description Language* (SDL), which is a graphical specification language based on state transition diagrams and dataflow diagrams.

- SDL process diagrams (state trancition diagrams) are used for specifiying behavior.
- SDL block diagrams (dataflow diagrams) are used for specifying static communication structure.

We decided to use SDL because

- SDL is a well known and widly used specification language;
- SDL has a graphical, diagram based representation;
- SDL specifications are executable and can therefore be used for generating and executing tests;

- SDL has sufficient tool support for the tasks we carried out in our work.

Although we use SDL to test our hypotheses, our results should carry over to the class of executable specifications.

1.1.7 Emperical Testing

The validity of the proposed testing strategy cannot be established without emperical testing. It is possible to make mathematical arguments concerning correctness and consistency of the semantics, but this does prove that the strategy, as a system development technique, yields correct results. Semantics are abstractions of reality. The only way of showing that abstract principles are valid in practice is by application. The need for experiments and emperical evaluation when introducing new principles and techniques to system development is discussed in [Tichy, 1998] and [Zelkowitz and Wallace, 1998].

A full validation based on emperical testing is theoretical impossible, but after conducting successful experiments on the strategy, there are reasons to be more confident in its validity. Furthermore we have no other method for measuring efficiency, which is a common requirement of system development techniques.

We are also of the opinion that purly theoretical reflections are of little value if we are not able to relate them to practical system development.

In [Tichy et al., 1995] and [Zelkowitz and Wallace, 1998] results are presented that show a considerably lack of experimentation and emperical testing in computer science compared to other sciences. Another motivation for conducting emperical tests is to contribute to a development in computer science towards a broader use of scientific methods.

1.2 Overview

When this thesis was written, we had tree goals conserning the structure:

1. It should be possible to read the thesis from beginning to end.
2. The chapters represent a logical partition of the work conducted.
3. Each of the chapters should be able to stand alone, to a degree as high as possible.

The two first goals we think we have managed fairly well. The last goal is not reached to the degree we wanted. The result is quite an amount of cross

references. We hope this not confuses the reader, but makes the thesis more readable.

The structure of the thesis is as follows:

- Chapter 2 provides background on the techniques and theories this work is based on.
- Chapter 3 provides a thorough analysis of the hypotheses of this work and an overview of the tasks executed in the investigation into these hypotheses.
- Chapter 4 provides the definition of a SDL subset, general semantics and a formalization of the SDL subset.
- Chapter 5 provides formalization of contract oriented specifications and a scheme for specifying contracts in SDL.
- Chapter 6 contains the composition rule, a corollary and a proof of the composition rule.
- Chapter 7 explains how we built two experimental prototype tools for testing of contract decompositions of SDL specifications; one based on direct testing and one based on the composition rule.
- Chapter 8 describes the tests conducted with the prototype tools and discusses the results of these tests.
- Chapter 9 provides a discussion of possible generalizations of the work presented in this thesis.

In addition there are tree appendices.

- Appendix A contains the code of the testing tools.
- Appendix B contains a full test example.
- Appendix C contains raw test results form testing of the tools.

Figure 1.1 provides a "dependency graph" of the main sections in this thesis, which we hope clearify the structure.

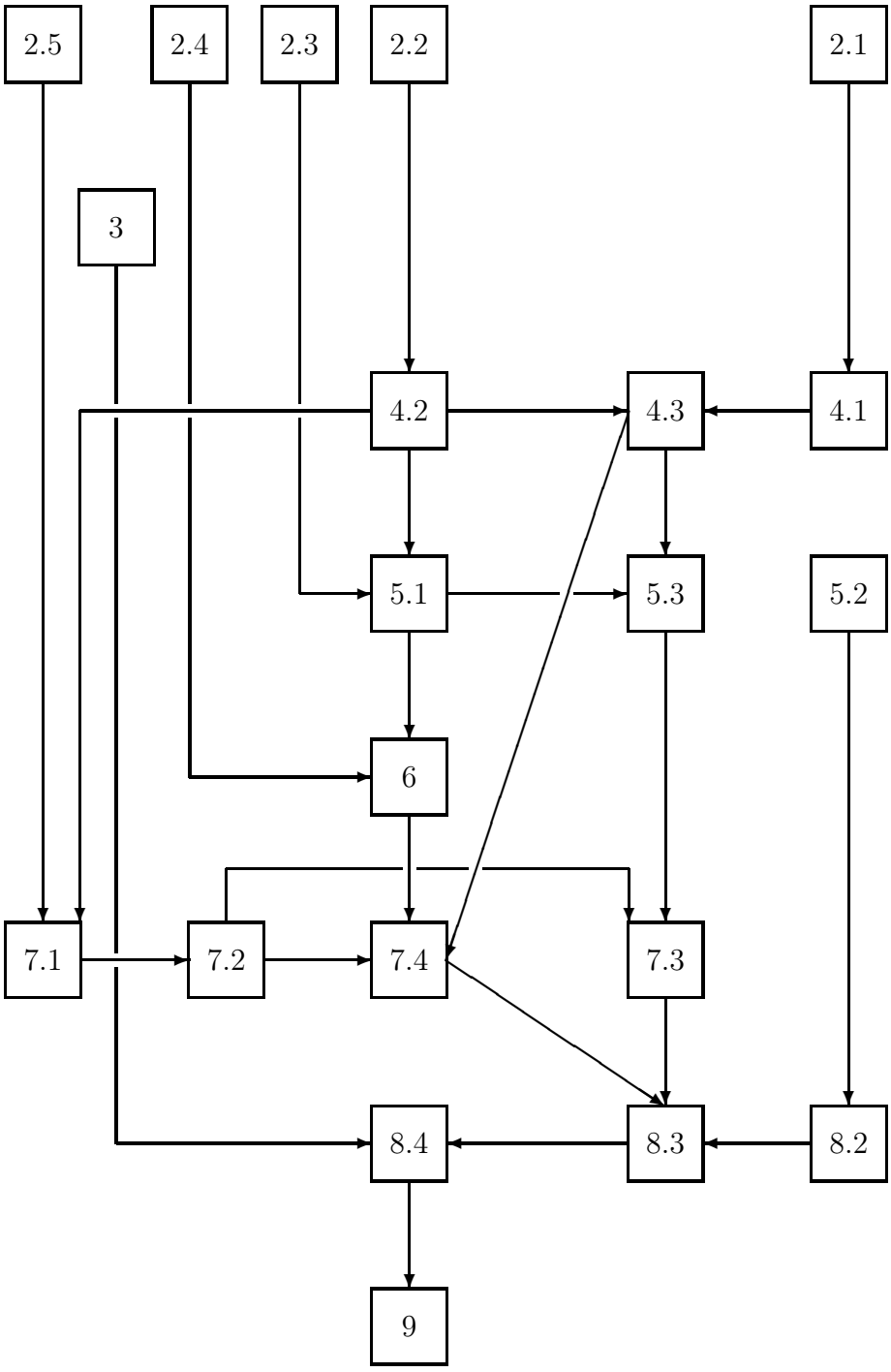


Figure 1.1: Overview of thesis

Chapter 2

Background

This chapter provides some background on the techniques and theories that the work of this thesis is based on. Section 2.1 provieds background on the *Specification and Description Language* and section 2.2 gives provides some background on the FOCUS method. In section 2.3 there is a presentation of contract oriented specifications, and in section 2.4 the composition principle is presented. Finally section 2.5 provides background on testing.

2.1 Specification and Description Language

The *Specification and Description Language* (SDL) [ITU-T, 2000b] is a standard of the International Telecommunication Union (ITU) and widely used as specification language in the telecommunication industry. SDL was first standardized by ITU in 1992, based on an earlier definition of the language. This standard is usually refered to as SDL-92. In 1996 minor changes to SDL-92 was made, and the result is often refered to as SDL-96. In 2000 ITU made a new standard for SDL, which is called SDL-2000. This work is based on SDL-92, but all features of SDL we use are also features of SDL-96 and SDL-2000.

SDL specifications have three main building blocks:

- SDL system;
- SDL block;
- SDL process.

An SDL system is specfied by a block interaction diagram, an SDL block is specified by a block interaction digram or a process interaction diagram and an SDL process is specified by a process diagrams. Block interaction

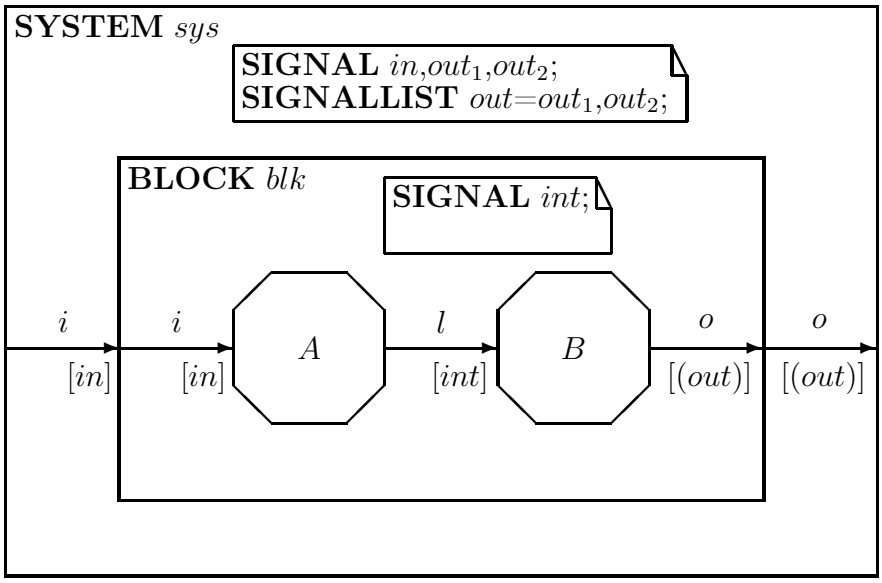


Figure 2.1: SDL block and process interaction diagrams

diagrams and process interaction diagrams are special kinds of dataflow diagram which shows repectively SDL blocks or SDL processes interconnected with channels[1]. The boundary of an SDL system represents the environment of the specified system. A point where a channel meets the boundary of a block is called a gate.

Process diagrams are a kind of state transition diagrams. These state transition diagrams are also called extended finite state machines; state machines with a finite number of states and extended with features like variables and timers.

Processes and blocks in an SDL system are assumed to run in parallel. Block and process interaction diagrams specify the static structure of the specified system, while the behavior is specified by process diagrams. Figures 2.1 and 2.2 show a small example SDL specification.

(It is also possible to specify processes as sevice interaction diagrams, where the services are specified by state transition diagrams. Further it is possible to specify block and process types which can have one or more instances, and procedures and functions. We do however not make use of these features.)

[1]In SDL-92 there is a distinction between channels and signal routes. In this thesis we do not distinguish between channels and signal routes, and refer to both as channels.

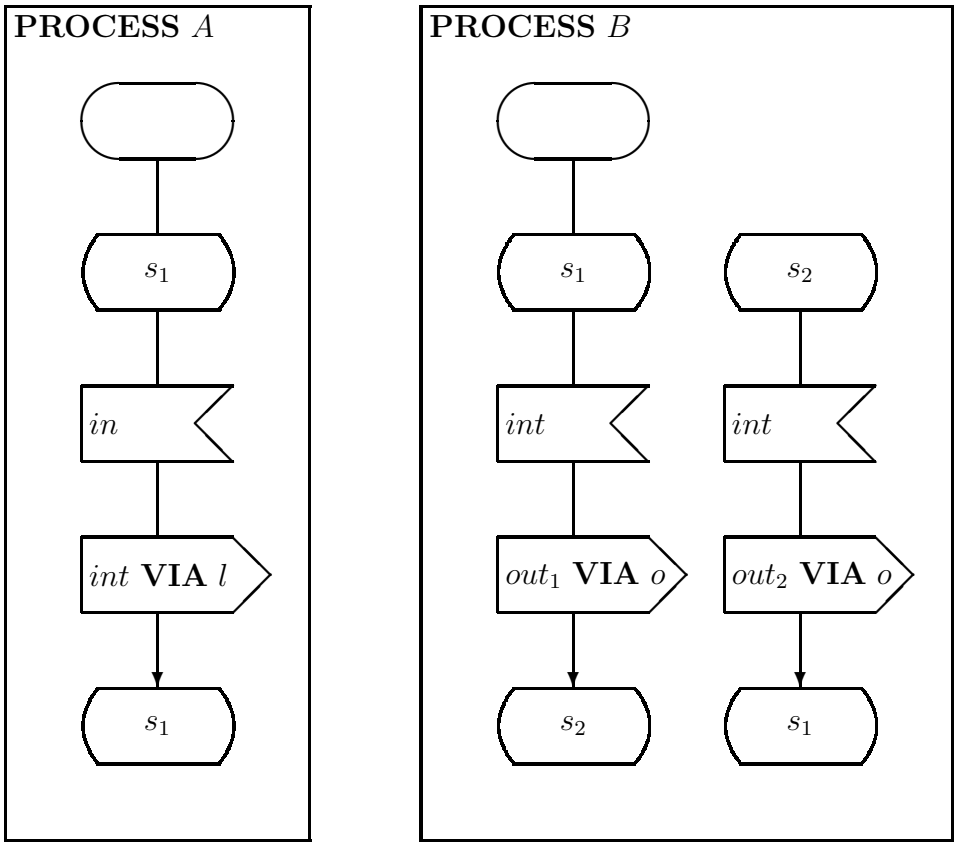


Figure 2.2: SDL state transition diagrams

2.2 FOCUS

The FOCUS method [Broy and Stølen, 2001] is a collection of system specification and description techniques, formalized in a stream-based semantics. Streams of messages represent communication histories, where the streams are infinite long and time dependent. Time is represented by time ticks in the streams, where each tick represtents the end of a time unit. In this semantics, behavior of systems and components are characterized by the relation between their input and output streams.

There are two specification techniques in FOCUS that are of special interest in our work; *dataflow diagrams* and *stream processing functions*.

Dataflow diagrams is a graphical technique for specifying compositions and communication structures of systems. FOCUS dataflow diagrams consist of components, drawn as boxes, and directed channels, drawn as arrows. Streams may be seen as recordings of the communications over channels.

Stream processing functions is a technique for specifying behavior of components in an algorithmic fashion that is easy to relate to state transition diagrams. A component is specified by a set of mutual recursive functions. The functions are defined by a **where-so that** clause and a set of equations on the functions; the functions are defined to be the functions that fulfil the conjunction of the equations.

These specifications are on the form

$$
\begin{aligned}
o = f_0(i) \textbf{ where } & f_0, f_1, \ldots, f_{n-1} \textbf{ so that } \forall i, o : \\
& e_1 \\
\wedge \quad & e_2 \\
& \vdots \\
\wedge \quad & e_m
\end{aligned}
$$

where i and o are tuples of input and output streams, $f_0, f_1, \ldots, f_{n-1}$ are the functions and $e_1, e_2, \ldots, e_m$ are the equation. The types of the functions and the streams are omitted, since they will follow from the context where this specification format is used.

2.3 Contract Oriented Specifications

The paradigm of contract oriented specifications[2] has its origin in the field of formal methods as a way of specifying components of component based systems with concurrency. The first explisit use of contracts are found in [Jones, 1981] and [Misra and Chandy, 1981], but in e.g. [Hoare, 1969] a kind of contracts is proposed. The paradigm is motivated by the fact that a system component usually will be specified for working inside a given context. This context – the component's environment – consists of other componenets of the system and possibly the external environment of the system. When specifying a component it is reasonable to make certain assumptions on the context in which the component will be working.

In the contract oriented paradigm, systems and system componets are specified by an *assumption* and a *guarantee*. The assumption describes the conditions under which a component is meant to run and the guarantee specify the requirements the component should fulfil when executed under the assumed conditions.

There exists various specification techniques and methods for validation within this paradigm. The contract oriented specifications in this thesis are

[2]Contract oriented specifications are also called assumption/guarantee, assumption/-commitment and rely/guarantee specifications. We also use the expressions contract specifications and contracts in this thesis.

inspired by the assumption/guaratee specifications for the Temporal Logic of Actions (TLA) in [Abadi and Lamport, 1995], the assumption/commitment specifications in [Stølen, 1996] and the assumption/guarantee style of specifications in the FOCUS method [Broy and Stølen, 2001]. Each of these techniques can be placed in one of two groups: state-based formalisms and stream-based formalisms.

2.3.1 Contracts in State-Based Formalisms

In state-based formalisms, behavior of systems and components are characterized by sequences of states. A state in this respect is seen as a tuple of values.

Hoare-Logic

An early and well-known state-based formalism that uses a kind of contracts is the Hoare-logic for program specification and verification [Hoare, 1969], with its pre-conditions and post-conditions. In Hoare-logic a program is specified by the triplet

$$\{P\}S\{Q\}$$

where S is a program and P and Q are predicates that specify program states[3]. The tiplet assert that if the pre-condition P is true when S is initialized, we require that the post-condition Q is true when S terminates, i.e., S must fulfil Q if the assumed program state P is true when the execution of S starts.

Temporal Logic of Actions

The Temporal Logic of Actions (TLA) [Lamport, 1994, Abadi and Lamport, 1995] is a specification language where systems are specified by logical formulas. Its semantics are state-based, and TLA can be seen as predicate-logic extended with actions that contain a temporal element.

In TLA the behavior of a system is viewed as an infinite sequence of states, where the states are assignments of values to variables. While a predicate concerns the properties of one state and is **true** or **false** for that state, an action is a relation between two consecutive states and is **true** for a pair of states if the action is able to bring the system from the first state to the other, and **false** otherwise.

[3]Hoare uses the notation $P\{S\}Q$.

TLA specifications have the form

$$M \stackrel{\text{def}}{=} \exists x : Init \wedge \Box[\mathcal{N}]_v \wedge L$$

and are either **true** or **false** for a behavior. A behavior σ is said to satisfy M iff M is **true** for σ. This is denoted by $\sigma \models F$. F is valid iff F is satisfied by all possible behaviors. This is denoted $\models F$.

$Init \wedge \Box[\mathcal{N}]_v$ specify the safety properties and L the liveness properties of M. $Init$ is a predicate that must be **true** for the first state of the behavior, i.e., the initial conditions. $\mathcal{N}$ is an action and $\Box$ is the temporal operator *always*, so $\Box\mathcal{N}$ is **true** if $\mathcal{N}$ is **true** for every pair of consecutive states. v is a tuple with the variables in the formula, and $\Box[\mathcal{N}]_v$ means we allow steps that leave v unchanged. $\exists x : F$ means that there exists a sequence of values for x that satisfies F. $\exists$ is used to hide internal variables of a specification.

Contract specifications in TLA are specified with a special symbol $\stackrel{+}{\triangleright}$. A contract is specified by

$$E \stackrel{+}{\triangleright} M$$

where E and M are TLA formulas representing respectively the assumption and the guarantee of the specified system or component.

$E \stackrel{+}{\triangleright} M$ means that M is true at least one step longer than E if E ever becomes **false**. If $\sigma|_n$ denotes σ truncated after the n first states, the precise definition is that σ satisfies $E \stackrel{+}{\triangleright} M$ iff σ satisfies $E \Rightarrow M$ and for every $n \in \mathbb{N}$, if E is **true** for $\sigma|_n$ then M is true for $\sigma|_{n+1}$.

2.3.2 Contracts in Steam-Based Formalisms

Examples of contract oriented specifications in stream-based formalisms can be found in e.g. [Broy and Stølen, 2001] and [Stølen, 1996], both based on the semantics of the FOCUS method.

In [Broy and Stølen, 2001] contract oriented specification is defined by

$$\forall j \in \mathbb{N} : A\downarrow_j \Rightarrow G\downarrow_{i:j}\downarrow_{o:j+1}$$

$$A \Rightarrow G$$

where the assumption A and guarantee G are predicates over timed streams. $A\downarrow_j$ denotes A where the streams are truncated after time j, and $G\downarrow_{i:j}\downarrow_{o:j+1}$ denotes G where the tuple of input streams i is truncated after time j and the tuple of output streams o are truncated after time $j+1$.

In [Stølen, 1996] contract oriented specifications are defined by the means of guarded (strongly causal) functions on tuples of streams. A function τ is guared iff

$$i{\downarrow_j} = s{\downarrow_j} \Rightarrow \tau(i){\downarrow_{j+1}} = \tau(s){\downarrow_{j+1}}$$

where i and s are streams and $\downarrow_j$ denotes truncations of streams after time j. The definition of a contract oriented specification is

$$\{\tau | \forall i, j : \langle A \rangle(i{\downarrow_{j+1}}, \tau(i){\downarrow_j}) \Rightarrow \langle G \rangle(i{\downarrow_{j+1}}, \tau(i){\downarrow_{j+1}})\}$$

where all functions τ are guared and $\langle P \rangle$ denotes the prefix closure of P.

2.4 The Composition Principle

Abadi and Lamport's Composition Theorem is probably the best known instance of the composition principle. In [Abadi and Lamport, 1995] the Composition Theorem is formulated as

If, for $i = 1, 2, \ldots, n$,

$$\models \mathcal{C}(E) \wedge \bigwedge_{j=1}^{n} \mathcal{C}(M_j) \Rightarrow E_i$$

and

$$\models \mathcal{C}(E)_{+v} \wedge \bigwedge_{j=1}^{n} \mathcal{C}(M_j) \Rightarrow \mathcal{C}(M)$$

$$\models E \wedge \bigwedge_{j=1}^{n} M_j \Rightarrow M$$

then

$$\models \bigwedge_{j=1}^{n} (E_j \stackrel{+}{\triangleright} M_j) \Rightarrow (E \stackrel{+}{\triangleright} M)$$

E_{+v} means that the vector v of free variables will stop changing if E ever becomes false. The precise definition is that σ satisfies E_{+v} iff σ satisfies E or there is an n such that E is true for $\sigma|_n$ and σ never changes after the $n+1$ first states.

$\mathcal{C}(M)$ denotes the closure of a TLA formula M. $\mathcal{C}(M)$ is satisfied by a behavior σ iff every prefix of σ satisfies M, i.e., $\mathcal{C}(M)$ specifiy the prefix closure of the set of behaviors specified by M. $\mathcal{C}(M)$ can be seen as extracting the safety properties of M.

Other instances of the composition principle are formulated in [Abadi and Lamport, 1990], [Cau and Collette, 1996] and [Stølen, 1996].

2.5 Testing

There exists a variety of techniques for testing of systems that can be specified by finite state machines and extended and communicating finite state machines [Lee and Yannakakis, 1996].

In the work of this thesis we make use of established techniques for generating and executing test cases. We do testing of the external behavior of systems, and do therefore not consider e.g. techniques for identification of system states.

This approach we refer to as *functional testing*. In [Beizer, 1984] we find the following definition:

> "In functional testing the element is treated as a black box. It is subjected to inputs and its outcomes are verified for conformance to its specification."

We find this definition suitable for our approach. However note that also "the elements" under test will be specifications in our testing strategy.

The testing technique relevant in this thesis is a technique usually refered to as *state space exploration*. A state space exploration of an extended finite state machine builds a reachability graph from the machine, where the nodes are called *configurations*. Each configuration is a combination of a control state and assignment of values to the variables in the state machine. In order to avoid state explosion, equivalent configurations are collapsed into single nodes. Test cases can be extracted from the reachability graph, and test cases can be verified by searching the reachability graph for matching paths.

A general discussion on state space explorations is found in [Lee and Yannakakis, 1996], and the concrete state space algorithms used by the tool used in our work are described in [Ek, 1993] and [Koch et al., 1998]

Chapter 3

Problem Analysis

In this chapter we describe and analyse the main hypotheses on which this work is based. We also describe how these hypotheses are investigated. Section 3.1 describes the main hypotheses. Section 3.2 characterizes the problems we face in the investigation of these hypotheses. Section 3.3 describes the concrete strategy used in the investigation and section 3.4 provides the success criteria we have defined for the results.

3.1 Main Hypoheses

There are two concrete hypotheses on which this work is based. Both are instances of an underlying hypothesis on system development. This underlying hypothesis we call *The existence of universal principles for system development*, and is not investigated directly. The two concrete hypotheses of this report we refer to as *Application of functional testing on contract specifications* and *Application of the composition principle in functional testing*. In this section both the underlying and concrete hypoteses are motivated and formulated.

3.1.1 The Existence of Universal Principles for System Development

In section 1.1.2 we claim that formal methods, despite of a relatively long history, has no large place in real-life system development. We are of the opinion that this does not make the principles from this field less valid in themselves; it only shows that the way these principles have been applied in system development has not been very fruitful. On this background we formulate the underlying hypothesis:

> The field of theoretical computer science provides a number of theories and abstract principles about computer science and system development. These theories may be unified into univeral principles that are valid for system development in general. It should be possible to transform these principles into practical system development methods and tools that
>
> - will improve the efficiency of system development;
> - will improve the quality of the resulting systems;
> - will simplify management and maintance of systems;
> - does not require particulare training or understanding of the underlying logic and mathematics.

As indicated there is a requirement that system deveolpment methods and tools built in accordance to the universal principles must hide the underlying logic and mathematics. A method based on an abstract principle should be usable also without special competence in the logic and mathematics in which the abstract principle is formulated.

The existence of such universal principles is supported by, e.g. [Cau and Collette, 1996] where it is shown that the composition principle is language independent and [Lamport and Schneider, 1984] which demonstrates the broad applicability of formal principles for program verification.

3.1.2 Application of Functional Testing on Contract Specifications

As indicated in the presentation of contract oriented specifications in sections 1.1.3 and 2.3, we believe that this is a useful specification technique. Contract specifications have been used in formal techniques for validating properties of specifications. Our opinion is that the paradigm of contracts is general enough to be applied in functional testing. The hypothesis is formulated:

> It is possible to construct a general method for functional testing of the decomposition of concurrent systems described by contract oriented specifications with executable assumptions and guarantees. The method can be realized in a computerized tool.

3.1.3 Application of the Composition Principle in Functional Testing

In section 1.1.4 we describe how the composition principle has been used for proving validity of contract decompositions by means of logical proofs. We believe that the composition principle is a universal principle, so if there is a way of relating functional testing to contract specifications, it is possible to also apply the composition principle to functional testing of contract decompositions. Hence our hypothesis is:

> It is possible to apply the composition principle to contract oriented specification made for functional testing and make a general method for functional testing of decomposition of concurrent systems, which will be more efficient than merely direct testing.

3.2 Characterization of the Problem

The main objective of this cand.scient. thesis is to investigate the hypotheses in sections 3.1.2 and 3.1.3. We do not try to verify or argue for the underlying hypothesis presented in section 3.1.1 beyond this.

In order to do this investigation the work was divied into the following tasks:

- Define a general semantics for specifications.
- Define a suitable subset of SDL and adapt the semantics to this subset.
- Define contract specifications for the semantics and relate them to SDL.
- Formulate a composition rule for the semantics and relate it to SDL.
- Implement prototypes for simulation of two computerized tools for testing of contact decompositions in SDL:
  - *Direct testing tool*, based on a conventional strategy;
  - *Abadi/Lamport testing tool*, based on the composition rule;

  and performing emperical testing of contract decompositions with these prototype tools.

Below these tasks are described in more detail.

3.2.1 Specification Language

Even though we want to develop a general method, it is useful to relate the method to a specification language/technique to better show how the method is supposed to work. It is also necessary to concretize if we are to actually try out the method.

As specification language we use a subset of the *Specification and Description Language* (SDL) [ITU-T, 2000b]. Our main requirement is that the specification language is executable. SDL specifications are executable and there exists commercial tools that simulate SDL specifications. In addition SDL has a graphical, diagram based representation.

In SDL, state transition diagrams (state machines) are used to specify behavior, while dataflow diagrams are used to specify the static structures of systems. In order to keep the work within the scope of what can be expected from a cand.scient. thesis, we do not include features like abstract datatypes, composite states, instancation and inheritance. The SDL subset is defined in section 4.1.

3.2.2 Semantics

If we are to reason formally about specifications, there is a need for underlying formal semantics. There are several requirements that such semantics should fulfil:

- it must be able to handle concurrency and composition;
- it need a way of representing contract oriented specifications;
- there must be a way of relating it to functional testing;
- it must be possible to adapt it to different specification languages.

We define a stream based semantics based on the FOCUS method. This semantics should fulfil the requirements listed above. We also use stream processing functions from FOCUS to make a link between the stream based semantics and SDL.

Executable specifications are able to express arbitrary computational safety properties, but not arbitrary liveness properties (see section 4.2.6). The choice of specification language thereby reduces the set of properties we are able to specifiy.

3.2.3 Adaption of Composition Principle

Part of the goal is to use the composition principle in a testing strategy. To be able to do this we formulate a composition rule in our semantics, inspired by Abadi and Lamport's Composition Theorem. Since we have a formal semantics we are able to do formal reasoning about it, and a proof of soundness of the composition rule is included. The same restictions as we have imposed on the specification language will apply to our instance of the composition principle.

3.2.4 Testing Tools

Part of the hypotheses is that the proposed testing strategy can be realized as a computerized tool for automated testing. To validate this we need to do empirical testing, which means the proposed tool should be realized and tried out.

A full implementation is not possible within the scope of an cand.scient. thesis. Instead we build two prototype tools by using functionality in SDL Validator, a part of Telelogic Tau SDL Suite [Telelogic, 2001], and small programs written in the programming language Perl [Schwartz and Christiansen, 1997] to handle SDL Validator and generated test cases:

- Direct testing tool;
- Abadi/Lamport testing tool.

The first tool tests contract decompositions in SDL directly and the second does the testing according to the composition rule. These prototype tools must be seen as experimental, i.e., we build them only in order to conduct experiments on the testing strategies.

Both tools validate the concrete hypothesis of section 3.1.2, while the Abadi/Lamport testing tool validates the first part of the hypothesis of section 3.1.3. If the hypothesis of section 3.1.3 is valid it seems reasonable to expect that the Abadi/Lamport testing tool is more efficient that the Direct testing tool. We investigate this by conducting the same tests on both tools and comparing the results.

3.3 Strategy for Validation of Hypotheses

Testing of the prototype tools have two purposes:

- establish the validity of the testing strategies;

- compare the two strategies in order decide whether the Abadi/Lamport testing tool is more efficient as postulated.

The testing tools are empolyed on a sample of 15 example decompositions; both tools are tested with all examples. Among the examples are both correct and incorrect decompositions. The examples are artificial, but we believe they are representative for a large class of contract decompositions. A more detailed description of the examples is found in section 8.2.

In order to compare the two testing tools, we monitor the following variables:

- whether the example decompositions are validated or not;
- the number of test cases generated and executed;
- the time used for generating and executing test cases;
- the symbol coverage achived during execution of a set of test cases.

Restrictions

To the specifications in the examples we have imposed the following restrictions:

- We only test on time independent behavior.
- The specifications do not contain variables (in a programming language sense).
- We do not specify environments which respond to feedback from components.

3.4 Success Criteria

In order to say that we have been able to verify our hypotheses (or at least not falsified them) we require that the following success criteria are fulfilled:

- We are able to simulate the behavior of the two tools by prototypes that work for the selected examples.
- The two testing tools validate exactly the same examples.
- The correct examples are validated and the incorrect examples are not validated.

In addition we hope to find:

- The Abadi/Lamport testing tool is more efficient that the Direct testing tool.
- The tracability of the Abadi/Lamport testing tool is better than the tracability of the Direct testing tool.

By *more efficient* we mean use of shorter and fewer test cases and less time, without decreasing the "quality" of the tests. By *tracability* we mean the support for finding errors in a not validated decomposition.

Chapter 4

Specifications, Semantics and Composition

In this chapter we describe the specification language used and define semantics. As specification language we use a subset of the *Specification and Description Language* (SDL). This subset is defined in section 4.1.

In section 4.2 we motivate the use of a stream based semantics from the FOCUS method and define general semantics. In section 4.3 the SDL subset is formalized using the defined semantics.

4.1 Specifications

As specification language we use a subset of the *Specification and Description Language* (SDL). The parts of SDL we use are:

- The reserved words

  SYSTEM which specifies the name of an SDL system;

  BLOCK which specifies the name of an SDL block;

  PROCESS which specifies the name of an SDL process;

  SIGNAL which specifies the signals used in a block or process interaction diagram;

  SIGNALLIST which is used for defining sets of signals;

  VIA which is used to specify which channel a signal is sent over;

  NONE which is used for specifying spontaneous transitions in state transition diagrams;

ANY which is used for specifying non-deterministic choices in state transition diagrams;

DCL which is used for specifying local variables in processes;

TRUE and **FALSE** which denote the boolean values;

- the special symbol

  ***** which is used to specify any other input;

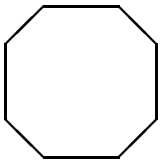
- block interaction digrams with
  - blocks, represented by

  - unidirected channels without delay, represented by unidirected arrows;
  - text boxes for declarations of signals, signallists and variables, represented by

- process interaction diagrams with
  - processes, represented by

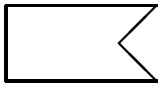
  - unidirected channels without delay, represented by unidirectional arrows;
  - text boxes for declarations of signals, signallists and variables;

- state transition diagrams with
  - start states, represented by

  - states, represented by

– input of signals, represented by



– output of signals, represented by

– choices (boolean and non-deterministic only), represented by

– tasks (assignment only), represented by

– transitions, represented by arrows between states with inputs, outputs, choices and tasks attached to them;

These constructs are more thouroghly described in section 4.3, where we define semantics for this subset of SDL.

4.2 Semantics

As semantic model for the chosen specification language we use infinite timed streams like the underlying semantics of the FOCUS method [Broy and Stølen, 2001]. Most of the definitions of streams and operations on streams are from FOCUS.

The SDL standard provides semantics for SDL. This semantics, however, has several weaknesses and ambiguities, and cannot be considered to be a formal semantics [Hinkel, 1998a]. When we define semantics for SDL in section 4.3 we do our best to model the intentions of the standardized semantics of SDL, but do not consider this semantics beyond that.

There exists state-based semantics for state machines. One example is found in [Pnueli and Shalev, 1991], which proposes a semantics for Statecharts [Harel, 1987]. In such semantics the main objectiv is to describe the state of a system or component for each step in a computation.

Our focus, however, lies on communication and interaction between a system and its environment and between system components. We therefore

adopt a blackbox view and let systems and components be characterized by their input and output rather than their internal states. Also, since we are occupied with distributed systems, we find the notion of global states, that e.g. [Pnueli and Shalev, 1991] uses, not suitable for our work.

4.2.1 Timed Streams

Components communicate by the means of sending streams of messages over directed channels. In addition to messages there is also a time tick (denoted $\surd$) which represent time. An infinite timed stream is a sequence of an infinite number of ticks and possibly an infinite number of messages. Each tick represents the end of a time unit, and for all $j \in \mathbb{N}$ there can be arbitrary finite number of messages between the j'th and the $(j+1)$'th tick. A finite timed stream always ends with a tick, so a timed stream can never be truncated between two messages, which would be to truncate the stream in the middle of a time unit.

Transmission of a message m over a channel c is instantaneous, which means that the output of m and the input of m happens within the same unit of time.

The behavior of a system or a component is defined as a relation between the input streams and output streams of that system or component. Since this is a binary relation, if defines a set of pairs of tuples of streams. This set may be called a *input/output relation* and its element are refered to as *input/output pairs*. An input/output pair can be viewed as a recording of a possible behavior of a system or system componet; as a communication history for that system or component. Since the streams contain infinite many ticks, a communication history is complete and may describe a possible behavior of a non-terminating system. An input/output pair is also refered to as *a behavior*.

A stream is written as a sequence of messages and ticks divided by commas and enclosed in angular brackets ($\langle\ \rangle$). An example stream could be

$$s = \langle m_1, m_2, \surd, m_3, \surd, \surd, m_4, \surd, m_5, \surd \rangle$$

4.2.2 Definitions

This section contains definitions of basic types and operations on streams. Most of the definitions are borrowed from or inspired by FOCUS. Throughout this thesis all logical operators and operations on sets have the usual meaning.

We use the following notation for sets:

- $\mathbb{N}$ is the set of all natural numbers,

$$\mathbb{N} \stackrel{\text{def}}{=} \{0, 1, 2, 3, \ldots\}$$

- $\mathbb{N}_+$ denote the set of all positive natural numbers,

$$\mathbb{N}_+ \stackrel{\text{def}}{=} \mathbb{N} \setminus \{0\}$$

- $\mathbb{N}_\infty$ is the set of natural numbers extended with an element ∞, which represtent a number greater than all natural numbers,

$$\mathbb{N}_\infty \stackrel{\text{def}}{=} \mathbb{N} \cup \{\infty\}$$

- $\mathbb{B}$ is the set of boolean values,

$$\mathbb{B} \stackrel{\text{def}}{=} \{\mathbf{true}, \mathbf{false}\}$$

- 2^A is the powerset of A when A is a set,

$$2^A \stackrel{\text{def}}{=} \{B | B \subseteq A\}$$

For a set of messages D we have:

- $D_\surd$ is the union of D and the time tick,

$$D_\surd \stackrel{\text{def}}{=} D \cup \{\surd\}$$

- D^∞ is the set of all infinite timed streams with messages form D. Mathematically a stream $s \in D^\infty$ is a mapping from positive natural numbers to messages, so

$$D^\infty \stackrel{\text{def}}{=} \{s \in \mathbb{N}_+ \to D_\surd | \forall j \in \mathbb{N}_+ \exists k \in \mathbb{N} : k \geq j \wedge s(k) = \surd\}$$

- D^* is the set of all finite timed streams with messages from D,

$$D^* \stackrel{\text{def}}{=} \bigcup_{n \in \mathbb{N}} ([1 \ldots n] \to D_\surd)$$

  where $[1 \ldots n]$ is an interval of natural numbers.

- D^ω is the set of all timed streams with messages from D,

$$D^\omega \stackrel{\text{def}}{=} D^\infty \cup D^*$$

We define the following operations for manipulating streams:

- ε denote the empty stream, i.e. a stream without any messages or ticks.
- $. : D^\omega \times \mathbb{N}_+ \to D_\surd$ is used to get the messages on a certain place in a stream. If s is a stream then $s.k$ is the k'th message of s.

$$s.k \stackrel{\text{def}}{=} s(k)$$

- $\frown : D^\omega \times D^\omega \to D^\omega$ is the concatenation operator. If s and r are stream, $s \frown r$ is the stream with the messages of s followed by the messages of r. If $s \in D^\infty$ then $s \frown r = s$. If $s = (s_1, s_2, \ldots, s_k)$ and $r = (r_1, r_2, \ldots, r_k)$ are tuples of streams then

$$s \frown r = (s_1 \frown r_1, s_2 \frown r_2, \ldots, s_k \frown r_k)$$

- s^k denotes the stream s repeated k times.

$$s^k = \underbrace{s \frown s \frown \cdots \frown s}_{k \text{ times}}$$

- $\& : D \times D^\omega \to D^\omega$ is used for appending a message to the beginning of a stream

$$m \& s \stackrel{\text{def}}{=} \langle m \rangle \frown s$$

- $\# : D^\omega \to \mathbb{N}_\infty$ denote the length of a stream. If s is a stream then $\#s$ is the number of messages in s.

$$\#\varepsilon \stackrel{\text{def}}{=} 0$$
$$\#(m \& s) \stackrel{\text{def}}{=} \#s + 1$$

- $\sqsubseteq$: $D^\omega \times D^\omega \to \mathbb{B}$ is the prefix relation. If s and r are streams, we have

$$s \sqsubseteq r \stackrel{\text{def}}{=} \exists u \in D^\omega : s \frown u = r$$

- $Ⓢ : 2^{D_\surd} \times D^\omega \to D^\omega$ is the filtering operator. If $M \subseteq D_\surd$ and s is a stream, $M Ⓢ s$ denote the string obtained be removing all occurrences of the messages not in M from s.

$$M Ⓢ \varepsilon \stackrel{\text{def}}{=} \varepsilon$$
$$m \in M \Rightarrow M Ⓢ (m \& s) \stackrel{\text{def}}{=} m \& (M Ⓢ s)$$
$$m \notin M \Rightarrow M Ⓢ (m \& s) \stackrel{\text{def}}{=} M Ⓢ s$$

- $\downarrow: D^\infty \times \mathbb{N}_\infty \to D^\omega$ is used to truncate a stream after a number of ticks.

$$s\downarrow_t \stackrel{\text{def}}{=} \begin{cases} s & \text{if } t = \infty \\ \varepsilon & \text{if } t = 0 \\ r & \text{otherwise, where } r \sqsubseteq s \wedge \#(\{\surd\}\circledS r) = t \wedge r.\#r = \surd \end{cases}$$

If $s = (s_1, s_2, \ldots, s_k)$ is a tuple of streams, we let

$$s\downarrow_t \stackrel{\text{def}}{=} (s_1\downarrow_t, s_2\downarrow_t, \ldots, s_k\downarrow_t)$$

When the order of the streams in a tuple is unrelevant, we sometimes view a tuple of streams as a set of streams and apply set operators to them.

4.2.3 System and Components

In the semantics, systems and system components are specified in the same manner. Whether we in the following operate with system of components or component of sub-components will not make any differece. A system can be viewed a component at the topmost level.

Figure 4.1 shows a component M with input channels i and output channels o. In textual notation we denote this component as $M(i \rhd o)$, where the $\rhd$ is used as a deliminator between the input channels and output channels.

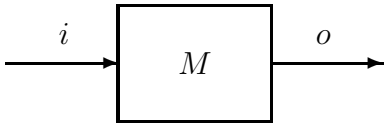
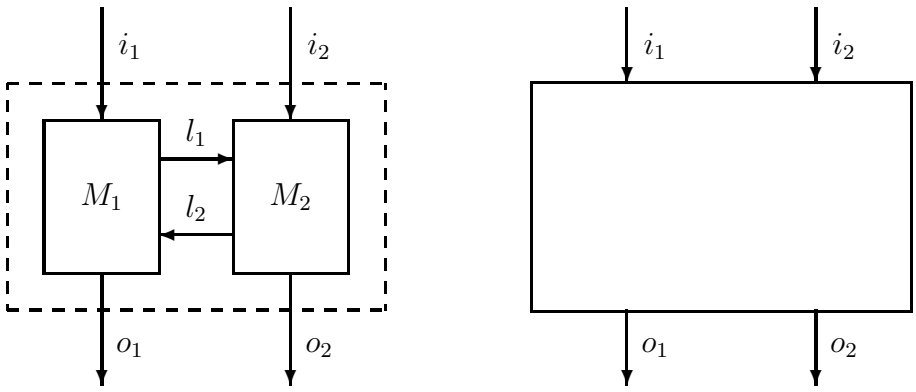


Figure 4.1: Component M

Semantically we think of systems and components as predicates over tuples of streams. If M has n input channels and m output channels and D is the set of messages that can be sent over these channels, we let a predicate

$$[\![M]\!] : (D^\omega)^n \times (D^\omega)^m \to \mathbb{B}$$

denote the semantic meaning of M. $[\![M]\!]$ is usually refered to as the *denotation* of M. This denotation is the input/output relation of M. A specified system or componenet is also refered to as a *specification*.



Glassbox view Blackbox view

Figure 4.2: Two views of composition

4.2.4 Composition

In FOCUS one way of making composite specifications is dataflow diagrams, where boxes represent components and directed arrows represent channels or tuples of channels.

To a composition we adobt either a blackbox or a glassbox view. With the blackbox view we are only able to observe the external behavior of the composition, while we with the glassbox view are able to look inside the composition and observe the internal structure and communication of the composition. Figure 4.2 shows these two views of composition.

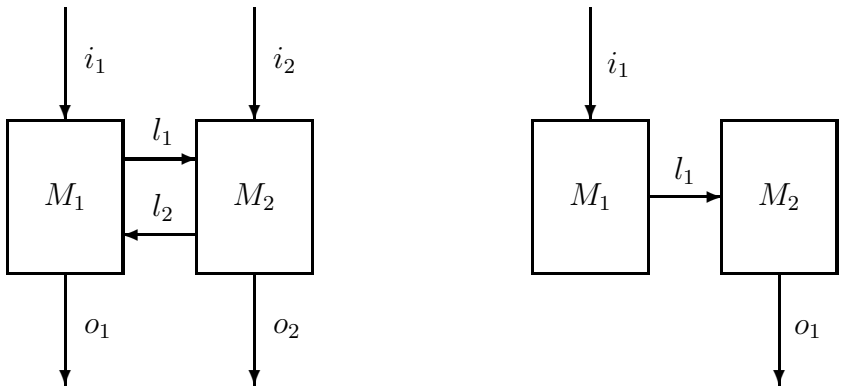
When a composition is made, there are channel that are external and channels that are internal relative to the composition. The streams over internal channels are communication between the combined components and the streams over external channels are the communication between the composition and its environment. If M is a composition, we let i_M and o_M denote the tuples of external input and output streams respectively and let l_M denote the tuple of internal streams.

In textual notation we use the symbols $\otimes$ and $\succ$ to denote composition. $\otimes$ is the most general compositon operator, because it allows the composites to have mutual feedback, i.e., they may be sending messages to each other. If

$$M(i_M \rhd o_M) = M_1(i_{M_1} \rhd o_{M_2}) \otimes M_2(i_{M_2} \rhd o_{M_2})$$

we have

$$l_M = (o_{M_1} \cap i_{M_2}) \cup (o_{M_2} \cap i_{M_2})$$



$M_1 \otimes M_2$ $M_1 \succ M_2$

Figure 4.3: Composition

$$i_M = (i_{M_1} \cup i_{M_2}) \setminus l_M$$

$$o_M = (o_{M_1} \cup o_{M_2}) \setminus l_M$$

Piping

$$M(i_M \rhd o_M) = M_1(i_{M_1} \rhd o_{M_2}) \succ M_2(i_{M_2} \rhd o_{M_2})$$

is the special case of mutual feedback when

$$o_{M_1} = i_{M_2} = l_M \wedge o_{M_2} \cap i_{M_1} = \emptyset$$

Examples of the two types of composition are shown in figure 4.3.

The $\otimes$ operator is cummutative

$$M_1 \otimes M_2 = M_2 \otimes M_1$$

and if the component's sets of input streams are disjoint ($i_{M_j} \cap i_{M_k} = \emptyset$ when $j \neq k$), also assosiative

$$(M_1 \otimes M_2) \otimes M_3 = M_1 \otimes (M_2 \otimes M_3)$$

This requirement on the sets of inputs streams is ensured by disallowing splitting of channels, so that a channel can only connect two components and a channel from the environment can only give input to one component.

Because of these properties of the $\otimes$ operator, there is no need for defining composition of more than two components.

Semantically we interpret composition as logical conjunction. If

$$M = M_1 \otimes M_2$$

and (i, o) is a possible behavior of M, we have:

- i is a tuple of streams on the external input channels of M;
- o is a tuple of streams on the external output channels of M;
- there exists streams $l = (l_1, l_2, \ldots, l_k)$ over the internal channels of M that are possible communications between the components;
- $i \cap l = \emptyset \wedge l \cap o = \emptyset \wedge i \cap o = \emptyset$, and for each $j = \{1, 2\}$

  $$(i_{M_j} \subseteq i \cup l) \wedge (o_{M_j} \subseteq l \cup o)$$

  so within (i, l, o) there are a possible behavior for both M_1 *and* M_2.

The triplet (i, l, o) contains possible behavior for both M_1 and M_2 at the same time. This is necessary because the components are assumed to run in parallel. Because the predicates that denote the components then are true at the same time, we can semantically interpret composition as conjunction.

If we adopt a blackbox view of the composition M, we do not see the internal channels, so in the semantic understanding of blackbox composition we hide the internal streams. The hiding is done with the existential quantifier $\exists$ since it is enough that there exists one possible stream over each of the internal channels.

Hence we get the following definition of composition:

Definition 4.1 (Composition) *If*

$$M = M_1 \otimes M_2$$

with k internal channels of type D, then

$$[\![M]\!] \stackrel{\mathsf{def}}{=} [\![M_1]\!] \wedge [\![M_2]\!]$$

in glassbox view, and

$$[\![M]\!] \stackrel{\mathsf{def}}{=} \exists l_M \in (D^\omega)^k : [\![M_1]\!] \wedge [\![M_2]\!]$$

where

$$l_M = (o_{M_1} \cap i_{M_2}) \cup (o_{M_2} \cap i_{M_2})$$

in blackbox view. □

4.2.5 Refinement

We say that a specification S' *refines* a specification S if all behaviors in $[\![S']\!]$ are contained in $[\![S]\!]$. If a specification S is refined by a specification S', we denote this as

$$S \rightsquigarrow S'$$

The formal definition of the refinement relation is:

Definition 4.2 (Refinement of specifications) *If S and S' are specifications, then*

$$S \rightsquigarrow S' \stackrel{\mathsf{def}}{=} [\![S']\!] \Rightarrow [\![S]\!]$$

□

This definition asserts that S' is a refinement of S if every possible behavior of S' is a possible behavior of S. The use of implication gives the refinement relation the property that if there are possible behaviors of S that are not possible behaviors of S', S' can still be a refinement of S.

This means that if $S \rightsquigarrow S'$, then S' specify the same system or component as S since S' has no "new" behaviors that S has not, but that S is less deterministic that S' (and S' probably closer to implementation than S) because there can be behaviors of S that are not behaviors of S'.

It is worth noticing that this definition only covers refinement of behavior, and not, for instance, refinement of interface. The number of external input and output streams are the same on both side of a refinement.

(An effect of this definition is that e.g. a rock will refine (and implement) all specified systems, because a rock has no behaviors in our interpretation of behavior. $[\![S_{rock}]\!] = \mathbf{false}$, so $([\![S_{rock}]\!] \Rightarrow [\![S]\!]) = \mathbf{true}$ for any specification S and behavior (i, o).)

Sets and predicates are essentially the same thing, so the denotation of a specification can be seen as a set of behaviors (input/output pairs). Implication then correspond to the subset relation, so the refinement $S \rightsquigarrow S'$ can also be defined as

$$[\![S']\!] \subseteq [\![S]\!]$$

$[\![S]\!]$ is true for a behavior (i, o) if and only if $(i, o) \in [\![S]\!]$. The only situations where the refinement does not hold is when $[\![S']\!]$ is true for (i, o), i.e., $(i, o) \in [\![S']\!]$, and $[\![S]\!]$ is false, i.e., $(i, o) \notin [\![S]\!]$.

We use the two notions of denotation and refinement interchangeable.

4.2.6 Safety and Liveness

Requirements of a system or component can be devided into safety properties and liveness properties. Informally safety properties are requirements that can be falsified in finite time, while liveness properties are requirements are that only can be falsified in infinite time.

When we formalize safety and liveness, we get the following definition.

Definition 4.3 (Safety and liveness properties) *A predicate* $P : (D^\omega)^n \times (D^\omega)^m \to \mathbb{B}$ *is a safety property iff*

$$\forall i, o : [\forall t \in \mathbb{N} : \exists z \in (D^\omega)^m : z{\downarrow_t} = o{\downarrow_t} \wedge P(i, z)] \Rightarrow P(i, o)$$

A predicate $P : (D^\omega)^n \times (D^\omega)^m \to \mathbb{B}$ *is a liveness property iff*

$$\forall i, o : \forall t \in \mathbb{N} : \exists z \in (D^\omega)^m : z{\downarrow_t} = o{\downarrow_t} \wedge P(i, z)$$

□

4.2.7 Interleaving

Specifications can be interpreted to be interleaving or non-interleaving. If a specification is non-interleaving it allows actions or events to happen simultaneously. Interleaving specifications do not allow this, so that all actions or events must by ordered.

Our semantics are interleaving; to make it non-interleaving would add complexity. According to [Hoare, 1985, page 24] it is possible to manage well without non-interleaving. In [Hoare, 1985] processing and sending messages are carried out in no-time, operations that need a noticeable amount of time are described as two actions; one that starts the operation and one that ends it.

As we explain in section 4.3.4, all input streams to a component are merged into a queue. If two merged streams both have messages between the j'th and the $(j + 1)$'th time tick we have no way of knowing their ordering, so their order in the queue is arbitrary.

All communication between components is done by interchanging messages. The only shared variables between two componets are their common streams, so problems concerning shared variables are prevented. Writing (i.e., outputting) to a stream does not affect the data (i.e., messages) already in the stream, and cannot prevent another component reading data (i.e., receiving messages) from the stream.

4.2.8 Causality

Generally we want to specify systems which follows our intuitve understanding of causalty between input and output, i.e., that a system is not allowed to produce output based on input it has not yet received. Executable specifications and systems cannot fail to be causal. It is impossible to realize a system that does computations on input it has not received.

In order to avoid specifications that are not caual, we add causalty constraints to specifications. We distinguish between two types of causalty; *weak causalty* and *strong causalty*. Both are formalized in the below definition.

Definition 4.4 (Causalty) *A predicate* $P : (D^\omega)^n \times (D^\omega)^m \to \mathbb{B}$ *is weakly causal iff*

$$\forall x, y : \forall t \in \mathbb{N} : x\downarrow_t = y\downarrow_t \Rightarrow \{o\downarrow_t \mid P(x, o)\} = \{o\downarrow_t \mid P(y, o)\}$$

A predicate $P : (D^\omega)^n \times (D^\omega)^m \to \mathbb{B}$ *is strongly causal iff*

$$\forall x, y : \forall t \in \mathbb{N} : x\downarrow_t = y\downarrow_t \Rightarrow \{o\downarrow_{t+1} \mid P(x, o)\} = \{o\downarrow_{t+1} \mid P(y, o)\}$$

□

4.3 Semantics for SDL

In this section the subset of SDL defined in section 4.1 is formalized in the semantics described above. This formalization is based on [Holz and Stølen, 1995], [Hinkel, 1998a] and [Hinkel, 1998b].

As mentioned in section 4.2, the SDL standard contains a semi-formal semantics for SDL. We refer to this as the *SDL semantics*. In this formalization we do our best to model the intentions of the SDL semantics.

The main structures of SDL specifications are systems, blocks and processes (see section 2.1). By the expression *SDL specification* we refer to an SDL system. For each SDL specification we assosiate a FOCUS specification, so they are semantically interpreted as predicates.

4.3.1 Systems and Blocks

SDL systems are specified by block interaction diagrams and SDL blocks are specified by block interaction diagrams or process interaction diagrams. Both types of interaction diagrams are dataflow diagrams, and are used for

specifying the static structure of SDL specifications. Block interaction diagrams consist of SDL blocks connected by channels, while process interaction diagrams consist of SDL processes connected by channels.

For each interaction diagram we assosiate a FOCUS dataflow diagram. SDL systems and blocks are then semantically interpreted as predicates where the semantics rule of composition defined in section 4.2.4 apply.

4.3.2 Signals

SDL processes communicate by interchanging signals. In the semantics each SDL signal is assosiated with a FOCUS message. Signals used in an SDL specification must be declared in a textbox at the same level as or an outer level from where they are used. The reserved word **SIGNALLIST** can be used to group signals in sets. The textbox

SIGNAL m_1, m_2, m_3;
SIGNALLIST $M = m_1, m_2, m_3$;

has the semantic meaning that m_1,m_2 and m_3 are messages in the specification the textbox appears and

$$M = \{m_1, m_2, m_3\}$$

is a set of messages.

4.3.3 Channels

For each channel in an SDL specification we assosiate a FOCUS channel. We can do this since we have restricted the SDL specifications to only contain directed channels with instantaneous signal passing. This correspond to FOCUS channnsls, which always are directed and without delay.

In SDL the signals that can be passed over a channel are listed in brackets ([ ]) attached to the channel. In this list there may be signals and signallists. Signallists are emphasized by paranteses. Semantically the list attached to a channel defines the type of the channel. A channel

$$\xrightarrow[[(M), m_4]]{d}$$

is of the type

$$D = M \cup \{m_4\}$$

and for a stream s over d we have $s \in D^\omega$.

4.3.4 Processes

SDL processes are state transition diagrams (state machines) and used for specifying the behavior of SDL specifications. These state transition diagrams are formalized by the means of stream processing functions.

In the SDL semantics, processes have a number of implicit features. In our formalization we express these features explicit in the below FOCUS model for SDL processes. This section also provides interpretations of time, safety and liveness, and causalty for SDL processes.

FOCUS Model of SDL Processes

An SDL process PR, shown in figure 4.4, is modeled by the FOCUS dataflow diagram shown in figure 4.5. In the SDL semantics signals over the input channels to an SDL process are inserted into an input queue from which the process receives the signals. The signals are put into the queue in the order the arrive. In our semantics this means that if a message m_1 arrive over i_1 after j time ticks and m_2 arrive over i_2 after k time ticks, m_1 is placed ahead of m_2 in the queue if $j < k$. If $j = k$ we have no way of knowing which message arrived first and they are arbitrarily ordered in the queue. This queue is modeled by a special component FM (*fair merge*), which is defined in section 4.3.5.

Spontaneous transitions in SDL are specified by input of a special signal **NONE**. The componenet NG outputs a stream of messages *none* with arbitrary many tick between each *none*. According to [Hinkel, 1998b], the denotation of NG can be defined as

$$[\![NG(\triangleright z)]\!] \stackrel{\mathsf{def}}{=} \mathbf{true}$$

where

$$z \in \{none\}^\infty$$

since all streams of only ticks and *none*s are possible behaviors of NG. This means that also the stream of zero *none*s is a legal behavior of NG.

The component PR' represents the explicit specified behavior of PR, i.e., the behavior specified by the state transition diagram of the SDL process PR.

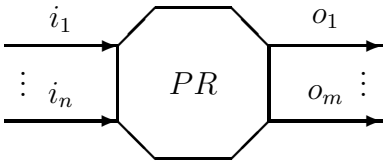


Figure 4.4: SDL process PR

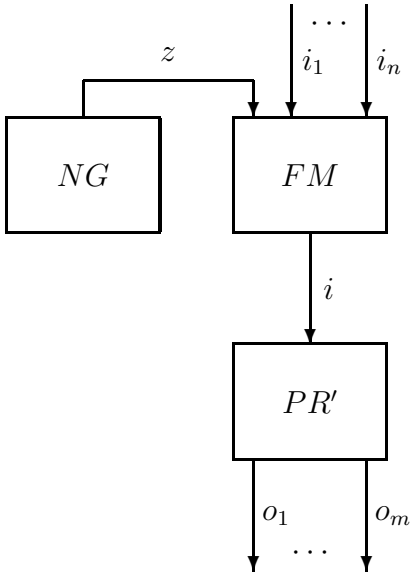


Figure 4.5: FOCUS model of PR

Denotation of SDL Processes

When we define the denotation of an SDL process, the state transition diagram of the SDL process is translated to a set of equations on stream processing functions. To each state we assosiate a function (with the same name as the state) and each transition between states is translated to an equation. How this translation is done is shown in section 4.3.7 by the means of nine example cases.

All SDL processes have exactly one start state. The start state is represented by a special symbol and is not named in the diagram. For each SDL process PR we call this state $start_{PR}$ and assert that control always starts in this state.

We assume that the input and output streams have the following types

$$i_1 \in I_1^\omega;\ i_2 \in I_2^\omega; \ldots; i_n \in I_n^\omega$$

$$o_1 \in O_1^\omega;\ o_2 \in O_2^\omega; \ldots; o_m \in O_m^\omega$$

We then have that $i \in I^\omega$ where

$$I = I_1 \cup I_2 \cup \cdots \cup I_n \cup \{none\}$$

We use the short hand notation

$$i_{PR} = i_1, i_2, \ldots, i_n$$

$$o_{PR} = o_1, o_2, \ldots, o_m$$

for the input and output streams, and

$$I^{\omega}_{PR} = I^{\omega}_1 \times I^{\omega}_2 \times \cdots \times I^{\omega}_n$$

$$O^{\omega}_{PR} = O^{\omega}_1 \times O^{\omega}_2 \times \cdots \times O^{\omega}_m$$

for the types.

Because SDL processes are time independent (except for the *timer* construct we have omitted from the subset), the stream processing functions in the translation of an SDL state transition diagram work on time independent streams. We let

$$\bar{\imath} = I \circledS i$$

$$\overline{o_{PR}} = O_1 \circledS o_1, O_2 \circledS o_2, \ldots, O_m \circledS o_m$$

denote i and o_{PR} with the time ticks removed.

We assume that PR has l control states $s_0, s_1, \ldots, s_{l-1}$, and T_{PR} is the set of equations on time independent stream processing functions obtained by the semantic translation of the state transition diagram of SDL process PR.

The denotation of PR' is defined to be

$$[\![PR']\!] = (\overline{o_{PR}} = start_{PR}(\bar{\imath}))$$
$$\textbf{where } start_{PR}, s_0, s_1, \ldots, s_{l-1} \textbf{ so that } \forall \bar{\imath}, \overline{o_{PR}} : \textstyle\bigwedge_{tr \in T_{PR}} tr$$

The **where-so that** clause defines the functions $start_{PR}, s_0, s_2, \ldots, s_{l-1}$ to be the functions that solve the conjunction of the equations in T_{PR}.

With the denotation $[\![PR']\!]$, PR' is not causal, so we have to ensure that the implicit causalty of SDL process PR is maintained in the translation to time independent functions.

In order to do this, we use the concept of *strategies* from FOCUS. We adopt the view that a specified component is playing a game. In this game the component receives input and produces output, and it wins if the resulting input/output pair satisfies its specification. The component plays according to a strategy, and if the component can always win a game, independent of the input it receives, it has a *winning strategy*.

Formally a strategy for an SDL process is a function $\tau \in I^{\omega} \to O^{\omega}_{PR}$, and is a winning strategy if

$$\forall i \in I^{\omega} : [\![PR']\!]^{[o_{PR}]}_{\tau(i)}$$

where $[\![PR']\!]^{[o_{PR}]}_{[\tau(i)]}$ means $[\![PR']\!]$ with all occurrences of o_{PR} substituted by $\tau(i)$. A strategy is weakly causal if

$$\forall x, y \in I^{\omega}; t \in \mathbb{N} : x\downarrow_t = y\downarrow_t \Rightarrow \tau(x)\downarrow_t = \tau(y)\downarrow_t$$

and strongly causal if

$$\forall x, y \in I^{\omega}; t \in \mathbb{N} : x\downarrow_t = y\downarrow_t \Rightarrow \tau(x)\downarrow_{t+1} = \tau(y)\downarrow_{t+1}$$

Weakly and strongly causal strategies are defined by using respectively $\stackrel{\mathsf{W}}{\rightarrow}$ and $\stackrel{\mathsf{S}}{\rightarrow}$ in their signatures. For an SDL process PR we define the set of causal strategies to be

$$str_{\mathsf{T}}(PR') \stackrel{\mathsf{def}}{=} \{\tau \in I^{\omega} \stackrel{\mathsf{T}}{\rightarrow} O^{\omega}_{PR} | \forall i \in I^{\omega} : [\![PR']\!]^{[o_{PR}]}_{[\tau(i)]}\}$$

In order to add causalty to PR' we define a new specification PR'', which has the denotation

$$[\![PR'']\!] = \exists \tau \in str_{\mathsf{S}}(PR') : \tau(i) = o_{PR}$$

The full denotation of the SDL process PR becomes

$$[\![PR(i_{PR} \rhd o_{PR})]\!] = [\![(NG(\rhd z) \otimes FM(z, i_{PR} \rhd i)) \succ PR''(i \rhd o_{PR})]\!]$$

Non-determinism

There are two ways of specifying non-determinism in SDL; input of **NONE** and choices with the reserved word **ANY**. Input of **NONE** is modeled by the *NG* component described above.

An **ANY**-choice in a transition asserts that all of the branches of the choice are enabled and any of them can be followed. This non-deterministic choice is modeled with a special kind of variable called *oracle*. An oracle is an arbitrary sequence of natural numbers which predicts the non-deterministic choices made in the future.

We need oracles because we define the denotation of an SDL process by the conjunction of equations on stream processing functions. Non-determinism could be modeled by defining the denotation by use of disjunction instead of conjunction. We feel oracles are more natural since SDL is a specification language that originally was deterministic and later got non-deterministic features added to it. In addition, oracles makes it easier to express fairness constraints.

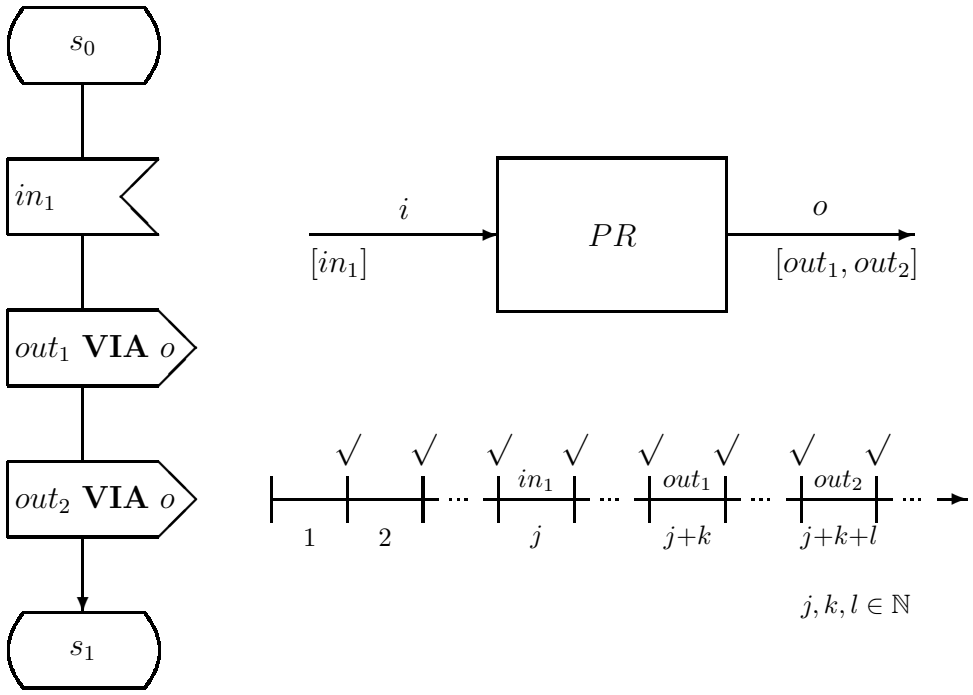


Figure 4.6: Interpretation of time in an SDL transition

Time

SDL has an ambiguous notion of time [Hinkel, 1998b]. In our semantics we follow [Hinkel, 1998b] and make the assumption that time always passes. There is nothing peculiar about this assumption; it is rather an intuitive interpretation of time. Every implementation of an SDL specification will have the property that time is running while it is executed.

Message passing is instantaneous, so there is no delay between output and input of a message. There is delay when an SDL process is in a control state waiting for input, and between events (e.g. input and output) in a transition (we make some exceptions in section 5.3.2). This delay is undefined, i.e., we know time is passing, but not how much. Figure 4.6 (from [Hinkel, 1998a]) illustrates how events in a transition in an SDL Process PR are spread along the time axis.

Safety and Liveness of SDL Specifications

SDL processes are executable state machines, and can specify arbitrary computational safety properties. They specify which transition and input and output events are allowed to happen when in a given state. Any other event

that occure is a violation of the specification. When an event has occured it necessarily has occured within finite time, so any state machine can be violated in finite time.

Arbitrary liveness properties cannot be expressed explicitly by state machines. There are for instance no way of specifying that a state machine eventually will reach a specific state or produce a specific output. It is however possible to let state machines to have implicit liveness. A way to do this is to say that a state machine are not allowed to stay in a state forever if a transition from that state is enabled and remains enabled forever. For example the in semantics for Statecharts in [Pnueli and Shalev, 1991] the assumption is made that all enabled transitions are carried out. The same assumption is also often made about computer programs, although not necessarily explicitly stated. This assumption may be called *weak fairness*, as opposed to *strong fairness* which states that an event cannot be enabled infinitely many times without happening. For example [Abadi and Lamport, 1995] uses the notions of weak and strong fairness.

To our SDL specifications we impose a weak fairness constraint which assert that a transition cannot stay enabled forever without happening and that time always proceeds. We do not consider liveness in the further discussion. Leaving out liveness allow us to make simplifications to the semantics and allow us to make a simpler composition rule than we would otherwise need.

Causalty of SDL Processes

Since SDL specifications are executable they are bound to be causal. We therefore impose causalty constraints to the semantics of SDL specifications.

Because we focus on time independent specifications (delays are undefined) the time units can be arbitrarily small. Small time units will not make restrictions to what we are able to specify; time units are a way of ordering events along the time axis, and the ordering cannot be violated by dividing the time units into smaller time units. If the time units are sufficiently small, it can be ensured that an input and an output as response to the input do not occure within the same time unit. Because of this, an SDL process is strongly causal if the time units are sufficiently small.

4.3.5 Fair Merge

A *fair merge* component FM which merges two timed streams i_1 and i_2 into i is defined as

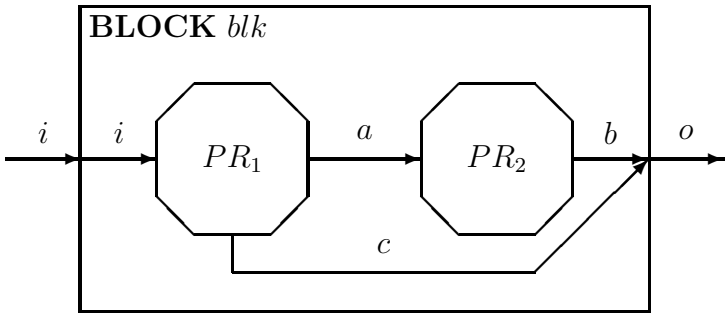


Figure 4.7: SDL block with merge at gate

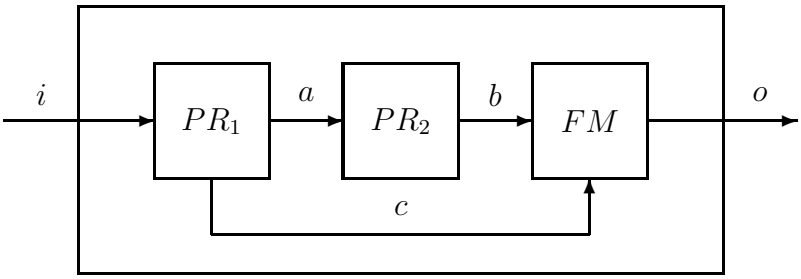


Figure 4.8: FOCUS model of merge at gate

$$
\begin{array}{ll}
\multicolumn{2}{l}{[\![FM(i_1, i_2 \rhd i)]\!] = \exists or \in \{1,2\}^\infty : i = fm[or](i_1, i_2)} \\
\multicolumn{2}{l}{\textbf{where } fm \textbf{ so that } \forall i_1, i_2, i, or :} \\
 & fm[or](\surd \& i_1, \surd \& i_2) = \surd \& fm[or](i_1, i_2) \\
\wedge & fm[or](m_1 \& i_1, \surd \& i_2) = m_1 \& fm[or](i_1, \surd \& i_2) \\
\wedge & fm[or](\surd \& i_1, m_2 \& i_2) = m_2 \& fm[or](\surd \& i_1, i_2) \\
\wedge & fm[1\& or](m_1 \& i_1, m_2 \& i_2) = m_1 \& fm[or](i_1, m_2 \& i_2) \\
\wedge & fm[2\& or](m_1 \& i_1, m_2 \& i_2) = m_2 \& fm[or](m_1 \& i_1, i_2)
\end{array}
$$

where *or* is an oracle and m_1 and m_2 denote any messages. Merges with more than to input streams can easily be defined in the same manner.

The *fair merge* is assumed to be instantaneous, i.e., there is no delay between input and output.

4.3.6 Merge at Gate

The place where a channel meets the bounderies of an SDL block is called a *gate*. In some situations output channels from processes in a block may meet at a gate, as the channels b and c in figure 4.7. Semantically we interpret

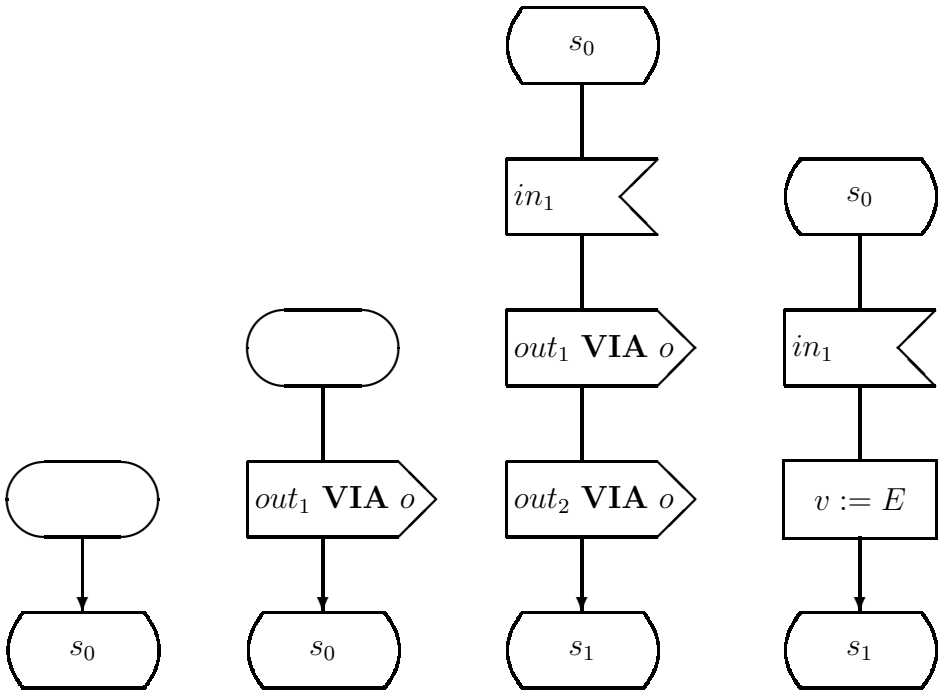


Figure 4.9: Transition cases 1-4

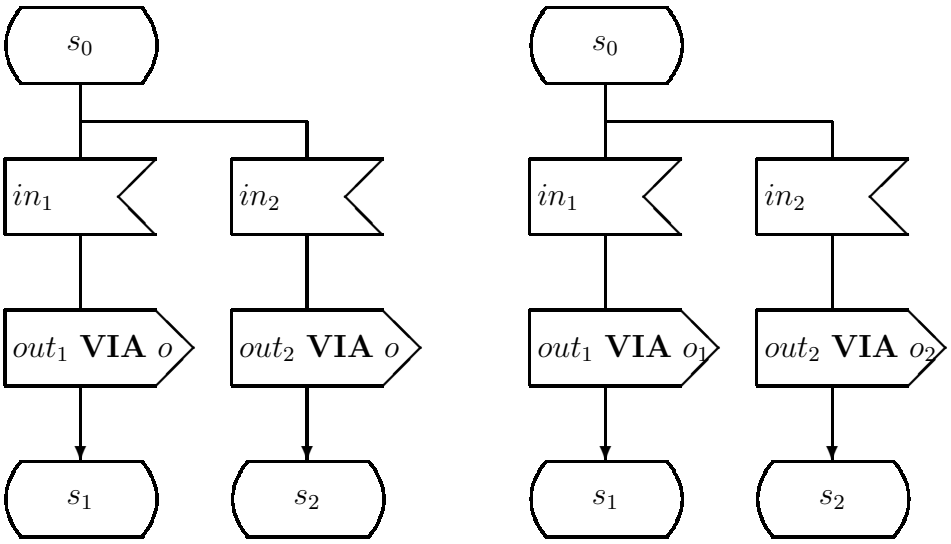
this as there being a *fair merge* component at the gate. A FOCUS model of the SDL block *blk* is shown in figure 4.8.

4.3.7 Translation of SDL State Transition Diagrams

Figures 4.9 to 4.12 show nine examples of transitions in SDL state transition diagrams. In all examples we assume that the input stream is named i. The conjunction of these examples should give a good picture of how the translation from SDL state transition diagrams to equations on stream processing functions is done. In these translations all stream processing functions are time independent.

States and Transitions

The cases in figure 4.9 are all single transitions, and are therefore translated into single equations.



Case 5 Case 6

Figure 4.10: Transition cases 5 and 6

Case 1 is a transition form the start state to the first ordinary state and is translated to

$$start_{PR}(\bar{\imath}) = s_0(\bar{\imath})$$

Case 2 is also a transition from the start state, but with output of a message:

$$start_{PR}(\bar{\imath}) = out_1 \& s_0(\bar{\imath})$$

Case 3 is a transition with input of in_1 and output of two messages:

$$s_0(in_1 \& \bar{\imath}) = \langle out_1, out_2 \rangle^\frown s_1(\bar{\imath})$$

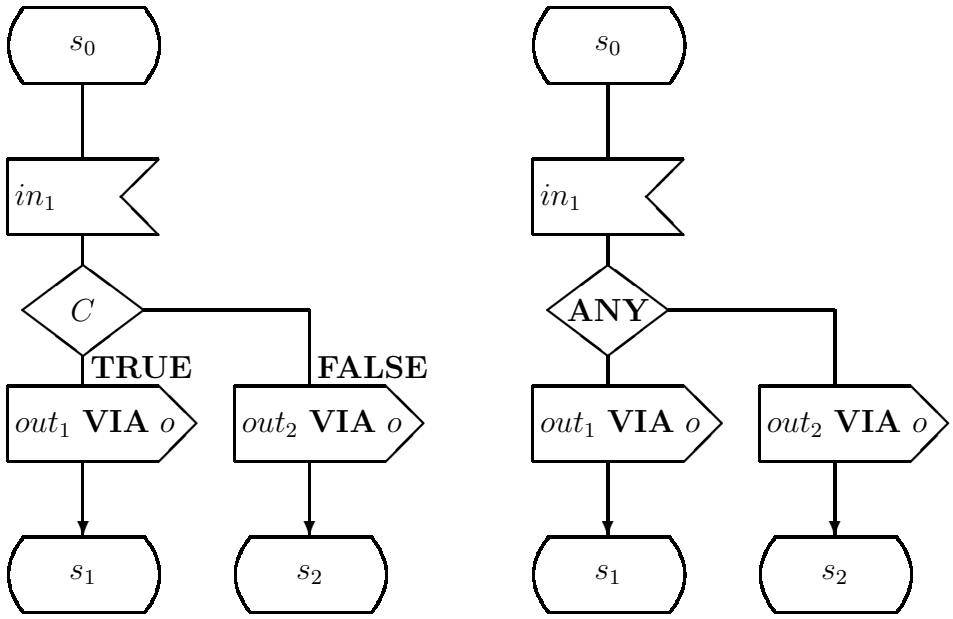
Case 4 is a transition with input of in_1 and then a task with assignment of an expression E to a local variable v. We translate this to

$$s_0[\sigma](in_1 \& \bar{\imath}) = s_0[\sigma[v := E]](\bar{\imath})$$

where σ is a list of local variable and $\sigma[v := E]$ denotes the list σ with v updated by E.

Both cases in figure 4.10 consist of two transitions, and are therefore translated into two equations.

Case 5 is translated to



Case 7 Case 8

Figure 4.11: Transition cases 7 and 8

$$s_0(in_1 \& \bar{\imath}) = out_1 \& s_1(\bar{\imath})$$
$$s_0(in_2 \& \bar{\imath}) = out_2 \& s_2(\bar{\imath})$$

Case 6 has two output channels and the transitions do output to different channels:

$$s_0(in_1 \& \bar{\imath}) = (\langle out_1 \rangle, \varepsilon) \frown s_1(\bar{\imath})$$
$$s_0(in_2 \& \bar{\imath}) = (\varepsilon, \langle out_2 \rangle) \frown s_2(\bar{\imath})$$

Choices and ANY

Figure 4.11 shows transitions with choices. Each of these transitions are translated to a single equation, were we use **if**-operators (with the usual semantics) to denote the choice.

Case 7 is a boolean choice and is translated to

$$s_0(in_1 \& \bar{\imath}) = \begin{array}{l} \textbf{if } C \textbf{ then } out_1 \& s_1(\bar{\imath}) \\ \textbf{else } out_2 \& s_2(\bar{\imath}) \end{array}$$

where C is a boolean condition.

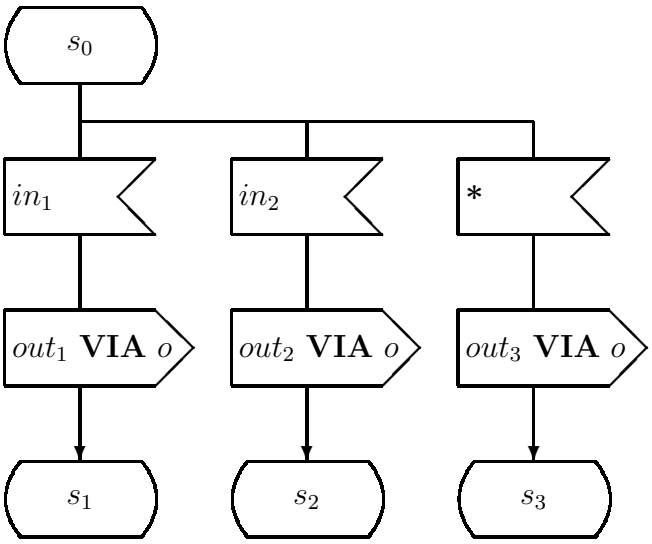


Figure 4.12: Transition case 9

Case 8 contains a non-deterministic choice. This is resolved by the means of an oracle *or*. The transition is translated to

$$s_0[r\&or](in_1\&\bar{\imath}) = \begin{array}{l}\textbf{if } r = 1 \textbf{ then } out_1\&s_1[or](\bar{\imath}) \\ \textbf{elseif } r = 2 \textbf{ then } out_2\&s_2[or](\bar{\imath})\end{array}$$

and the denotation of the SDL process is quantified by

$$\exists or \in \{1,2\}^\infty$$

at the outermost level.

Asterisk Input and Implicit Transition

Case 9 has three transitions where one of them are triggered by the asterisk (*) input. This is interpreted as "any other input", and is closely related to the *implicit transition*.

If I is the set of all possible input to the SDL process (the type of the input queue), then "any other input" in state s_0 in *Case 9* must be input from the set $I \setminus \{in_1, in_2\}$. The asterisk input does however not apply to the *none* message, so the translation is

$$s_0(in_1\&\bar{\imath}) = out_1\&s_1(\bar{\imath})$$
$$s_0(in_2\&\bar{\imath}) = out_2\&s_2(\bar{\imath})$$
$$m \in I \setminus \{in_1, in_2, none\} \Rightarrow s_0(m\&\bar{\imath}) = out_3\&s_3(\bar{\imath})$$

Implicit tranisiton is a transition which resambles the "any other input" tranistion. In all SDL processes there is an implicit transition for each state except the start state. The implicit transition processes any input that does not enable any transition in the current state of the SDL process and brings control to the same state without carrying out any other tasks.

For each state s_k we assosiate a set M_{s_k} of all messages that enables transitions when control is in state s_k. To model the implicit transition, an equation

$$m \in I \setminus M_{s_k} \Rightarrow s_k(m\&i) = s_k(i)$$

is added for each state s_k.

Notice that for *Case 9*, $I \setminus M_{s_0} = \{none\}$. If transitions from a state s_k include both a transition with input of *none* and a transition with asterisk input, then $I \setminus M_{s_k} = \emptyset$.

4.3.8 PID and SENDER

In addition to the defined SDL subset, we define semantics for the SDL type **PID** and the special expression **SENDER**. (For explanation see section 7.3.2).

Every process in an SDL specification has a unique *process instance identifier* (pid). Variables that hold these pids are of the predefined type **PID**.

Semantically we interpret the type **PID** as a subset *PID* of $\mathbb{N}$. We let $\mathcal{P}$ be the set of all processes in an SDL specification (system), and define a function

$$.pid : \mathcal{P} \rightarrow \mathbb{N}$$

which for each process $P \in \mathcal{P}$ returns the pid for P. The type *PID* is then defined as

$$PID \stackrel{\text{def}}{=} \{n \in \mathbb{N} | \exists P \in \mathcal{P} : P.pid = n\}$$

Since all processes have a unique pid we add the constraint

$$\forall P \in \mathcal{P} : \exists n \in PID : P.pid = n \wedge (\forall Q \in \mathcal{P} : P \neq Q \Rightarrow n \neq Q.pid)$$

Every message sent between processes carry with them the the pid of the process from which they are sent. We interpret this as hidden parameter of the message, and define a *.pid* function for messages as well.

$$.pid : M \rightarrow \mathbb{N}$$

where M is the set of all messages in a specification.

In SDL processes the reserved word **SENDER** can be used to get the pid of the last processed signal. The only interesting use of **SENDER** (at least for us) is in the transition tiggered by the signal to which **SENDER** apply. When we do the semantic translation of a transition in which **SENDER** is used, we merely substitute **SENDER** with the function *.pid* applied to the message processed in the resulting equation.

An example in which this is used can be seen in the definition of the *Switch* component in section 7.3.2.

Chapter 5

Contract Oriented Specifications

In this chapter we introduce the format of contract oriented specifications we use in this thesis. In section 5.1 we describe the notion of contract oriented specifications and formalize their semantics. In section 5.2 behavioral refinement of contract specifications is discussed. Section 5.3 provides a scheme for making contract oriented specifications with SDL as specification language.

5.1 Formalization of Contract Oriented Specifications

In the contract oriented style a component C is specified by an assumption A and a guarantee G. Together they form the specification S of C, and we write $S = (A, G)$.

Contract oriented specifications are based on the principle that a specified component should fulfil the guarantee as long as the assumption is met. Informally the relationship between A and G is:

- A describes the behavior of the relevant part of the environment under which C is required work as specified;
- G describes the behavior guaranteed by C when the environment behaves according to A.

The environment of a component consists of other components in the system it belongs, and possibly the external environment of the whole system. It is obvious that A cannot specify the whole environment of C, since this would be a specification of the rest of the system and the environment. When we say that A specifies the relevant part of C's environment, we mean that A

specifies the environment from C's point of view. In practice A will mainly specify the format of the input to C.

G specifies the functionality of C. If C should be specified with a tuple of input channels i and a tuple of output channels o, then G must have the same input channels i and output channels o. Since A specifies the behavior of the relevant part of C's envioronment, the output of A must be i. Symmetrically the input of A must be o, since C should be allowed to interact with its environment. (In section 5.3 we make some simplifications in this respect.)

This relationship is shown graphically as a dataflow diagram in figure 5.1.

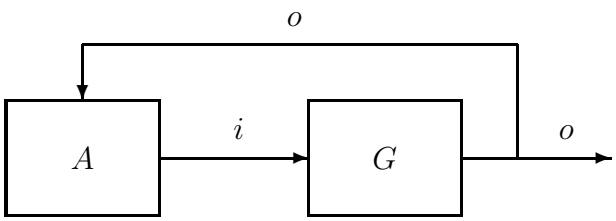


Figure 5.1: Relationship between A and G

A component will typically give feedback to its environment and the environment will be able to behave according to this feedback. The environment of a component will usually consist of other components and we must expect components in a composite system to be mutual dependent upon the messages they send to each other. A component may even be able to send messages to itself. This may lead to circularities that make reasoning about composition of parallel components a non-trivial matter.

In order to make reasoning about specifications with this kind of circularies easier, we follow [Abadi and Lamport, 1995] and add the requirement to the contract specifications that the guarantee G of a specification S must hold as long as the assumption A holds and at least one step longer. Since we use a semantics based on timed streams, this step will be equal to one time unit. When we formalize the notion of contract specifications, this new requirement must be present.

A realized system is always causal, and some delay between input and output must be assumed due to use of computational time. Because of this, the required one time unit delay in the guarantee of a specification does not restrict the class of realizable systems we are able to specify.

5.1.1 Definition of Contract Specifications

A component C with n input channels and m output channels of type D is specified by a contract specification

$$S = (A, G)$$

where A and G are predicates over timed streams of type D^ω:

$$A : (D^\omega)^m \times (D^\omega)^n \to \mathbb{B}$$

$$G : (D^\omega)^n \times (D^\omega)^m \to \mathbb{B}$$

For a contract specification $S = (A, G)$ we define its denotation $[\![S]\!]$:

Definition 5.1 (Denotation of contract specification) *If S is a contract specification $S = (A, G)$ of a component with n input streams and m output streams of type D, then*

$$[\![S]\!] \stackrel{\text{def}}{=} \forall t \in \mathbb{N} : A(o{\downarrow_t}, i{\downarrow_t}) \Rightarrow G(i{\downarrow_t}, o{\downarrow_{t+1}})$$

□

This definition captures both the requirement that the guarantee should hold as long as the assumption holds and the "one step longer" requirement.

The use of implication in the definition captures the fact that we do not specify the behavior of the component when the assumption fails. In the case where the assumption is false at time t, $[\![S]\!]$ is trivially fulfilled at any time $t' > t$, no matter what the guarantee specifies. We interpret this as the specified component starts to produce chaos, i.e., any possible output, after the "plus one"-step.

In a way the specification is closed since the assumptions on the context of the specified component are made explicite and the component is specified under these assumptions. A componenent may implement a specification even if the assumption of the specification is false in the context. This is not remarkable. If the environment of a component does not meet the assumptions this does not necessarily mean that there is something wrong with the component.

5.2 Behavioral Refinement of Contract Specifications

In section 4.2.5 we define a refinement relation $\rightsquigarrow$, and state that in our definition this relation only applies to refinement of behavior. There are two ways of making a legal behavioral refinement of contract specifications [Abadi and Lamport, 1990]:

1. Strengthening the guarantee, i.e, making it more specific (less general).
2. Weakening the assumption, i.e., making it more general (less specific).

By this we mean that a refinement

$$(A, G) \rightsquigarrow (A', G')$$

is a legal behavioral refinement if

1. $G' \Rightarrow G$
2. $A \Rightarrow A'$

This correspond to, e.g., strengthening of a Hoare triple in [Dahl, 1992], but is not the most general formulation of the principle. In [Abadi and Lamport, 1990] the form of the principle is:

$$(A, G) \rightsquigarrow (A', G')$$

if

$$A \Rightarrow A'$$
$$A \wedge G' \Rightarrow A \wedge G$$

In [Abadi and Lamport, 1995] a rule is presented, which has the form:

$$(A, G) \rightsquigarrow (A, G')$$

if

$$A \wedge G' \Rightarrow G$$

Strenghtening is done by allowing fewer behaviors, i.e., by reducing non-determinism. Weakening is done by allowing more behaviors, i.e., by increasing non-determinism.

The result is that weakening the assumption reduces the number of behaviors with chaos (which is legal behavior) and strengthening the guarantee makes a specification more concrete (and closer to implementation). Both ways of making behavioral refinement reduce the number of possible behaviors, which is in corresondence with the use of subset relation in the semantics of refinement. This can be illustrated by figures 5.2 and 5.3 [den Braber, 2001].

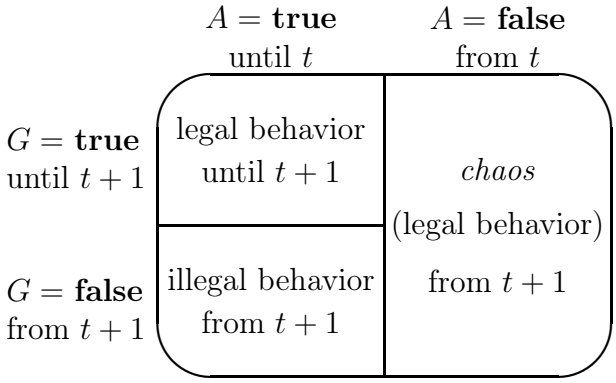


Figure 5.2: Behavior in contract specifications

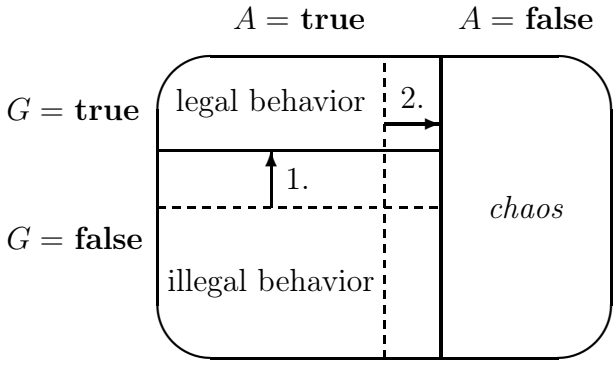


Figure 5.3: Behavioral refinement of contract specifications

5.3 Contracts in SDL

In this section we present a scheme for specifying contract specifications with SDL as specification language, i.e., an SDL representation of $[\![(A, G)]\!]$. This representation of $[\![(A, G)]\!]$ is specified as a special SDL block AG. The goal is to get an executable representation of contract specifications which fulfil the equality

$$[\![AG]\!] = [\![(A, G)]\!]$$

The semantic relation between AG and (A, G) is discussed in section 5.3.6.

This scheme is made with functional testing of the specifications in mind. We do not try to make general guidelines of how to represent contracts in SDL. The purpose of this representation of contracts in SDL is to do experimental testing of the proposed testing strategies.

Some pragmatism is deployed and the result is probably not the most general way of making contract specifications with SDL. We do however account for the pragmatic features when we formalize the SDL contract.

5.3.1 The AG Block

When we specify contracts in SDL, the assumption A and the guarantee G are both specified by SDL processes. In the definition of contract specifications, A is said to describe the behavior of the environment of the component specified by $S = (A, G)$. As a starting point, A is specified in a simple form where it only show which messages the environment is allowed to send over which channels when in a given state. In this simple form the transitions of each state in A will look like in figure 5.4.

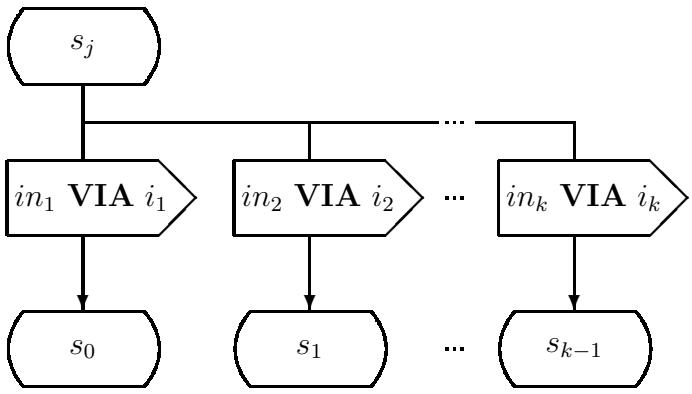


Figure 5.4: A in simple form

Notice that this is a violation of the SDL syntax. In an SDL process all transitions, except the transition from the start state, must begin with an input symbol. It is also worth noticing that figure 5.4 is meant as a general picture; it shows one of possibly many states in A, and we allow $i_r = i_p$ and $s_r = s_p$ for $r \neq p$.

Assumptions of this form represent a restriction of the expressiveness of the contract specifications; only a finite number of possible input messages can be specified, behavior depending on the computation history cannot be specified, and feedback is not taken into account. This means that the class of

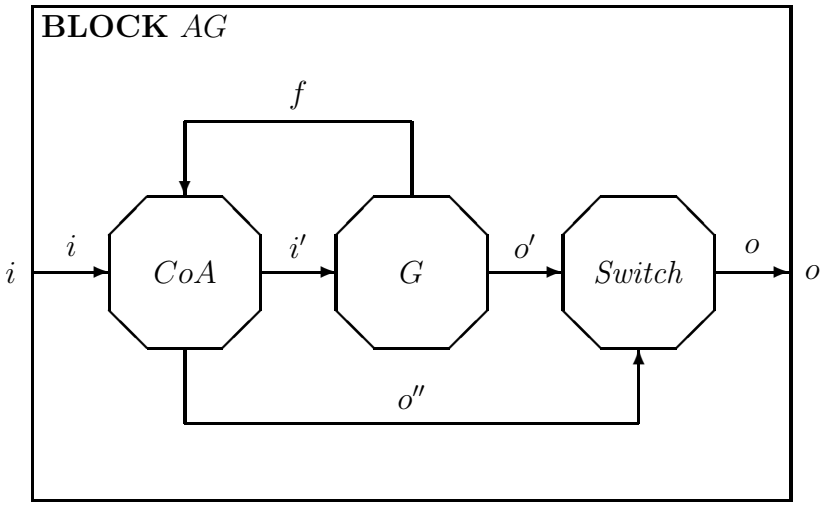


Figure 5.5: SDL block for contract specifications

contract specifications expressable by our scheme for SDL contracts is smaller than (but a proper subset of) the class of contract specifications expressable by the semantics.

In section 4.3.4 we explain how we can do a semantic translation of SDL processes into predicates. Figure 5.1 shows the relationship between A and G as a composition, which according to definition 4.1 should be interpreted as conjunction. This does not correspond to the denotation $[\![S]\!]$ of $S = (A, G)$, which is defined by use of implication, nor does it capture the "one step longer"-semantics used in the definition of $[\![S]\!]$.

Because we are only interested in the behavior of a component as long as the assumption holds (and because we want to be able test specifications by the means of test cases), we make the assumption that if the assumption fails the component will start producing chaos after the "plus one"-step. It is worth noticing that one possible chaotic behavior is that the component produces correct output, so the "plus one"-step assert that the component guarantees correct output in *at least* one time unit after the assumption fails.

In order to model this, (A, G) is specified by a special SDL block AG, shown in figure 5.5. Instead of A simulating the environment as would be the case for the network in figure 5.1, an SDL process CoA that recognizes legal input is used. The reason for this is a pragmatic one; the testing tool we build to do functional testing of SDL contracts needs the channel i from the boundery of the SDL block to CoA in order to monitor the input (see section 7.2.1). We also make use of this channel when we compose AG blocks in section 7.3.

In addtion to recognizing legal (assumed) input, CoA is the part of the specification that does the chaos production in case the assumption is violated. As long as CoA receives legal input, the input is forwarded to G over i', but if CoA receives unexpected input it ignores all later input and starts outputting chaos on o''. CoA is a modification of A. How this modification is done is explained in section 5.3.3.

The *Switch* component forwards the output of G as long as the assumption is not violated, but switches to forward the output of CoA if CoA starts producing chaos. In order to model the "plus one"-semantics, *Switch* has a delay of one time unit before conducting the switch, i.e., if CoA is violated after t time ticks, *Switch* forwards messages from G until time tick $t+1$ before switching. In addition, some special time and causalty constraints have to be made about the components in AG. These are discussed in section 5.3.2. The *Switch* component is defined in section 5.3.5

In AG there is also a channel f from G to CoA. This is the *feedback channel*, and is used if we want to specify that the environment reponds to output from the component. Since we already have omitted this possibility from the specification scheme, the channel f is neither used nor discussed in the rest of this thesis. The channel appears in the figure as an illustration of how we believe an extension to SDL contract specifications with feedback could be done.

For simplicity this AG block has only one input and one output channel, but the AG block scheme can easily be generalized to handle multiple input and output channels. For each output channel o_j there must be a seperate *Switch* component and channels o'_j and o''_j. The channels i' and f, however, need not be multiplied, since CoA and G in any case have input queues that behave like fair merges.

5.3.2 Interpretation of Time and Causalty

In the AG block we interpret CoA and $Switch$ to be instantaneous. Intuitively this is a reasonable assumption, since they are simple components that would not use any considerable amount of computational time.

The guarantee G is assumed to have a delay of at least one time unit. G specifies the functionality of the specified component, and it is resonable to assume that there is some delay due to use of computational time in the component.

SDL specifications are executable, and therefore have "built in" causalty. CoA and $Switch$ are assumed to be weakly causal. Because they are assumed to be instantaneous, output until time t depends on input until time t.

We assume that G is strongly causal, i.e., that the output until time $t+1$

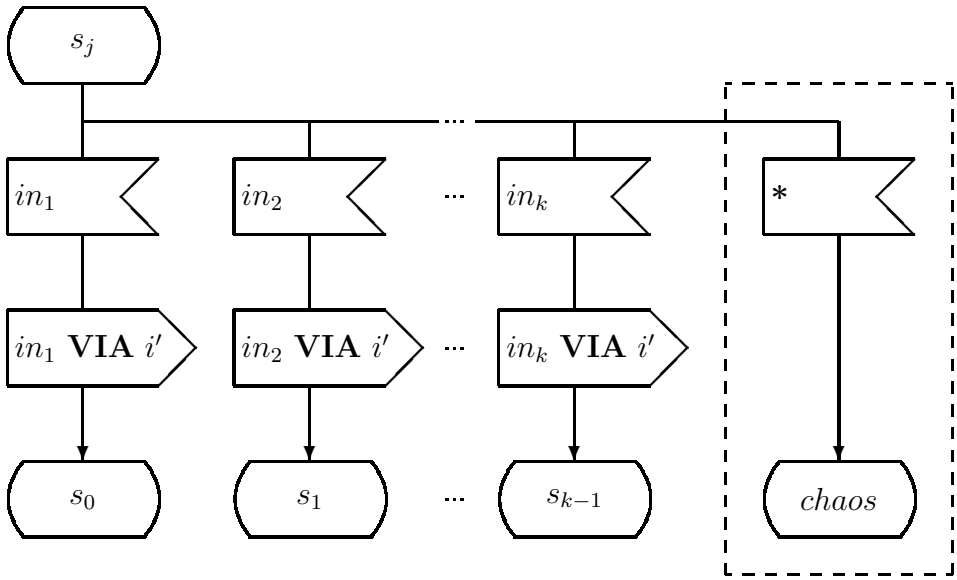


Figure 5.6: Transitions in *CoA*

depends entirely on input until time t. This assumption is supported by the assumption that G has at least one time unit delay between input and output.

5.3.3 Modification of A into CoA

The modification of A into CoA has two purposes; transformation from specifying legal input to recognizing legal input, and adding chaos production functionality.

All SDL processes have an *implicit transition* that is triggered every time the process receives a message that does not trigger any of the specified transitions from the current state. The implicit transition consumes the message and go back to the current state without performing any other actions. With an assumption that recognizes legal input, as we use, an implicit transition will be triggered in exactly the situations where the assumption is violated.

In *CoA* the implicit transition is overridden by an extra transition from all states. These transitions bring control to a state *chaos* if any unexpected input is received. The result of transforming the transitions of a state s_j in A (figure 5.4) to corresponding transitions in *CoA* is shown in figure 5.6. The transition that overrides the implicit transition is emphasized by a dashed box.

If *CoA* reaches the *chaos* state, it ignores all input and starts producing

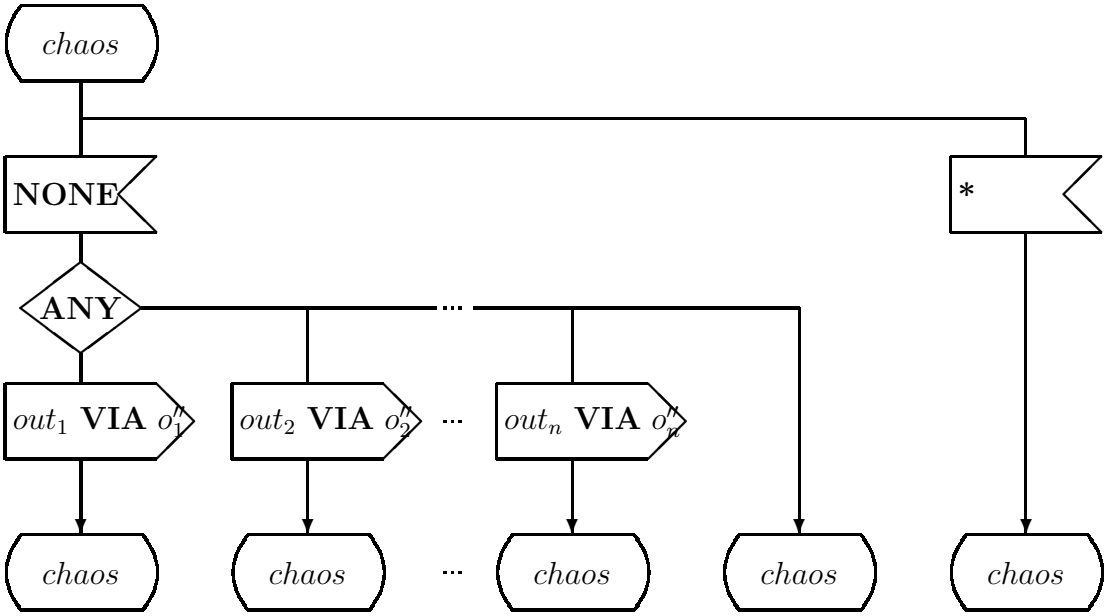


Figure 5.7: Chaos transition in *CoA*

chaos by non-deterministically sending any of the messages in the types of the output channels or nothing at all. This is specified by adding the state *chaos* with the transitions shown in figure 5.7 to *CoA*.

Note that *CoA* has k legal inputs in the state s_j, and that G (and the chaos production of *CoA*) has n possible output messages. Since we have omitted variables and messages with parameters from this discussion, we are only able to specify components with a finite number of input and output messages. This must be seen as a simplification compared to the semantics.

In figure 5.7, the leftmost **ANY**-branch (with no output) is semantically obsolete. In section 4.3.4 we define the denotation of NG to be $[\![NG]\!] = \mathbf{true}$, with the result that the stream of no *none* messages is a legal behavior of NG. This transition is present because of the SDL tool we used for conducting tests, which cannot be said to fulfil this assumption (see section 7.2.1).

5.3.4 Denotation of CoA

If we make the assumption that the block AG has one input channel i of type I and one output channel o of type O, where

$$I = \{in_1, in_2, \ldots, in_r\}$$
$$O = \{out_1, out_2, \ldots, out_n\}$$

we get the following denotation of CoA:

$$[\![CoA(i, \triangleright i', o'')]\!] = [\![(NG(\triangleright z) \otimes FM(z, i, \triangleright q)) \succ CoA'(q \triangleright i', o'')]\!]$$

Let $\mathcal{S}_{CoA} = \{s_0, s_1, \ldots, s_{l-1}\}$ be the set of states in CoA except the start state and let s_0 be the state reached by the start state. For each state $s \in \mathcal{S}_{CoA}$ let $M_s \subseteq I$ be the set of legal input when CoA is in state s and assume that if CoA receives $m \in M_s$ when is state s, control is transferd to a state $s' \in \mathcal{S}_{CoA}$.

The denotation of CoA' is defined as:

$[\![CoA'(q \triangleright i', o'')]\!] = \exists or \in \{1, 2, \ldots, n{+}1\}^\infty : (i', o'') = start_{CoA'}(q)$
where $start_{CoA'}, s_0, s_1, \ldots, s_{l-1}$ **so that** $\forall or, q, i', o''$:

[1]		$start_{CoA'}(q) = s_0(q)$
[2]$\wedge_{s \in \mathcal{S}_{CoA}}$	[1]$\wedge_{m \in M_s}$	$s(m\&q) = (\langle m \rangle, \varepsilon) \frown s'(q)$
	[2]$\wedge$	$s(none\&q) = s(q)$
	[3]$\wedge$	$s(\surd\&q) = (\langle \surd \rangle, \langle \surd \rangle) \frown s(q)$
	[4]$\wedge$	$m \in I \setminus (M_s \cup \{none\}) \Rightarrow s(m\&q) = chaos[or](q)$
[3]$\wedge$		$chaos[p\&or](none\&q) =$
		if $p = 1$ **then** $(\varepsilon, \langle out_1 \rangle) \frown chaos[or](q)$
		elseif $p = 2$ **then** $(\varepsilon, \langle out_2 \rangle) \frown chaos[or](q)$
		$\vdots$
		elseif $p = n$ **then** $(\varepsilon, \langle out_n \rangle) \frown chaos[or](q)$
		elseif $p = n + 1$ **then** $chaos[or](q)$
[4]$\wedge$		$chaos[or](\surd\&q) = (\langle \surd \rangle, \langle \surd \rangle) \frown chaos[or](q)$
[5]$\wedge$		$m \in I \setminus \{none\} \Rightarrow chaos[or](m\&q) = chaos[or](q)$

The numbers in brackets are in the below proofs used for refering to the equations.

5.3.5 Switch

In this section we give a semantic time dependent definition of the *Switch* component. This is an "ideal *Switch*", and we do not give an SDL representation of it. Because of problems with representing time and distinguishing between messages from different channels, another switch was used when tests on SDL contract specifications were conducted. This switch is specified and discussed in section 7.3.2.

The denotation of *Switch* is

$[\![Switch(o', o'' \rhd o)]\!] = (o = start_{Switch}(o', o''))$
where $start_{Switch}, s_1, s_2, s_3$ **so that** $\forall o, o', o'' :$

[1]	$start_{Switch}(o', o'') = s_1(o', o'')$
[2]$\wedge$	$s_1(\surd\&o', \surd\&o'') = \surd\&s_1(o', o'')$
[3]$\wedge_{m\in M}$	$s_1(m\&o', \surd\&o'') = m\&s_1(o', \surd\&o'')$
[4]$\wedge_{m'\in M}$	$s_1(\surd\&o', m'\&o'') = s_2(\surd\&o', o'')$
[5]$\wedge_{m,m'\in M}$	$s_1(m\&o', m'\&o'') = m\&s_1(o', m'\&o'')$
[6]$\wedge$	$s_2(\surd\&o', \surd\&o'') = \surd\&s_3(o', o'')$
[7]$\wedge_{m'\in M}$	$s_2(\surd\&o', m'\&o'') = s_2(\surd\&o', o'')$
[8]$\wedge$	$s_3(o', \surd\&o'') = \surd\&s_3(o', o'')$
[9]$\wedge_{m'\in M}$	$s_3(o', m'\&o'') = m'\&s_3(o', o'')$

where

$$o, o', o'' \in M^\omega$$

The numbers in brackets are used as references to the equations in the below proofs.

It is this component that handle the delay of one time unit between the assumption has been violated and AG starts outputting chaos. This is done as follows: If we assume that o'' has its first message after the t'th tick, i.e., $o'' = \langle\surd\rangle^t{}^\frown\langle m\rangle{}^\frown r$, *Switch* will stay in state s_1 after the t'th tick is processed (equations [2] and [3]). In s_1 all messages between the t'th and the $t+1$'th tick in o' are forward before control is transfered to s_2 (equations [5] and [4]). In s_2 all messages between the t and $t+1$'th tick in o'' are consumed without *Switch* making any output (equation [7]). Control is then (by eqauation [6]) transfered to s_3 where only messages from o'' is forwarded (equations [8]-[10]).

5.3.6 Denotation of AG

If we assume the AG block has n input channels $i_1, i_2, \ldots, i_n$ and m output channels $o_1, o_2, \ldots, o_m$, the full denotation of AG becomes

$$\begin{aligned}
&[\![AG(i_1, i_2, \ldots, i_n \rhd o_1, o_2, \ldots, o_m)]\!] = \\
&\quad [\![\ CoA(i_1, i_2, \ldots, i_n \rhd i', o''_1, o''_2, \ldots, o''_m) \\
&\quad \otimes\ G(i' \rhd o'_1, o'_2, \ldots, o'_m) \\
&\quad \otimes_{j=1}^{m}\ Switch(o'_j, o''_j \rhd o_j)]\!]
\end{aligned}$$

We now state two propositions on the semantic relationship between AG and (A, G).

Proposition 5.1 *If (A, G) is a contract specification where A and G are saftey properties, AG is an SDL representation of (A, G) and the assumption that CoA and Switch are instantaneous is true, then*

$$[\![AG]\!] = [\![(A, G)]\!]$$

□

Proposition 5.2 *If AG and (A, G) are as in proposition 5.1, except that CoA and Switch have additional delay, then*

$$[\![AG]\!] \subseteq [\![(A, G)]\!]$$

□

Proof of proposition 5.1

In this proof we assume that AG has one input channel i and one output channel o. Since we have ommitted feedback from AG, we use

$$[\![(A, G)]\!] = \forall t \in \mathbb{N} : A(i{\downarrow_t}) \Rightarrow G(i{\downarrow_t}, o{\downarrow_{t+1}})$$

as denotation of (A, G). In the proof, m and m' denote any messages, r any timed stream, and k and p any natural numbers.

(1) $[\![AG]\!] = [\![(A, G)]\!]$

is proved by proving

(2) $[\![AG]\!] \subseteq [\![(A, G)]\!]$

and

(3) $[\![(A, G)]\!] \subseteq [\![AG]\!]$

Proof of (2):
We assume

(4) $(i, o) \in [\![AG]\!]$

and prove

(5) $(i, o) \in [\![(A, G)]\!]$

(5) is equivalent to

(6) $\forall t \in \mathbb{N} : A(i{\downarrow_t}) \Rightarrow G(i{\downarrow_t}, o{\downarrow_{t+1}})$

We prove (6) by assuming

(7) $(i{\downarrow_t}, o{\downarrow_{t+1}}) \in [\![AG]\!]$

and proving

(8) $A(i{\downarrow_t}) \Rightarrow G(i{\downarrow_t}, o{\downarrow_{t+1}})$

for an arbitrary t. We have two cases: AG has received unexpected input before time t or AG has not received unexpected input before time t. In the first case

(9) $A(i{\downarrow_t}) = \mathbf{false}$

and (8) follows trivially. For the other case we assume

(10) $A(i{\downarrow_t})$

and prove

(11) $G(i{\downarrow_t}, o{\downarrow_{t+1}})$

(7) is equivalent to

(12) $$\begin{aligned}\exists i', o', o'' : \ & (i{\downarrow_t}, (i'{\downarrow_t}, o''{\downarrow_t})) \in [\![CoA]\!] \\ & \wedge (i'{\downarrow_t}, o'{\downarrow_{t+1}}) \in [\![G]\!] \\ & \wedge ((o'{\downarrow_{t+1}}, o''{\downarrow_{t+1}}), o{\downarrow_{t+1}}) \in [\![Switch]\!]\end{aligned}$$

If AG has not received unexpected input before time t, then $o''{\downarrow_t} = \langle\surd\rangle^t$ and $i{\downarrow_t} = i'{\downarrow_t}$ by equations [2.1]-[2.3] of $[\![CoA]\!]$. We assume that G on input $i{\downarrow_t}$ produces output $r^\frown s^\frown\langle\surd\rangle$ where $\#(\{\surd\}Ⓢ r) = t$, $r.\#r = \surd$ and $s = \langle m\rangle^k$. We then have $o'{\downarrow_{t+1}} = r^\frown\langle m\rangle^k{}^\frown\langle\surd\rangle$. By equations [2] and [3] of $[\![Switch]\!]$ we have $o{\downarrow_t} = r$.

We have two possibilities; AG does not receive unexpected input before time $t+1$ or AG does receive unexpected input before time $t+1$. In the first case $o''{\downarrow_{t+1}} = \langle\surd\rangle^{t+1}$, so by equations [2] and [3] of $[\![Switch]\!]$ we have $o{\downarrow_{t+1}} = r^\frown\langle m\rangle^k{}^\frown\langle\surd\rangle$. In the other case we have that $o''{\downarrow_{t+1}} = \langle\surd\rangle^t{}^\frown\langle m'\rangle^p{}^\frown\langle\surd\rangle$ by equations [2.4], [3] and [4] of $[\![CoA]\!]$. If $p = 0$, this is the same as the first case. If $p > 0$ we have $o{\downarrow_{t+1}} = r^\frown\langle m\rangle^k{}^\frown\langle\surd\rangle$ where $\langle m\rangle^k = s$ by equations [5] (applied k times), [4], [7] (applied $p-1$ times) and [6] of $[\![Switch]\!]$. From this (11) follows, and (2) is proved.

Proof of (3):
We assume

$$(13) \qquad (i, o) \in [\![(A, G)]\!]$$

and prove

$$(14) \qquad (i, o) \in [\![AG]\!]$$

From (13), we have

$$(15) \qquad A(i{\downarrow_t}) \Rightarrow G(i{\downarrow_t}, o{\downarrow_{t+1}})$$

for an arbitrary t, and prove

$$(16) \qquad (i{\downarrow_t}, o{\downarrow_{t+1}}) \in [\![AG]\!]$$

(16) is equivalent to

$$(17) \qquad \begin{array}{ll} \exists i', o', o'' : & (i{\downarrow_t}, (i'{\downarrow_t}, o''{\downarrow_t})) \in [\![CoA]\!] \\ & \wedge (i'{\downarrow_t}, o'{\downarrow_{t+1}}) \in [\![G]\!] \\ & \wedge ((o'{\downarrow_{t+1}}, o''{\downarrow_{t+1}}), o{\downarrow_{t+1}}) \in [\![Switch]\!] \end{array}$$

We have two cases

$$(18) \qquad A(i{\downarrow_t}) = \textbf{false}$$

and

$$(19) \qquad A(i{\downarrow_t}) = \textbf{true}$$

(18) asserts that illegal input is received before time t, so by (15) any output is allowed after time t. By equations [2.4], [3] and [4] of $[\![CoA]\!]$ and equations [4], [6], [8] and [9] of $[\![Switch]\!]$ such output exists, and (16) follows for this case.

For (19) we have

$$(20) \qquad G(i{\downarrow_t}, o{\downarrow_{t+1}}) = \textbf{true}$$

Since we have assumed that AG and (A, G) specify the same system or compenent, we can by (20) assume that

$$(21) \qquad (i{\downarrow_t}, o{\downarrow_{t+1}}) \in [\![G]\!]$$

We then have to show

$$(22) \qquad \begin{array}{ll} \exists o'' : & (i{\downarrow_t}, (i{\downarrow_t}, o''{\downarrow_t})) \in [\![CoA]\!] \\ & \wedge ((o{\downarrow_{t+1}}, o''{\downarrow_{t+1}}), o{\downarrow_{t+1}}) \in [\![Switch]\!] \end{array}$$

By equations [2.1]-[2.3] of $[\![CoA]\!]$ the first conjunct of (22) holds and we have that $o''\downarrow_t = \langle\surd\rangle^t$.

In order to show the second conjunct of (22), we show

$$(23) \qquad ((o'\downarrow_{t+1}, o''\downarrow_{t+1}), o\downarrow_{t+1}) \in [\![Switch]\!] \wedge o\downarrow_{t+1} = o'\downarrow_{t+1}$$

We assume $o'\downarrow_{t+1} = r^\frown s^\frown\langle\surd\rangle$ where $\#(\{\surd\}Ⓢ r) = t$, $r.\#r = \surd$ and $s = \langle m\rangle^k$. Because $o''\downarrow_t = \langle\surd\rangle^t$ we have that $o\downarrow_t = r$.

We then have two possibilities; $o''\downarrow_{t+1} = \langle\surd\rangle^{t+1}$ or $o''\downarrow_{t+1} = \langle\surd\rangle^t{}^\frown\langle m'\rangle^p{}^\frown\langle\surd\rangle$. In the first case $o\downarrow_{t+1} = r^\frown s^\frown\langle\surd\rangle$ by equations [2] and [3] of $[\![Switch]\!]$. If $p = 0$ in the second case, this is the same as the first case. If $p > 0$ in the second case, then $o\downarrow_{t+1} = r^\frown\langle m\rangle^k{}^\frown\langle\surd\rangle$, were $\langle m\rangle^k = s$, by equations [5] (applied k times), [4], [7] (applied $p-1$ times) and [6] of $[\![Switch]\!]$. From this (23) follows and (22) is proved. (17) follows from (21) and (22), and (3) is proved.

Proof of proposition 5.2

If CoA and $Switch$ have additional delay and we assume (10) in the proof of (2), we have $\overline{i'\downarrow_t} \sqsubseteq \overline{i\downarrow_t}$ and $\overline{o\downarrow_{t+1}} \sqsubseteq \overline{o'\downarrow_{t+1}}$. Since the specifed component is assumed to have undefined delay, (11) is still true and (2) still holds. If CoA has delay of at least one time unit, we always have $o''\downarrow_{t+1} = \langle\surd\rangle^{t+1}$ when $A(i\downarrow_t) \wedge \neg A(i\downarrow_{t+1})$. The specified delay when $Switch$ switches then become unnecessary.

If we do not have the assumption that CoA is instantaneous, (3) in the proof of proposition 5.1 does not hold. By (20) and the assumption that G is strongly causal, output $o\downarrow_{t+1}$ depend on input $i\downarrow_t$ when $A(i\downarrow_t) = \mathbf{true}$. If there is delay in CoA we have that $\overline{i'\downarrow_t} \sqsubseteq \overline{i\downarrow_t}$, so $i\downarrow_t = i'\downarrow_t$ does not generally hold. The conjunction of (21) and (22) then does not hold, because we have no guarantee that input $i'\downarrow_t$ yields output $o\downarrow_{t+1}$.

Chapter 6

Adaption of Composition Principle

In this chapter we formulate a composition rule for validation of decomposition of contract oriented specifications as defined in chapter 5. In section 6.1 the semantic composition rule and a corollary are formulated and section 6.2 provides a proof of the composition rule. How this rule is applied to SDL is shown in section 7.4.

6.1 Composition Rule

We formulate a rule for validation of the refinement $S \rightsquigarrow S'$ where S is a contract specification $S = (A, G)$ of a system (or component) and $S' = \otimes_{j=1}^{n} S_j$ is a composition of n contract specifications $S_j = (A_j, G_j)$ of subcomponents. The rule is inspired by the Composition Theorem in [Abadi and Lamport, 1995] and the network rules in [Stølen, 1996].

Let i and o denote the tuples of, respectively, input and output streams of S, and for $j = 1, 2, \ldots, n$ let i_j and o_j denote the tuples of input and output streams of S_j.

We have that

$$l = \cup_{j,k \in \{1,2,\ldots,n\}, j \neq k} (o_j \cap i_k)$$

are the internal streams of $\otimes_{j=1}^{n} S_j$, so clearly

$$i = (\cup_{j=1}^{n} i_j) \setminus l$$

$$o = (\cup_{j=1}^{n} o_j) \setminus l$$

Because we want the rule to be as general as possible, we assume that all components may have external streams and that all components may

communicate with all other components. This will of course not always be true, so the tuples of streams in this formulation of the rule may be tuples of zero streams.

We assume that A, G, A_j and G_j, for $j = 1, 2, \ldots, n$, only specify safety properties.

Proposition 6.1 (Composition rule) *If (A, G) and (A_j, G_j), for $j = 1, 2, \ldots, n$, are contract specifications with streams as described above, then*
if

$$\begin{array}{l} \textit{for each } k = 1, 2, \ldots, n, \\ \forall t \in \mathbb{N} : A(o{\downarrow_t}, i{\downarrow_t}) \wedge \bigwedge_{j=1}^{n} G_j(i_j{\downarrow_t}, o_j{\downarrow_{t+1}}) \Rightarrow A_k(o_k{\downarrow_{t+1}}, i_k{\downarrow_{t+1}}) \end{array}$$

and

$$\forall t \in \mathbb{N} : A(o{\downarrow_t}, i{\downarrow_t}) \wedge \bigwedge_{j=1}^{n} G_j(i_j{\downarrow_t}, o_j{\downarrow_{t+1}}) \Rightarrow G(i{\downarrow_t}, o{\downarrow_{t+1}})$$

then

$$(A, G) \rightsquigarrow \otimes_{j=1}^{n}(A_j, G_j)$$

□

In section 4.2.5 we interpret the implication in the definition of refinement as a subset relation. In a way the rule states that the composition of the assumption A of the system and the guarantees G_j of the sub-components refines both the assumptions A_j of the sub components and the guarantee G of the system, with some special constraints on the streams. We use this observation to formulate an alternative composition rule, which we think is more intuitive in relation with functional testing.

Definition 6.1 *We define LS, $RS_{1.k}$, for $k = 1, 2, \ldots, n$, and LS_2 to be the following specifications:*

$$LS \stackrel{\text{def}}{=} A \otimes (\otimes_{j=1}^{n} G_j)$$

$$RS_{1.k} \stackrel{\text{def}}{=} A_k$$

$$RS_2 \stackrel{\text{def}}{=} G$$

□

With these definitions we get the following alternative formulation of the composition rule.

Proposition 6.2 (Alternative composition rule) *If LS, $RS_{1.k}$, for each $k = 1, 2, \ldots, n$, and RS_2 are defined as in definition 6.1, then*
if

$$\begin{array}{l} \textit{for each } k = 1, 2, \ldots, n, \\ \forall t \in \mathbb{N} : (i_k{\downarrow_t}, o_k{\downarrow_{t+1}}) \in [\![LS]\!] \Rightarrow (o_k{\downarrow_{t+1}}, i_k{\downarrow_{t+1}}) \in [\![RS_{1.k}]\!] \end{array}$$

and

$$\forall t \in \mathbb{N} : (i{\downarrow_t}, o{\downarrow_{t+1}}) \in [\![LS]\!] \Rightarrow (i{\downarrow_t}, o{\downarrow_{t+1}}) \in [\![RS_2]\!]$$

then

$$(A, G) \rightsquigarrow \otimes_{j=1}^{n} (A_j, G_j)$$

□

6.1.1 Corollary on Behavioral Refinement

If we in the refinement

$$S \rightsquigarrow S'$$

let $n = 1$, i.e., $S' = (A', G')$, we get a corollary from the composition rule that apply to behavioral refinement of contract specifications.

Proposition 6.3 (Behavioral refinement rule) *If (A, G) and (A', G') are contract specifications, then*
if

$$\forall t \in \mathbb{N} : A(o{\downarrow_t}, i{\downarrow_t}) \wedge G'(i{\downarrow_t}, o{\downarrow_{t+t}}) \Rightarrow A'(o{\downarrow_{t+1}}, i{\downarrow_{t+1}})$$

and

$$\forall t \in \mathbb{N} : A(o{\downarrow_t}, i{\downarrow_t}) \wedge G'(i{\downarrow_t}, o{\downarrow_{t+t}}) \Rightarrow G(i{\downarrow_t}, o{\downarrow_{t+1}})$$

then

$$(A, G) \rightsquigarrow (A', G')$$

□

We observe that this is in agreement with the discussion on behavioral refinement in section 5.2.

6.2 Proof of Composition Rule

This section contains a proof of proposition 6.1. The proof is made by help from the proofs in [Stølen, 1995]. We prove the rule with no constraints on the streams, and therefore we can assume that the rule is valid for all type-correct streams. Each occurence of t are universially quantified over $\mathbb{N}$.

Assuming

$$(1) \qquad A(o\downarrow_t, i\downarrow_t) \wedge \bigwedge_{j=1}^{n} G_j(i_j\downarrow_t, o_j\downarrow_{t+1}) \Rightarrow A_k(o_k\downarrow_{t+1}, i_k\downarrow_{t+1})$$

for $k = 1, 2, \ldots, n$ and

$$(2) \qquad A(o\downarrow_t, i\downarrow_t) \wedge \bigwedge_{j=1}^{n} G_j(i_j\downarrow_t, o_j\downarrow_{t+1}) \Rightarrow G(i\downarrow_t, o\downarrow_{t+1})$$

we prove

$$(3) \qquad (A, G) \rightsquigarrow \otimes_{j=1}^{n}(A_j, G_j)$$

From definition 4.2, (3) is the same as

$$(4) \qquad [\![\otimes_{j=i}^{n}(A_j, G_j)]\!] \Rightarrow [\![(A, G)]\!]$$

Assuming

$$(5) \qquad [\![\otimes_{j=1}^{n}(A_j, G_j)]\!]$$

we need to prove

$$(6) \qquad [\![(A, G)]\!]$$

By definition 5.1, (6) is equivivalent to

$$(7) \qquad A(o\downarrow_t, i\downarrow_t) \Rightarrow G(i\downarrow_t, o\downarrow_{t+1})$$

Assuming

$$(8) \qquad A(o\downarrow_t, i\downarrow_t)$$

we need to prove

$$(9) \qquad G(i\downarrow_t, o\downarrow_{t+1})$$

By premiss (2), (9) can be proved by proving

$$(10) \qquad A(o\downarrow_t, i\downarrow_t) \wedge \bigwedge_{j=1}^{n} G_j(i_j\downarrow_t, o_j\downarrow_{t+1})$$

Because we already have assumed (8), we need to prove

(11) $\bigwedge_{j=1}^{n} G_j(i_j{\downarrow}_t, o_j{\downarrow}_{t+1})$

From (5) and definition 4.1 we obtain

(12) $\exists l \in (D^\omega)^p : \bigwedge_{j=1}^{n} [\![(A_j, G_j)]\!]$

We assume that l exists and omit the $\exists$-quantifier. From definition 5.1 we get

(13) $\bigwedge_{j=1}^{n} (A_j(o_j{\downarrow}_t, i_j{\downarrow}_t) \Rightarrow G_j(i_j{\downarrow}_t, o_j{\downarrow}_{t+1}))$

By (13), (11) can be proved by proving

(14) $\bigwedge_{j=1}^{n} A_j(o_j{\downarrow}_t, i_j{\downarrow}_t)$

We prove (14) by induction on t. The induction start

(15) $\bigwedge_{j=1}^{n} A_j(o_j{\downarrow}_0, i_j{\downarrow}_0)$

follows from the assumption that the A_j only are allowed to express safety properties; before time has started A_j cannot be violated. We assume

(16) $\bigwedge_{j=1}^{n} A_j(o_j{\downarrow}_t, i_j{\downarrow}_t)$

and prove

(17) $\bigwedge_{j=1}^{n} A_j(o_j{\downarrow}_{t+1}, i_j{\downarrow}_{t+1})$

(17) can by proved by proving each

(18) $A_j(o_j{\downarrow}_{t+1}, i_j{\downarrow}_{t+1})$

By premiss (1), to prove (17) it suffices to prove

(19) $A(o{\downarrow}_t, i{\downarrow}_t) \wedge \bigwedge_{j=1}^{n} G_j(i_j{\downarrow}_t, o_j{\downarrow}_{t+1})$

which follows from (8), (13) and (16).

Chapter 7

Testing Tool

In section 3.2.4 two prototype tools are suggested:

- Direct testing tool;
- Abadi/Lamport testing tool.

In this chapter we describe how we use functionality from the specification tool Telelogic Tau SDL Suite 4.2 [Telelogic, 2001] and small programs in the programming language Perl [Schwartz and Christiansen, 1997] to simulate these tools. These simulations are done in order to conduct experiments on the testing proposed strategies.

Section 7.1 discusses how validation of refinement is related to testing with test cases, and section 7.2 explains how we use Telelogic Tau to do testing of refinement.

The Direct testing tool is described in section 7.3 and the Abadi/Lamport testing tool in section 7.4. Section 7.5 contains a small comment on automation. The tests conducted with these tools are described and discussed in chapter 8.

7.1 Testing of Refinement

There have been done considerable reseach in the field of testing and test case generation. If we are to benefit from theories from this field, we need to relate them to the semantics we have defined and used for formalizing SDL.

The purpose of our tools is to validate refinements

$$S \rightsquigarrow \otimes_{j=1}^{n} S_j$$

where S and the S_j are contract specifications. In section 4.2.5 we assert that a refinement $S \rightsquigarrow S'$ can be interpreted by a subset relation on sets

of behaviors, i.e., $[\![S']\!] \subseteq [\![S]\!]$. In our semantics a behavior is a pair (i, o) of tuples of streams where i is the input streams and o the output streams.

When related to the testing tools, we assosiate test cases with behaviors, i.e., a test case for a specification S is a behavior $(i, o) \in [\![S]\!]$ of S. Validation of the refinement is done by showing that a set of test cases for a specification S' is a subset of the set of test cases for a specification S. From definition 4.2 refinement can also can be interpreted as logical implication. We make specifications that correspond to each side of the implication in the refinement to be tested.

A suitable set of test cases is generated from the specification representing the left side of the implication. These test cases are then tested against the specification representing the right side of the implication. If all test cases generated from the left side are possible executions of the right side the subset relation is ensured, but only in a weak sense of the word. A full validation by this strategy is impossible, because both specifications may have an infinite number of possible behaviors. In the opposite situation – the left side of the implication produces a test case the right side cannot execute – we can be entirely sure that the refinement is invalid, because the subset relation does not hold.

When test cases are generated from a specification, the result will be a finite set of finite test cases, while the specification may have infinitly many infinitly long behaviors. This does probably not represent any problem, since our specification only specify safety properties. For each point in time t there can only be a finite number of behaviors; infinitly many behaviors is a result of infinitely long time. A finite set of test cases is probably sufficient for testing safety properties if the set provides a full cover of the specification under test.

7.2 Testing with Telelogic Tau SDL Suite

The parts of Telelogic Tau SDL Suite we use is the SDL editor and SDL Validator. In this section we describe how we use SDL Validator to do testing as explained in section 7.1. SDL Validator is described in section 7.2.1. The test case format is Message Sequence Charts (MSCs) [ITU-T, 2000a]. MSC test cases and their semantics are discussed in section 7.2.2. In section 7.2.3 we describe how testing of refinement is done by the means of SDL Validator and simple programs in the Perl programming language.

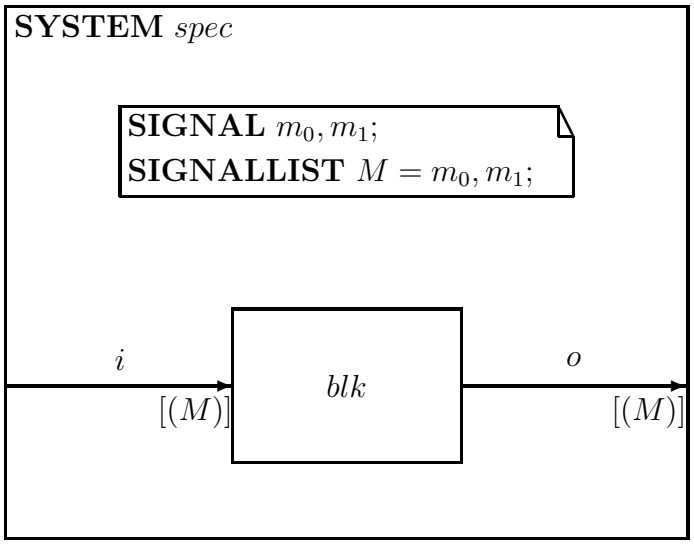


Figure 7.1: SDL system

7.2.1 SDL Validator

An SDL Validator is an executable program that is generated from a concrete SDL specification prepared with the SDL editor. This means that an SDL Validator is specific for an SDL specification. The generation is done automatically by Telelogic Tau SDL Suite. The features of SDL Validator we use are:

- generate test cases in the form of MSCs from an SDL specification;
- verify MSC test cases against an SDL specification.

The algorithm used for test case generation is called *Tree Walk*. During a *Tree Walk* the SDL Validator conducts a state space exploration and builds a behavioral tree (reachability graph) for the concrete SDL specification the SDL Validator was generated from. The behavioral tree is based on input from the environment of the specified system, and branches every time different input leads to different behavior and every time the system is allowed to do a non-deterministic choice.

The exploration continues until a given symbol coverage is reached or a given upper limit on computational time expires. When the exploration is finished, paths of the behavioral tree are stored as MSCs. A description of the algorithm can be found in e.g. [Koch et al., 1998].

When test cases are generated, the SDL Validator treats the specification as a blackbox. This means the test cases only show communication over the

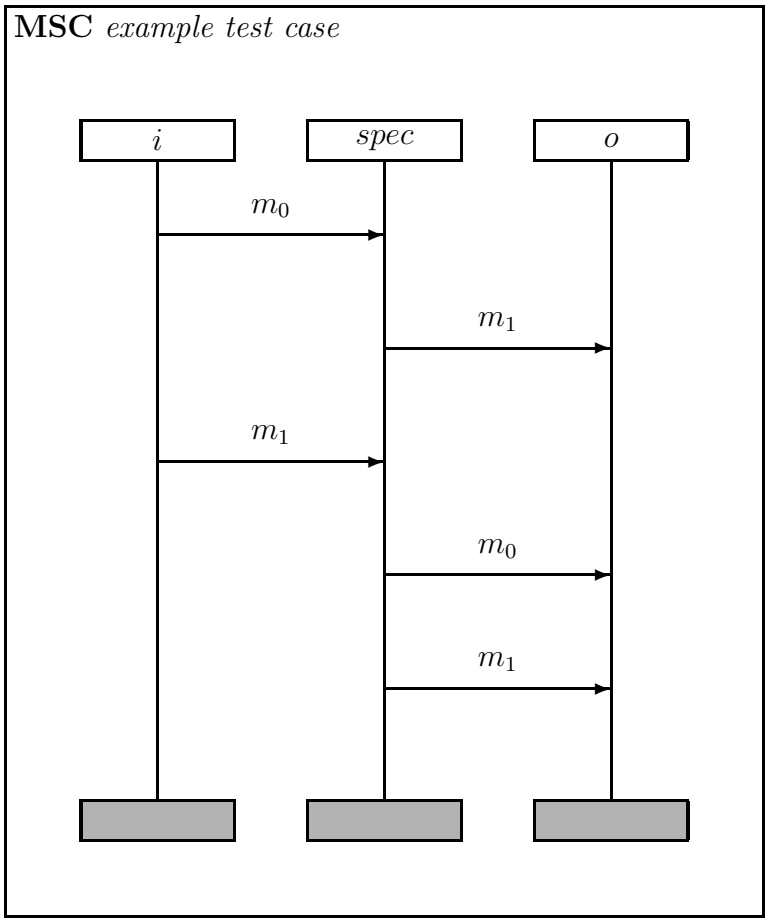


Figure 7.2: Example MSC

external channel of the system. Figure 7.1 shows a simple SDL system with one block, one input channel and one output channel. Test cases generated from this specification will only show messages over i and o, regardless of internal structure of *blk*.

MSC verification is done by a variant of state space exploration. The SDL Validator explores the behavior of the SDL specification for an execution path that enables the communication specified by the MSC to be verified. If such a path is found during the exploration, the MSC is verified. This algorithm is explained in [Ek, 1993] and [Koch et al., 1998].

SDL Validator has a weak handling of time. According to [Telelogic, 2001] the time used for executing one symbol in an SDL process (e.g. input or output) can be defined to be zero or undefined, and further that time is not represented when SDL Validator does state space explorations. As a

result of the latter the special function **NOW** will always return zero during such explorations.

In section 4.3.4 we make the assumption about SDL specifications that time always passes. Because of this we let the execution time of a symbol be undefined.

Since SDL Validator actually simulates a specification on a computer when generating or executing test cases, and during this simulation time will pass monotonously, this assumption is not violated. The function **NOW** will not cause any problems since we have restricted SDL to time independent specifications (and **NOW** is excluded from the SDL subset we use). Causality requirements are ensured by the fact that an executed specification is bound to be causal, even strongly causal if the time units are sufficiently small.

In section 4.3.4 we modeled the input of **NONE** by a component NG, and asserted that the behavior of NG could be described by **true**. This means that all possible streams of *none*s, including the stream of zero *none*s are possible behaviors of NG. There are reasons to believe that this assumption is not true when SDL Validator is generating test cases. SDL Validator will then try to reach a symbol coverage as high as possible, and therefore simulate input of **NONE** if input of **NONE** is specified.

7.2.2 MSC Test Cases

The main constructs of MSCs are *instances* and *messages*. Instances represent communicating objects or processes, and are drawn as two boxes with a vertical line between them. The upper box is the *instance head* with the name of the instance and the lower box is the *instance end*. The line is called the *instance axis* and represents the local time axis of the instance, where time passes on downwards. In the example MSC in figure 7.2, *spec* is an example of an instance. Messages are drawn as arrows labeled with the message name.

The MSC test cases generated by an SDL Validator are very simple; they do not contain constructs like loops or references, only instances and messages. Each MSC test case contains an instance representing the SDL system from which the SDL Validator is generated, and one instance per external channel of this SDL system. A message between an instance representing a channel and the instance representing the system represents a message send to or from the system over that channel. An MSC test case generated from the SDL system in figure 7.1 could look like the MSC in figure 7.2. We view an MSC test case as a possible behavior (input/output pair) of a specification, and nothing more.

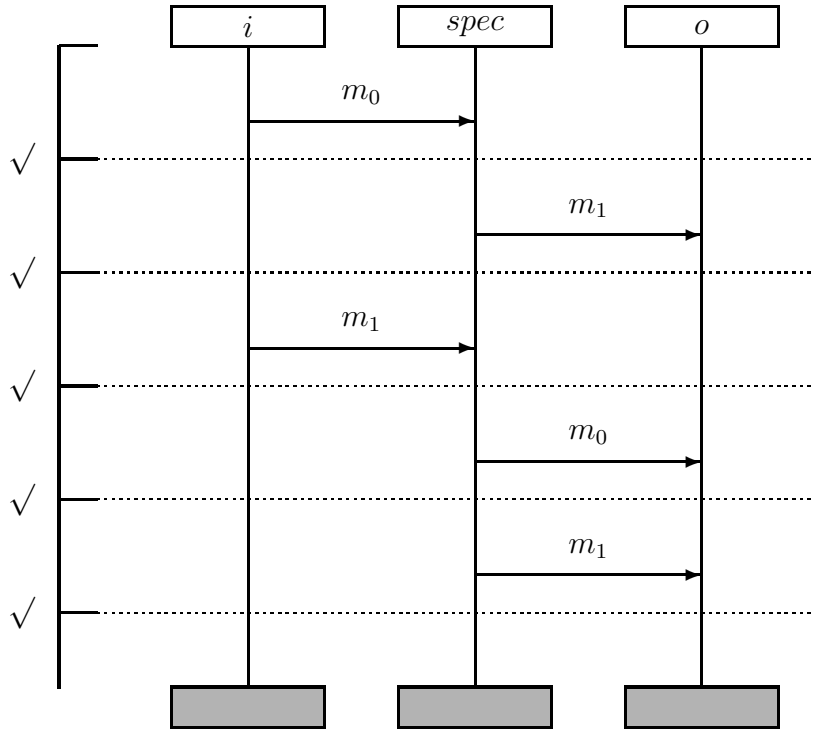


Figure 7.3: Example MSC with time ticks

The semantics of MSC state that events only can happen in the order they appear top down on an instance axis. We have also made the assumption that message passing is instantaneous, i.e., there is no delay between the output and input of a given message. Because of this and because all messages in an MSC test case go either to or from the instance representing the SDL system, the ordering of events (message input or message output) along this instance axis provides us with a total ordering of all events in the MSC.

Because of this total ordering, and because all channels are unidirected, it is quite easy to relate MSCs to input/output pairs of (tuples of) streams. If we divide the instance axis into sufficiently small time units, we are able to extract timed streams from the MSCs with at most one message per time unit, and the total ordering prevents us from getting causalty problems.

Figure 7.3 illustrates how the MSC form figure 7.2 can be divided into time units. With these time units the MSC represents the input/output pair

$$(i, o) \in M^* \times M^*$$

where

$$M = \{m_0, m_1\}$$

and

$$i = \langle m_0, \surd, \surd, m_1, \surd, \surd, \surd \rangle$$
$$o = \langle \surd, m_1, \surd, \surd, m_0, \surd, m_1, \surd \rangle$$

The translation can also easily be done the other way. An input/output pair truncated after a given point in time t, can be translated into a finite set of MSC test cases. The axis of the instance representing the SDL system is divided into time units. A message between the j'th and $j+1$'th time tick in an input stream is placed as an input message to the system's instance in the $j+1$'th time unit. Symmetrically, a message in an output stream is placed as an output message from the system's instance in the same way. If two messages appear in the same time unit we have no way of knowing their order, but the input/output pair can then be represented by two MSC test cases. Since there can only be a finite number of such ambiguities in a finite input/output pair, we only need a finite number of MSC test cases to represent a trunctated input/output pair.

7.2.3 Testing of Refinement with SDL Validator

This section describes the general idea of how we use SDL Validator to do testing of refinement. We assume we have two SDL specifications $Spec_1$ and $Spec_2$ and want to validate the refinement

$$Spec_1 \rightsquigarrow Spec_2$$

We use Telelogic Tau SDL Suite to generate an SDL Validator from each of these SDL specifications. The SDL Validators are executable computer programs, in this discussion refered to as, respectively, $Validator_1$ and $Validator_2$.

SDL Validators can be used as command line programs, and are able to read command files. When we use SDL Validators to do testing of refinements, we use these qualities. After $Validator_1$ and $Validator_2$ are generated, we use Perl programs to manage the actual testing. These Perl programs conduct the following tasks:

1. Generate a command file with commands that make an SDL Validator run the *Tree Walk* algorithm and generate MSC test cases. As a standard we use 100% symbol coverage and 10 minutes time limit as parameters to the *Tree Walk*. This command file can been seen in section A.5.

2. Start $Validator_2$ and make it execute the command file generated in 1.

3. Analyze the output from $Validator_2$ and extract the names of the generated MSC test cases.

4. Do necessary modifications to the MSC test cases. In all generated MSCs there will be an instance named $Spec_2$, and this name has to be changed to $Spec_1$ if $Validator_1$ is to be able to execute them. The Abadi/Lamport testing tool require other modifications of the MSC test cases as well. These modifications are described in section 7.4.2.

5. Generate a command file with commands that make an SDL Validator verify (execute) the generated MSC test cases modified with respect to $Spec_1$. A typical example of such a command file is presented in section A.6 .

6. Start $Validator_1$ and make it execute the command file generated in 5.

7. Analyze the output from $Validator_1$ and extract the results of the execution of the MSC test cases.

8. Write a report based on the output from $Validator_1$.

7.3 Direct testing

The Direct testing tool is based on testing the contract specifications described in section 5.3 directly. This means we do testing of

$$(A, G) \rightsquigarrow \otimes_{j=1}^{n}(A_j, G_j)$$

by directly testing

$$[\![\otimes_{j=1}^{n}(A_j, G_j)]\!] \subseteq [\![(A, G)]\!]$$

In the SDL context (A, G) and each (A_j, G_j) are represented by SDL blocks AG and AG_j, for $j = 1, 2, \ldots, n$. AG and all AG_j are specified by the scheme of the special AG block described in section 5.3.1. An SDL system $Spec_1$ is specified by placing AG in it alone, like in figure 7.4. Another SDL system $Spec_2$ is specified by composing the SDL blocks AG_j. An example where $j \in \{1, 2\}$ is shown in figure 7.5. This is not the most general example, since AG_2 does not have external channels.

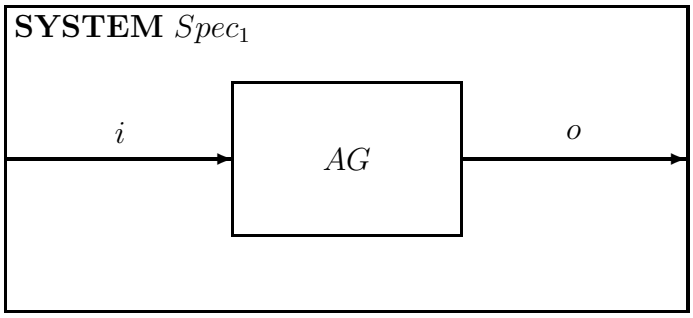


Figure 7.4: **SYSTEM** $Spec_1$

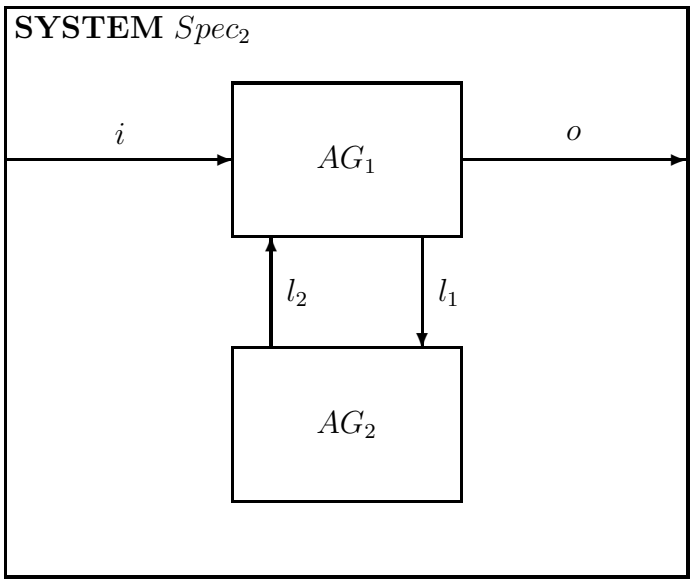


Figure 7.5: **SYSTEM** $Spec_2$

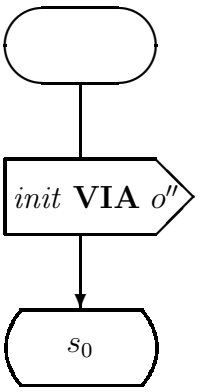
7.3.1 Validation

After specification of the SDL systems $Spec_1$ and $Spec_2$, the refinement to test is

$$Spec_1 \rightsquigarrow Spec_2$$

The test is conducted by generating SDL Validators from $Spec_1$ and $Spec_2$ and running a Perl program that works as explained in section 7.2.3. This Perl program is shown section A.1.

If all test cases generated by the SDL Validator for $Spec_2$ are verified by the SDL Validator for $Spec_1$, we have some evidence that the refinement



$init.pid = CoA.pid \wedge$
$start_{CoA'}(q) = (\varepsilon, \langle init \rangle) \frown s_0(q)$

Figure 7.6: Start transition in CoA

from $Spec_1$ to $Spec_2$, i.e., the decomposition of (A, G) into $\otimes_{j=1}^{n}(A_j, G_j)$, is valid. Symmetrically, if one or more of the test cases generated from the SDL Validator for $Spec_2$ is not verified by the SDL Validator for $Spec_1$, the decomposition is incorrect.

7.3.2 The Switch Used

In section 5.3.5 an ideal *Switch* component was defined. While conducting test with the Direct testing tool, we used another switch. The reason for this was problems with representing delay and problems with finding a sensible way of distinguishing between messages received over different channels. We were also at that point unaware of the possibility of getting errors with this *Switch*. This possibility and why errors were avoided are discussed below.

The *Switch* component we used is shown in figure 7.7. Except for the leftmost transitions, the transitions in this figure are repeated for all out_j for $j = 1, 2, \ldots, n$. The type of the of channels o and o' is assumed to be $M = \{out_1, out_2, \ldots, out_n\}$, and the type of o'' is assumed to be $(M \cup \{init\})$. The leftmost transition is used to obtain the pid (see section 4.3.8) of CoA, which is used to distinguish between messages from CoA and G. This means that the transition of the start state of CoA has to be changed to the transition shown in figure 7.6, assuming s_0 is the first ordinary state of CoA and AG only has one *Switch*. If AG has more than one output channel and therefore more than one *Switch*, CoA sends a message *init* to each. The denotation of *Switch* becomes

$$[\![Switch(o', o'' \rhd o)]\!] = [\![FM(o', o'' \rhd q) \succ SW(q \rhd o)]\!]$$

where

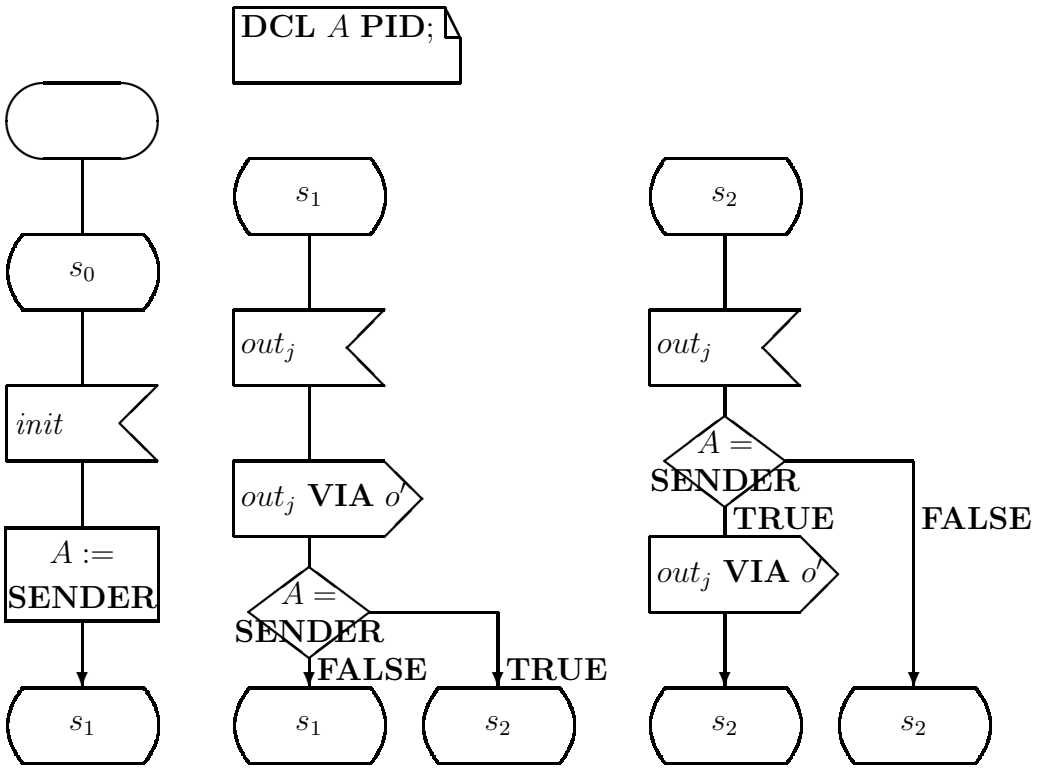


Figure 7.7: The *Switch* component

$$
\begin{array}{ll}
\multicolumn{2}{l}{[\![SW(q \rhd o)]\!] = \exists A \in PID : (o = start_{SW}[A](q))} \\
\multicolumn{2}{l}{\textbf{where } start_{SW}, s_0, s_1, s_2 \textbf{ so that } \forall A, q, o :} \\
\wedge & start_{SW}[A](q) = s_0[A](q) \\
\wedge & s_0[A](init\&q) = s_1[A := init.pid](i) \\
\wedge_{m \in M} & s_1(m\&q) = m\&(\ \textbf{if } A = m.pid \textbf{ then } s_2[A](q) \\
 & \qquad\qquad \textbf{else } s_1[A](q)) \\
\wedge_{m \in M} & s_2[A](m\&q) = \ \ \textbf{if } A = m.pid \textbf{ then } m\&s_2[A](q) \\
 & \qquad\qquad \textbf{else } s_2[A](q)
\end{array}
$$

and

$$A \in PID$$

$$o, o' \in M^{\omega}$$

$$o'', q \in (M \cup \{init\})^{\omega}$$

Discussion

There are two problems with the *Switch* we used:

- If G starts producing output before the *init* message arrives, this output may be lost. This situation can occure because channels are assumed to be without delay, and because G generally is allowed to spontaneously output messages at any time.

- There are no specified delay in the used *Switch*, as there is in the the ideal *Switch*. If CoA receives illegal input between the t'th and the $t+1$'th time tick, it will imidiately start producing chaos. Because CoA, *Switch* and the channels are assumed to be instantaneous, this chaos may displace output that G produces in this time unit. Hence the "plus one"-semantics are violated.

Both these erroneous situations were avoided by the configuration of SDL Validator. In this configuration, internal events had higher priority than input from the environment and spontaneous output (input of **NONE**). In addition all SDL processes were assumed to have undefined delay, because we had no way of specifying different time assumptions for the different SDL processes when using SDL Validator.

The first erroneous situation was avoided because processing of *init*, which is an internal event, had priority over spontaneous output from G.

The second erroneous situation was partially avoided because internal events had priority over input from the environment. When G got input from CoA, CoA did not get any new input from the environment before G had finished processing the input and *Switch* had finished processing any output from G as a result of this input. The result is that there was always delay after input from the environment, before the environment could send more input. Because of this the "plus one"-semantics was ensured in the sense that AG was allowed to produce output until $t+1$ for any legal input received until t. In addition the assumption that CoA and *Switch* are instantaneous does not hold for SDL Validator since all processes in an SDL specification have to have the same time assuptions Therefore there must always have been some delay between AG received illegal input and it started outputing chaos. As a conclusion we believe that the set of possible behaviors the tool was able to simulate was reduced, but did not contain behaviors that violated the "plus one"-semantics.

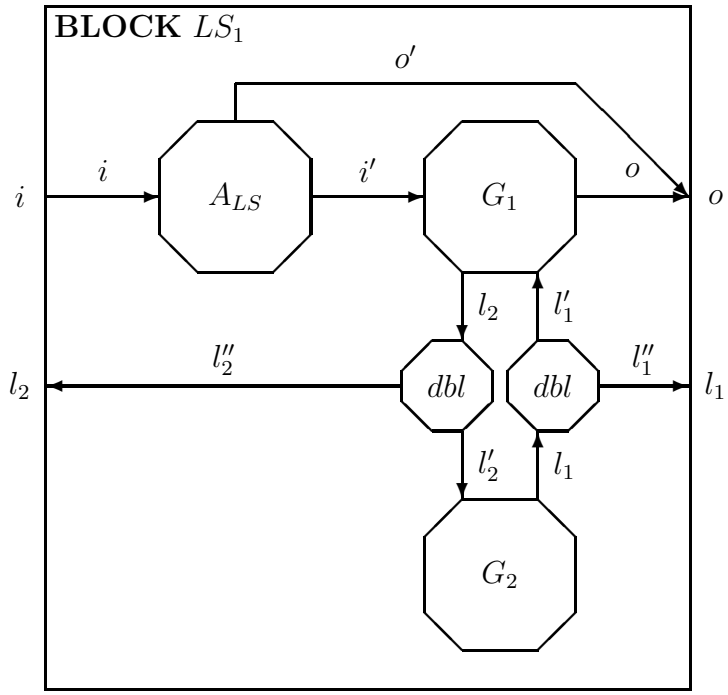


Figure 7.8: **BLOCK** LS_1

7.4 Abadi/Lamport testing

The Abadi/Lamport testing tool is based on proposition 6.2. Testing of the refinement

$$(A, G) \rightsquigarrow \otimes_{j=1}^{n}(A_j, G_j)$$

is done by testing each of the premisses in proposition 6.2. In order to do this, we make SDL specifications LS, $RS_{1.k}$, for $k = 1, 2, \ldots, n$, and RS_2 in correspondence with definition 6.1. In LS and $RS_{1.k}$, the assumptions A and A_k are modified relatively to assumptions in simple form (shown in figure 5.4). These modifications are explained in section 7.4.1.

In the formulation of the rule, LS is equal for both the n first premisses and the last premiss, but the input/output pairs are different because $RS_{1.k}$ for $k \in \{1, 2, \ldots, n\}$ and RS have different external channels. We solve this by making one SDL specification LS_1 representing LS is the first n premisses, and one SDL specification LS_2 representing LS in the last premiss.

For the example used in section 7.3 these will look like the SDL blocks in figures 7.8 and 7.9 as the only blocks in a SDL system each. (In accordance with the restrictions we have made, the feedback channel is omitted.) The

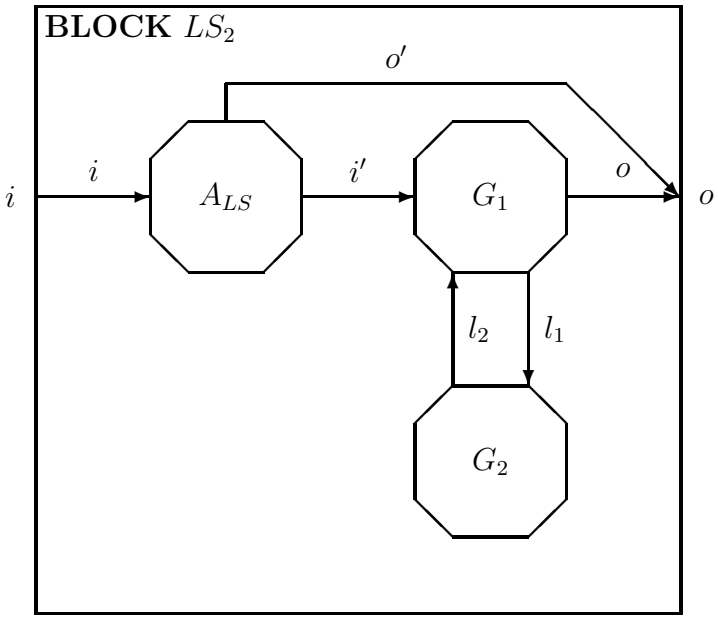


Figure 7.9: **BLOCK** LS_2

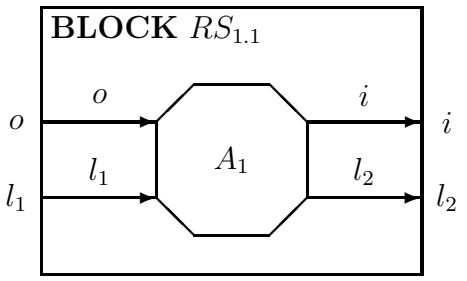


Figure 7.10: **BLOCK** $RS_{1.1}$

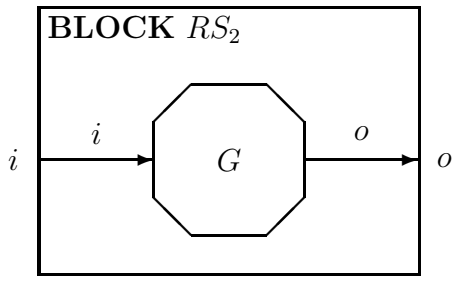


Figure 7.11: **BLOCK** RS_2

difference between them is that LS_1 has *dbl* processes, which for each message they receive send one copy to each of the output channel, and some extra channels that are used for sending copies of internal messages out of the specification. A general SDL process *dbl* is shown in figure 7.12.

The reason for specifying LS_1 this way, is that the $RS_{1.k}$ will have internal channels of LS_1 as their external channels. Test cases generated from LS_1 contain communication over all channels in LS_1. Before verifying these test cases against $RS_{1.k}$, some of these communications may have to be removed. This is further explained in section 7.4.2. Since LS_2 and RS_2 always have the same input and output channels, LS_2 does not need the *dbl* components and the extra channels.

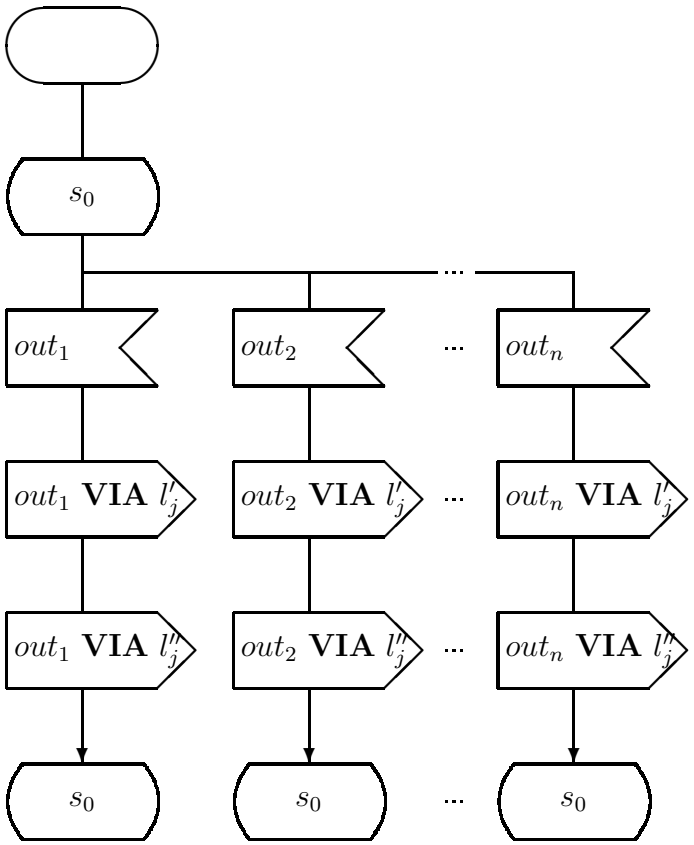


Figure 7.12: **PROCESS** *dbl*

$RS_{1.1}$ and RS_2 for the same example as above are shown in respectively figures 7.10 and 7.11. Both are the only blocks in an SDL system each. Note that in $RS_{1.1}$ some of the input channels are channels that are output channels in LS_1 and vice versa. This is of great significance when we modify test cases.

7.4.1 Modification of Assumptions

In order to detect illegal input in LS_1 and LS_2 we make a modification to A relative to its simple form. This modified assumption is called A_{LS}. A_{LS} sends a special messge err over the channel o' if illegal input is received. A_{LS} is similare to CoA in figure 5.6, but instead the implicit transition is overridden by a special transition that outputs a message err and transfer control to a state $fault$. A state of A_{LS} with its transitions is shown in figure 7.13, where the special transition is emphasized by a dashed box.

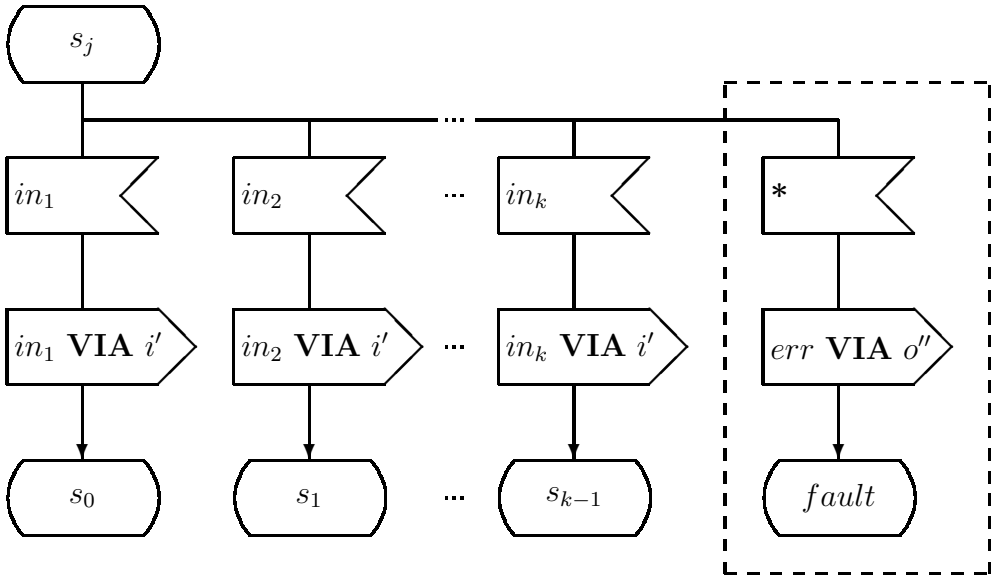


Figure 7.13: State with transitions in A_{LS}

While in the $fault$ state, A_{LS} consumes all input and performs no other actions. The $fault$ state with its transition is shown in figure 7.14. An alternative could have been to use the transition in figure 7.15 instead of the transition in the dashed box in figure 7.13 and the $fault$ state in figure 7.14. The idea would be that violation of the assumption is simulated by LS_1 and LS_2 not receiving any more input after a given point in time, but continuing to produce output based on input received until this point.

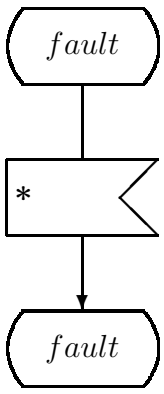


Figure 7.14: Fault trans. in A_{LS}

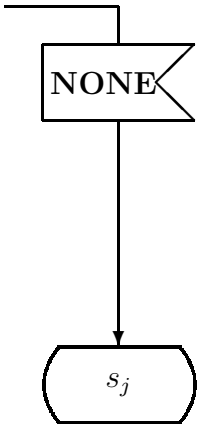


Figure 7.15: Alt. mod. of A_{LS}

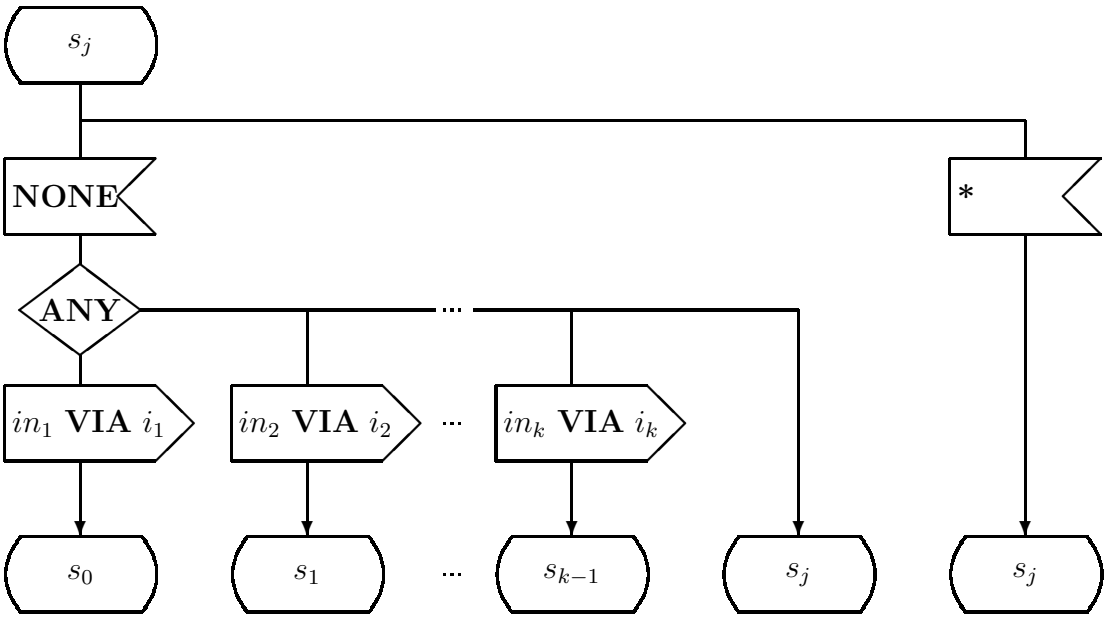


Figure 7.16: Modified assumption

A_k in $RS_{1.k}$ also has to be modified, but in a different way. A_{LS} and CoA are used to recognice and forward legal input and in different ways responding to illegal input. In $RS_{1.k}$ we just want the A_k to specify the possible input to G_k, because the $RS_{1.k}$ are used for verifying test cases, not generating test cases as opposed to LS_1 and LS_2. A state with transitions like A has in the simple form in figure 5.4, is for A_k in $RS_{1.k}$ modified to be like the state and transitions in figure 7.16. The asterisk input is a result of omitting feedback; A_k is specified to not respond to any messages it receives.

7.4.2 Modification of Test Cases

The test cases genrated from LS_1 and LS_2 have to be modified before they can be verified against the $RS_{1.k}$ and RS_2. If a test case contains the special *err* message, this means that A_{LS} has received illegal input. The input message before the *err* message, which always is the illegal input, the *err* message and all input messages after the *err* message are removed. Assuming the illegal input arrived after time t, the test case will after the modification contain legal input until time t and legal output until at least time $t+1$. This makes sure that the testing corresponds to the semantics of the compostion rule.

For the test cases that are to be verified against $RS_{1.k}$ some of the instances representing channels may have to be removed, because $RS_{1.k}$ may

have fewer channels that LS_1. Messages may also have to be turned around, i.e., input messages become output messages and output messages become input messages. The input of G_k is the output of A_k and the output of G_k is the input of A_k, and all internal messages in LS_1 will appear as output messages in the unmodified test cases generated from LS_1.

Since the test cases generated from LS_1 in any way need to be modified, LS_2 is actually superfluous. We could have modified the test cases generated form LS_1 to become the test cases generated from LS_2 by removing the instances representing the internal channels.

7.4.3 Validation

SDL Validators are generated from the SDL specifications LS_1, LS_2, $RS_{1.k}$, for $k = 1, 2, \ldots, n$, and RS_2. Testing is done by two Perl programs which work as described in section 7.2.3; one for the n first premisses, and one for the last premiss. The first program generates test cases from LS_1 and execute them against an $RS_{1.k}$ (and are therefore run n times). The second program generates test cases from LS_2 and execute them against RS_2. These programs also do the necessary modifications of test cases. The code of the programs are shown in respectively sections A.2 and A.3.

If all test cases generated from LS_1 after modification are verified by each $RS_{1.k}$, for $k = 1, 2, \ldots, n$, and all test cases generated from LS_2 after modification are verified by RS_2, there is some evidence that the refinement

$$(A, G) \rightsquigarrow \otimes_{j=1}^{n}(A_j, G_j)$$

is valid. If one of the test cases generated in testing of one of the premisses is not verified, we consider the decomposition to be invalid.

7.5 Automation

Both testing tools conduct three distinct tasks:

1. Prepare the specifications used for testing.
2. Generate SDL Validators from these specifications.
3. Generate and execute test cases with the SDL Validators.

While the second task is done automatically be Telelogic Tau SDL Suite and the third task is done automatically by Perl programs, the first task has been done manually during the work of this thesis.

We see no reasons why it should not be possible to automate also the first task, and make a tool that let all the modifications done to the specifications be hidden from the user. In a complete realization of any of these tools, the user should only need to specify guarantees, assumptions in a simple form and connections between components in the decomposition, and let the tool do the rest.

Chapter 8

Testing with Testing Tools

In this chapter we present the tests we conducted with the prototype testing tools :

- Direct testing tool, described in section 7.3;
- Abadi/Lamport testing tool, described in section 7.4;

and present the results from these tests.

Section 8.1 explains the terminology of this chapter. In section 8.2 we give a detailed description of the tests, and section 8.3 provieds a discussion of the results. In section 8.4 some conclusions based on the results are drawn. The raw test results are found in appendix C.

8.1 Terminology

Each of the examples consists of two specifications, and for each example we refer to these two specifications as the *original specification* and the *decomposition*.

When we use the expression *equivalent* about assumptions and guarantees, we mean that two assumptions or two guarantees have the same set of behaviors.

In the discussion of the examples and test results, we view the decomposition in an example as a blackbox. This means that when we in this chapter speak of the assumption or guarantee of a decomposition, we mean the overall effect of the assumptions and guarantees of the components in the decomposition on the external input and output streams.

When we say that an assumption is equivalent to **true**, the assumption allows all behaviors with messages from the types of the streams.

For a decomposition with n components, the Abadi/Lamport testing tool tests $n+1$ premisses. In accordance with the numbering of SDL specifications in section 7.4, we refer to these premisses as AL 1.1, AL 1.2, ..., AL 1.n and AL 2.

8.2 Example decomposition

15 example decompositions were tested by both tools. All examples consist of of two SDL specifications which were made within the SDL subset defined in section 4.1 and in accordance with the restrictions presented in section 3.3.

The examples are artificial in the sense that we have construced them in order to conduct these tests; they are not examples from real system development. Gathering real examples or applying the tools in a real-life, full scale system development project would be tasks too big for the time and resources available.

We have tried to cover different situations that may occure in decompositions, and most of the examples are dedicated to test specific features of the testing strategies. Among the examples are both correct and incorrect decompositions. All examples are also fairly simple compared to what we may find in real-life system development (an average of 9.2 states per example).

Zelkowitz and Wallace [1998] propose a taxonomy for experimentation in computer science. In this taxonomy our testing of the tools would be classified as *simulation*, a sub-class under *controlled methods*. According to [Zelkowitz and Wallace, 1998] the biggest weakness of the simulation method is that we do not know how well the artificial data applied (in our case decomposition examples) represent reality. We cannot be completely certain that our results are valid for the domain the artificial data models. We however believe that our sample is representative for a large class of contract decompositions.

Among the examples there are:

- nine examples of correct decomposition (examples 1, 2, 3, 5, 6, 8, 12, 13 and 14) and six examples of incorrect decomposition (examples 4, 7, 9, 10, 11 and 15);
- 14 examples where the decomposition has two components and one example where the decomposition has three components (example 2);
- 13 examples where the specifications have one input and one output channel, one example where the specifications have one input channel

and two output channels (example 2) and one example where the specifications have two input channels and one output channel (example 4).

The full example decomposition number 8 is presented in appendix B.

We observe two reasonable ways of grouping the examples. These are refered to as *Grouping 1* and *Grouping 2*.

Grouping 1

If we analyze the examples in accordance with the theory of behavioral refinement of contract specifications in section 5.2, each of the examples can be placed in one of six groups:

1. The original specification and the decomposition have equivalent assumptions and equivalent guarantees. This group contains examples 1, 2, 3, 12 and 14.

2. The original specification and the decomposition have equivalent assumptions, but the guarantee of the decomposition is strengthened relatively to the guarantee of the original specification. This group contains examples 5, 6 and 13.

3. The original specification and the decomposition have equivalent guarantees, but the assumption of the decomposition is weakend relatively to the assumption of the original specification. This group contains example 8.

4. The original specification and the decomposition have equivalent assumptions, but the guarantee of the decomposition iss weakend relatively to the guarantee of the original specification. This group contains examples 7, 10 and 11.

5. The original specification and the decomposition have equivalent guarantees, but the assumption of the decomposition is strengtened relatively to the assumption of the orginal specification. This group contains examples 4 and 15.

6. In one example the assumption of the decomposition is weakend relatively to the assumption of the original specification, and the guarantee of the decomposition is changed relatively to the guarantee of the original specification by removing and adding behaviors. This can be seen

as both a strengthening and a weakening of the guarantee at the same time. This group contains example 9.

According to the theory, the examples of groups 1-3 are legal behavioral refinements, while the example of groups 4-6 are not, so the examples of groups 1-3 are valid and the examples if groups 4-6 are invalid.

Grouping 2

This grouping concerns the use of assumptions. The groups are defined as follows:

1. Examples where the assumption of the decomposition is not equivalent to **true**. This group consists of examples 1, 2, 3, 4, 10, 14 and 15.

2. Examples where the assumption of the decomposition is equivalent to **true**, but the assumption of the original specification is not. This group consists of examples 8 and 9.

3. Examples where both the assumption of the decomposition and the assumption of the original specification are equivalent to **true**. This group consists of examples 5, 6, 7, 11, 12 and 13.

8.3 Discussion of Results

This discussion is divied into three parts; section 8.3.1 discusses the correctness of the tools, section 8.3.2 discusses the efficiency of the tools and section 8.3.3 discusses the tracability of the tools.

8.3.1 Correctness

In this section we group the examples in accordance to *Grouping 1*. The results of validating the examples with the testing tools are presented in tables 8.1 and 8.2. The row *Valid* asserts whether or not the examples are valid according to our analyses, the row *Direct validation* asserts whether or not the examples were validated with the Direct testing tool and the row *AL validation* asserts whether or not the the examples were validated by the Abadi/Lamport testing tool. N,1 in the last row, means the example was falsified by premiss AL 1.x and N,2 means the example was falsified by premiss AL 2.

We see that the testing tools gave the same results with respect to validation for all 15 examples, i.e., the testing tools validated exactly the same

Group	**1**					**2**			**3**
Example	1	2	3	12	14	5	6	13	8
Valid	Y	Y	Y	Y	Y	Y	Y	Y	Y
Direct validation	Y	Y	Y	Y	Y	Y	Y	Y	Y
AL validation	Y	Y	Y	Y	Y	Y	Y	Y	Y

Table 8.1: Result of testing groups 1-3

Group	**4**			**5**		**6**
Example	7	10	11	4	15	6
Valid	N	N	N	N	N	N
Direct validation	N	N	N	N	N	N
AL validation	N,2	N,2	N,2	N,1	N,1	N,2

Table 8.2: Result of testing groups 4-6

examples. Nine of the example decompositions were validated and six were not validated.

Valid Examples

The nine examples of groups 1-3 are all valid according to our analyses and were all validated by both tools. When tested on valid examples, both tools gave the expected results.

Invalid Examples

According to our analyses, all six examples of groups 4-6 are invalid. Neither of these examples were validated by any of the tools. Both tools gave the expected results when tested on invalid examples.

Special cases

Among the example decompositions there are some that deserve some extra discussion:

- In example 4 the specifications have two input channels (in the 14 other examples there is only one). This represented some problems in testing with the Abadi/Lamport testing tool. When LS_1 and LS_2 were specified, the assumption of the original specifications was not

able to distinguish between messages received over different channels. We resolved this by including two instances of the assumption of the original specification in LS_1 and LS_2, in order to get test cases for all relevant scenarios. A result of this modifications was that one of the test cases generated from LS_2 had to be discarded.

This problem is not a problem with the general Abadi/Lamport strategy, but a problem with this concrete SDL representation of the strategy. We believe the problem had not been present if there had been an easy way of letting SDL proceses distinguish between input from different channels.

- Example 6 was made as an example of invalid decomposition, but where verified by both testing tools. On further analysis we realized that the "error" we had planted must be considered a liveness property. Since we only test on safety properties, this example was then counted among the valid ones.

  The assumptions of the original specification and the decomposition are equivalent, but there is made a change in the guarantee of the decomposition relative to the guarantee of the original specification. On input of a specific message m, the original specification should output a message m', while the decomposition should do nothing on input of m.

  A test case was generated from the decomposition with just input of m and no output. This test case was verified by the original specification, which should not be a surprise. The weak liveness of SDL only assert that after the original specification received m, it should *eventually* output m', but since the test case had finale length it could not falsify this property. Because of this we got a strenghtening of the guarantee, since behavior was added and not removed.

  We also made an example where the guarantees of the original specification and the decomposition were switched (example 7). In this example, the decomposition should output m' on input of m, while the original specification should do nothing on this input, so this example is invalid. This example was not validated. From the decomposition a test case with input of m and output of m' was generated. This test case violated the safety property of the original specification that it should *not eventually* output m' after receiving m. In this decomposition we got a weakening of the guarantee, because behavior was removed.

- In example 14 the assumption of the orginal specification and the decomposition are equivalent, while the guarantee of the decomposition is changed (both strenghtend and weakend) relatively to the guarantee of the original specification. The example is valid because the guarantees are equivalent relative to the assumptions, i.e., equivalent in the cases the assumptions are fulfilled. If at the same time the assumption of the decomposition had been weakend relatively to the assumption of the original specification, the example had been invalid and had most likely not been validated. We mention this example because we feel it emphasizes the role of the assumptions.

8.3.2 Efficiency

In the discussion of efficiency we group the examples in accordance to *Grouping 2*. The interesting examples are all in group 1, because it is in this group we observe differences in effectiveness between the two tools. Groups 2 and 3 are discussed in a short note at the end of this section.

In order to find any differences in the efficiency of the two testing tools, we measured the time used for generating and executing test cases and the symbol coverage when test cases were executed (the fraction of the SDL symbols covered after executing the set of generated test cases). In addition we counted states in the examples as a measure of size, counted generated test cases and counted messages in test cases as a measure of length. The results from group 1 are found in table 8.3.

	Group 1	**Ex 1**	**Ex 2**	**Ex 3**	**Ex 4**	**Ex 10**	**Ex 14**	**Ex 15**
	Valid	Y	Y	Y	N	N	Y	N
	No. states	7	15	20	10	14	9	7
Direct	Exec. time	7:25	6:33	11:37	10:08	7:06	0:48	6:57
	Test cases	10	15	14	10	10	5	10
	Av. length	94.4	1.5	121.2	74.8	117.9	41.4	116.4
	Coverage	95.2	70.0	96.1	97.6	88.7	96.4	100.0
AL	Exec. time	0:11	0:16	0:20	0:14	0:13	0:12	0:12
	Test cases	2	6	5	4	2	2	2
	Av. length	4.0	4.3	5.6	2.0	8.0	3.0	4.0
	Coverage	100.0	100.0	97.9	100.0	100.0	100.0	100.0

Table 8.3: Efficiency of testing tools

The numbers in the table is calculated as follows:

- The number of states is the totale number of states in all assumptions and guarantees of each the example. States added when preparing the specifications for testing are not counted. The number of states for each assumption and guarantee are found in table C.1.
- Execution time is the sum of the time used for generating test cases and the time used for executing test cases, and given in minutes and seconds. The way we made the Abadi/Lamport testing tool, test cases are generated for each premiss. Since all premisses generate test cases from the same specification it had been enough to generate the test cases only once (see section 7.4.2). Because of this the execution time of the Abadi/Lamport testing tools is calculated as the maximum test case generation time of any of the premisses plus the test case execution time of all the premisses.
- In the table the number of test cases and the average length of test cases are given. Empty (zero messages) test cases are not counted and not taken into account when average lengths are calculated. For the Abadi/Lamport testing tool the maximum number of test cases generated from any of the premisses are given, and the average length of test cases are from the premiss with the longest test cases.
- Symbol coverage for the Abadi/Lamport testing tool is caluculated as the overall symbol coverage of all the premisses. Symbol coverage is given in percentage.

From the table we observe :

- The Direct testing tool had much longer execution times than the Abadi/Lamport testing tool.
- The Direct testing tool generated more and longer test cases than the Abadi/Lamport testing tool.
- The Direct testing tool gave some lower symbol coverage than the Abadi/Lamport testing tool.

We see a clear trend that the Abadi/Lamport testing tool is more efficient than the Direct testing tool, with both respect to the use of time and the number and length of test cases. We believe the reason for the differences is that the Direct testing tool used considerably time on genrating test cases with chaos.

We also see that there are no loss in symbol coverage from the Direct tool to the Abadi/Lamport tool. When we in addition know that the tools validate exactly the same examples, we can conclude that the "quality" of the tests conducted by the Abadi/Lamport testing tool is not lower than the "quality" of the Direct testing tool. It is hard to see any effect of the size of the examples, neither can we see any effect caused by validity.

Example 2 represents an exception with respect to the length of test cases generated by the Direct testing tool. The reason is that the decomposition of example 2 contains a component that consume the chaos produced by the other componenets without itself being violated. The output was however still chaotic in the sense that the output had no relation with the input after the assumption was violated.

Groups 2 and 3

In the examples of groups 2 and 3 all the decompositions has assumptions equivalent to **true**. As a result there are no differences in the effectiveness of the testing tools, because there are never chaos production when test cases are generated. For group 3, the testing strategies must be seen as equivalent; testing of premisses AL 1.x are superfluous and premiss AL 2 tests exactly the same as the Direct testing tool. The only noteworthy about group 2 is that the symbol coverage of the Direct testing tool is somewhat low. This may be because the chaos production part of the original specification is never covered.

8.3.3 Tracability

By tracability we mean the support for finding errors in a not validated decomposition. With both tools we know which test cases that are not verified, which give some indication of where the errors are. In the Direct testing tool there were no indications of where in the decomposition the errors may be apart from this. There are however functionality in the SDL Validator for showing exactly where in the SDL specification the error occured.

In table 8.2 the results of testing invalid examples are presented, grouped by *Grouping 1*. From the table we observe:

- When the examples of group 5 were tested by the Abadi/Lamport testing tool, they were all falsified by permiss AL 1.x.
- When the examples of groups 4 and 6 were tested by the Abadi/-Lamport testing tool, they were all falsified by permiss AL 2.

In the examples of group 5, the assumption of the decomposition is strengthened relatively to the original specification, while in the examples of groups 4 and 6, the guarantee of the decomposition is weakend relatively to the original specification. This is two different kinds of illegal behavioral refinement.

We see by this that the Abadi/Lamport testing tool reveals the nature of the error by which premiss that is not validated. The Direct testing tool does not provide any similare functionality.

8.4 Conclusions on Testing

We have made a prototype tool for direct testing of decomposition of contract oriented specifications and a prototype tool for testing of decompostion of contract oriented specifications based on the composition principle. Within the restrictions of the tested specifications, both tool work for small examples; valid examples are validated and invalid examples are not validated. Furthermore the two prototype tools validate exactly the same examples. We conclude that the success criteria of section 3.4 are fulfiled.

With these results, we are of the opinion that the hypotheses formulated in sections 3.1.2 and 3.1.3 are validated within the restrictions we have made. We have shown that it is possible to make a method for testing decompostion of contract specifications based on functional testing and the composition principle, and that it is possible to realize it as an automated tool. We have also shown that the method validate decompositions in accordance with the theory of behavioral refinement.

We have empirical results that show that the Abadi/Lamport testing tool is more efficient than the Direct testing tool when the decomposition of a specification has assumptions that are not equivalent to **true**, and that the Abadi/Lamport testing tool does not do tests with lower quality than the Direct testing tool.

We have also results which indicate that the Abadi/Lamport testing tool provides better tracability than the Direct testing tool because it reveals the nature of errors.

Chapter 9

Discussion

We have in this thesis proposed a general strategy, refered to as the Abadi/-Lamport testing strategy, for testing of contract decomposition based on the composition principle and functional testing.

In order to show that this strategy works as predicted, we implemented a restricted version of the strategy in a prototype tool for testing of contract decomposition in SDL, refered to as the Abadi/Lamport testing tool. Testing of this prototype tool, and comparison with another prototype tool based on direct testing of contract decomposition, showed promising results.

In this chapter we try to generalize from the results we have obtained from the work of this thesis.

9.1 Restrictions

In the construction of the prototype testing tools, we made several restrictions compared to the semantics. This means the set of specifications the tool was able to handle is a proper subset of the specifications expressable by the semantics.

Specifications with local variables, feedback and time dependent behavior were omitted. We do not think this significantly reduces the generality of our results.

Local variables in the specifications would increase the state space of the specifications and could lead to state explosion. This is however a well-known problem, and considerable research has been invested in order to find testing techniques that handle this [Lee and Yannakakis, 1996]. For example the state space exploration algorithm used by SDL Validator applies considerable heuristics for reducing the state space [Graboski et al., 1996]. Both prototype tools will suffer from state space explosion under certain conditions. We

believe, however, that the chaos production makes the Direct testing tool more vulnerable to state explosion than the Abadi/Lamport testing tool.

Adding the possibility of feedback would result in more complex behavior. Both the semantics of contract specifications and the semantic formulation of the composition rule are made with handling of feedback in mind. In the proof of the composition rule, feedback is taken into account. Both testing strategies are realizable also with the possibility of feedback, and we cannot see that feedback should have any impact on the relative efficiency of the two testing strategies.

If the testing strategies should be applied to time dependent specifications, this requires specification languages with reasonable representations of time and testing techniques that are able to handle time. In FOCUS, time is represented by ticks that divide streams into time units. FOCUS state transition diagrams and stream processing functions treat the time ticks as a kind of pseudo-messages. If e.g. SDL had the same representation of time, this would lead to larger state spaces, but we cannot see that this would have any particular effect on the Abadi/Lamport strategy in comparison with other testing strategies. We are aware of ongoing research on testing of time dependent systems [Salva et al., 2001].

9.2 Liveness

The composition rule we formulate does not take liveness into account. A liveness property is a property that only can be falsified in infinite time. It is impossible to generate infinitely long test cases in finite time, so generally it is impossible to make a testing tool that decides whether a liveness property is true or false. This is a general restriction to executable specifications, so we will never be able to realize a tool that tests arbitrary liveness properties. There are however testing techniques that address liveness; for example SDL Validator allows the user to specify observer processes that can be used to check whether liveness properties are fulfilled.

9.3 Realization of tool

The restrictions we made when constructing our prototype tools do not prevent us from generalizing from our results; hence the proposed Abadi/-Lamport testing strategy can be realized in a full implementation.

We vision a tool where the input is specifications of components by assumptions in some standardized form, in addition to guarantees and the

connections between components. Our prototype tool required some modifications to the assumption. These modifications, and the setup of the specifications under test, are quite strait forward and schematic, and can be automated without much trouble. In addition it can be argued that the modifications we did to assumption in the prototype Abadi/Lamport tool are unnecessary complicated (because we insisted on letting assumptions recognize instead of simulating input from the environment).

We used SDL as specification language, but the only real requirement on the specification language is that it is executable. Another natural choice of specification language could be the *Unified Modeling Language* (UML) [Rumbaugh et al., 1999], which is probably the most used and fastest growing graphical specification language in the system development industry today. State transition diagrams in UML are heavily based on Statecharts [Harel, 1987]. The biggest conceptual difference between Statecharts and SDL-92 is the hierarchical structure of Statecharts with composite states, i.e., one or more Statecharts inside a state, which are assumed to run in parallel. A way to handle this could be to view composite states as compositions in the FOCUS sense.

There exists proposals on how to generate test cases from UML state transition diagrams, for example [Kim et al., 1999], [Hartmann et al., 2000] and [Offnutt and Abdurazik, 1999]. The first two proposals mentioned are based on flattening Statecharts with composite states to unhierarchical state transition diagrams.

UML has no dataflow diagrams comparable to the block diagrams in SDL. Since UML is an object oriented language, a possible choice for specifying composition could be class diagrams.

Other possible specification languages could be traditional programming languages, as e.g. Java, since programming laguages most certainly are executable. One could even imagine a tool that supports different specification languages, for example a tool that supports both UML and Java. This tool could be used throughout iterative system deveolpment processes. In early iterations, UML specifications could be decomposed into other UML specifications, and in the later iterations Java implementations of components could be integrated in the testing. A tool that supports both UML and SDL could be used in system development processes where UML is used for high level specifications and SDL for lower level specifications and system design.

There are no reasons why the assumption part and the guarantee part of a contract specification have to be specified in the same specification language. We can for example imagine that standard components implemented with Java could be delivered with assumptions written in UML.

During the work of this thesis, our implicit focus has been on devel-

opment of software systems. The composition principle should however be general enough to support a much larger class of systems. The general testing strategy could as well be adopted to development of hardware systems or embedded systems.

The FOCUS semantics we have used for formulating the composition rule, is based on asynchronous communication. Synchronous communication is a specialization of the asynchronous case. It should be possible to realize a tool that validates refinements of asynchronous specifications into synchronous specifications. We see no principal objections to making a testing tool for validation of refinement in development of embedded systems. This is supported by e.g. [Stølen and Fuchs, 1995], where an approach to hardware/software co-design based on FOCUS semantics and contract oriented specifications are presented.

Another application of a tool based on the general testing strategy could be tesing of operator procedures. The only requirement would be operator procedures expressed in an executable specification language.

9.4 Application of Contract Specifications to System Devlopment

We have shown the possibility of relating contract oriented specification to a functional setting. We believe this could be applied in system development, and system development would benefit from it. The main reason is that contract specifications and the composition principle are able to handle modularity. In development of large systems good methods for handling modularity is necessary, or even unavoidable.

The work of this thesis is based on the assumption that contract oriented specifications would have been useful in system development. What we have not addressed directly is whether this assumption is true or not.

We believe the best way to find this out would be to apply a form of contract specifications to a real-life system development project. In such a project a full implementation of the Abadi/Lamport testing tool would have a natural place together with methodology guidelines on how contract specifications should be used.

9.5 Universal Principles

We view the composition principle as one of many universal principles for system deveolpment. This thesis indicates that the Abadi/Lamport principle

can be realized as a practical testing tool. This opens the possibility of translating other rules from the field of formal methods into practical testing tools. One might even consider building a tool that generates such testing tools based on rules from formal methods as input.

Bibliography

M. Abadi and L. Lamport. Composing specifications. Digial System Research Center. Research Report 66, Oct. 1990.

M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. Prog. Lang. Syst.*, 17(3):507–534, May 1995.

B. Beizer. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold Electrical/Computer Science and Engeneering Seris. Van Nostrand Reinhold Company, 1984.

M. Broy and K. Stølen. *Specification and Devlopment of Interactive Systems. FOCUS on Streams, Interface, and Refinement*. Springer-Verlag, 2001.

A. Cau and P. Collette. A unifying approach for shared variable and distributed massage passing concurrency. *Acta Inf.*, 33:153–176, 1996.

A. Church. A set of postulates for the foundation of logic. *The Annals of Mathematics*, 34(4):839–864, Oct. 1933.

O. J. Dahl. *Verifiable Programming*. Prentice Hall, 1992.

F. den Braber. A precise foundation for modular refinement of MSC specifications. Master's thesis, Leiden Institute of Advanced Computer Science, Faculty of Mathematics and Natural Science, Leiden University, 2001.

A. Ek. Verifying Message Sequence Charts with the SDT Validator. In O. Færgemand and A. Sarma, editors, *SDL'93 – Using Objects*, pages 237–249. Elsevier, 1993.

N. Fenton and S. L. Pfleeger. Can formal methods always deliver? *IEEE Computer*, 30(2):34–34, Feb. 1997.

K. Finney. Mathematical notation in formal specifications: Too difficult for the masses? *IEEE Trans. Softw. Eng.*, 22(2):158–159, Feb. 1996.

K. Finney and N. Fenton. Evaluating the effectiveness of Z: The claims made about CICS and where we go from here. *J. Systems Software*, 35(3): 209–216, 1996.

J. Graboski, R. Scheurer, D. Toggweiler, and D. Hogrefe. Dealing with the complexity of state space exploration algorithms for SDL systems. In *Proceedings of the 6th GI/ITG Technical Meeting on 'Formal Description Techniques for Distributed Systems'*, volume 20 of *Arbeitsberichte das Institutes für mathematische Maschinen- und Dataverarbeitung (Matematik)*, 1996.

D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Programming*, 8:231–274, 1987.

J. Hartmann, C. Imoberdorf, and M. Meisinger. UML-based integration testing. In M. J. Harrold, editor, *2000 International Symposium on Software Testing and Analysis (ISSTA00), Proceedings of the ACM SIGSOFT*, volume 25 of *Software Engineering Notes*, pages 60–70, Sept. 2000.

U. Hinkel. *Formale, semantische Fundierung und eine darauf abgestützte Verifikationsmethode für SDL.* PhD thesis, Fakultät für Informatik, Technischen Universität München, 1998a.

U. Hinkel. Verification of SDL specifications on base of a stream semantics. In Y. Lahar, A. Wolisz, J. Fischer, and E. Holz, editors, *Proceedings of the 1st workshop of the SDL Forum Society on SDL and MSC, Volume II*, pages 241–250, 1998b.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–583, Oct. 1969.

C. A. R. Hoare. *Communicating Sequential Processes.* Series in Computer Science. Prentice-Hall International, 1985.

E. Holz and K. Stølen. An attempt to embed a restricted version of SDL as a target language in Focus. In D. Hogrefe and S. Leue, editors, *Proc. Formal Description Techniques VII (FORTE'94)*, pages 324–339. Chapman and Hall, 1995.

ITU-T. *Message Sequence Chart (MSC), ITU-T Recommendation Z.120*, 2000a.

ITU-T. *Specification and description language (SDL), ITU-T, Recommendation Z.100*, 2000b.

C. B. Jones. *Development Methods for Computer Programs. Including a Notion of Interference*. PhD thesis, Oxford University, 1981.

C. B. Jones. The search for tractable ways of reasoning about programs. Department of Computer Science, University of Manchester. Technical Report Series UMCS-92-4-4, 1992.

Y. G. Kim, H. S. Hong, D. H. Bae, and S. D. Cha. Test cases generation from UML state diagrams. *IEE Proc.-Softw.*, 146(4):187–192, Aug. 1999.

B. Koch, J. Grabowski, D. Hogrefe, and M. Schmitt. Autolink. A tool for automatic test generation from SDL specifications. In *IEEE International Workshop on Industrial Strength Formal Specification Techniques (WIFT98)*, 1998.

L. Lamport. The temporal logic of actions. *ACM Trans. Prog. Lang. Syst.*, 16(3):872–923, May 1994.

L. Lamport and F. B. Schneider. The "Hoare Logic" of CSP, and all that. *ACM Trans. Prog. Lang. Syst.*, 6(2):281–296, Apr. 1984.

B. Langefors. *Theoretical Analysis of Information Systems*. Studentlitteratur, Lund, Sweden, fourth edition, 1973. First edition was published in 1966 by the same publisher.

D. Lee and M. Yannakakis. Principles and methods of testing finite state machines—a survey. *Proceedings of the IEEE*, 84(8):1090–1123, Aug. 1996.

J. Misra and K. M. Chandy. Proof of networks of processes. *IEEE Trans. Softw. Eng.*, 7(4):417–425, July 1981.

J. Offnutt and A. Abdurazik. Generating tests from UML specifications. In R. France and B. Rumpe, editors, *UML'99 – The Unified Modeling Language Beyond the Standard, Second International Conference*, number 1723 in Lecture Notes in Computer Science, pages 416–429. Springer-Verlag, 1999.

A. Pnueli and M. Shalev. What is in a step: On the semantics of statecharts. In T. Ho and A. R. Meyer, editors, *Theoretical Aspects of Computer Software, International Conference TACS'91*, number 526 in Lecture Notes in Computer Science, pages 244–264. Springer-Verlag, Sept. 1991.

J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, 1999.

S. Salva, E. Petitjean, and H. Fouchal. A simple approach to testing timed systems. In E. Brinksma and J. Tretmans, editors, *Proceedings of the Workshop on Formal Approaches to Testing of Software FATES'01*, BRICS Notes Series NS-01-4, pages 93–107, 2001.

R. L. Schwartz and T. Christiansen. *Learning Perl*. O'Reilly, second edition, July 1997.

K. Stølen. Assumption/commitment rules for data-flow networks — with an emphasis on completeness. Institut für Inforamatik, Technische Universität München. Sonderforschungsbereich 342/09/95 A, 1995.

K. Stølen. Assumption/commitment rules for dataflow networks — with an emphasis on completeness. In H. R. Nielson, editor, *Proc. 6th European Symposium on Programming (ESOP'96)*, number 1058 in Lecture Notes in Computer Science, pages 356–372. Springer-Verlag, 1996.

K. Stølen and M. Fuchs. A formal method for hardware/software co-design. Institut für Inforamatik, Technische Universität München. Sonderforschungsbereich 342/10/95 A, 1995.

Telelogic. *Telelogic Tau 4.2. User's Manual*, 2001.

W. F. Tichy. Should computer scientists experiment more? *IEEE Computer*, 31(5):32–40, May 1998.

W. F. Tichy, P. Lukowicz, L. Prechelt, and E. A. Heinz. Experimental evaluation in computer science: A quantitative study. *J. Systems Software*, 28: 9–18, 1995.

A. M. Turing. On computable numbers, with an application to the Entschidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.

M. V. Zelkowitz and D. R. Wallace. Experimental models for validating technology. *IEEE Computer*, 31(5):23–31, May 1998.

Appendix A

Programs

This appendix provides the Perl programs used in the prototype tools described in chapter 7. Section A.1 contains the Perl program for Direct testing, and sections A.2 and A.3 contain the Perl programs for testing respectively the first and second premiss of the Abadi/Lamport testing tool.

Section A.4 presents the code of the small C program `starter.exe`, which is used for giving the SDL Validators their first input. In section A.5 the SDL Validator command file for test case generation is shown and in section A.6 a typical SDL Validator command file for test case execution is shown.

A.1 Direct Testing

```
#####################################################################
#                                                                   #
# Direct Testing                                                    #
#                                                                   #
# Verifying                                                         #
#   Spec1 ~> Spec2                                                  #
# or                                                                #
#   Spec2 => Spec1                                                  #
# by generating test cases from Spec2 and executing them            #
# against Spec1                                                     #
#                                                                   #
# Left side: Spec2                                                  #
# Right side: Spec1                                                 #
#                                                                   #
#####################################################################
```

```
# Declarations
my($tc_dir,
   $gen_coms,
   $test_coms,
   $ls,
   $rs,
   $vl_postfix,
   $tc_pref,
   $mins,
   $coverage,
   $no_tc,
   @tc_names,
   $name,
   $new_name,
   $o_pref,
   $n_pref,
   $first_tc,
   $last_tc,
   $test_coverage,
   %tc_exec_res,
   $no_not,
   $no_ver,
   $line,
   $g_res,
   $time,
   $gtime);

# Predifined values
$tc_dir="test_cases";
$gen_coms="testcasegen.com";
$test_coms="testcasetest.com";
$vl_postfix="_vlb.exe";
$tc_pref="TC";
$mins=10;
$coverage=100;
$o_pref="Tree";
$n_pref="Conv";

# Test case counters
$no_tc=0;
$first_tc=0;
$last_tc=0;
$no_not=0;
```

```
$no_ver=0;

$test_coverage=0;

# Get validator prefixes
print STDERR "Left side => Right side\n";
print STDERR "(Right side ~> Left side)\n";
print STDERR "Left side: ";
$ls=<STDIN>;
chomp($ls);
print STDERR "Right side: ";
$rs=<STDIN>;
chomp($rs);
print "\n";
print "DIRECT TESTTING\n\n";
print "of ".$ls." => ".$rs."\n\n";

# Create test case direcroty if not exists
if (! -e $tc_dir) {
    print "Creating directory for test cases: '.\\".$tc_dir."\\'\n";
    mkdir ($tc_dir, 0777)
or die "Could not make test case directory";
}

# Make command file for test case generation
print "Making command file for test case generation: '".$gen_coms."'\n";
open(TCG, ">".$gen_coms)
    or die "Could not open file ".$gen_coms.": ".$!;
print TCG "# Command file for test case generation\n\n";
print TCG "Define-Symbol-Time Undefined\n";
print TCG "Define-MSC-Test-Case-Directory .\\".$tc_dir."\n";
print TCG "Tree-Walk ".$mins." ".$coverage."\n";
print TCG "Save-Reports-As-MSC-Test-Cases TreeWalk ".$tc_pref."\n";
print TCG "Exit no";

close(TCG) or die "Could not close file ".$gen_coms.": ".$!;

# Launch left side validator and execute commands for
# test case generation
print "Generating test cases";

$gtime=time();
$g_res="";
```

```
open(LSV, "starter.exe ".$gen_coms."|".$ls.$vl_postfix."|")
    or die "Could not launch ".$ls.$vl_postfix.": ".$!;


# Count test cases and extract test case names
while (<LSV>) {
    if (/Test report \#(\d+) of length (\d+) added./) {
print ".";
    }
    if (/MSC test case is saved in file '\.\\$tc_dir\\(.+)'./) {
$tc_names[$no_tc]=$1;
$no_tc++;
    }
    if (/States: .*/) {
$g_res=$_;
    }

}
print "\n";

close(LSV)
    or die "Could not close ".$ls.$vl_postfix.": ".$!;

$gtime=time()-$gtime;

if ($tc_names[0] =~ /$tc_pref_Tree_(\d+).mpr/) {
$first_tc=$1;
}
if ($tc_names[$no_tc-1] =~ /$tc_pref_Tree_(\d+).mpr/) {
$last_tc=$1;
}

# Converting test cases so right side will be able to read them
print "Converting test cases";
foreach $name (@tc_names) {
    $new_name = $name;
    $new_name =~ s/$o_pref/$n_pref/;

    open(TCI, ".\\".$tc_dir."\\".$name)
or die "Could not open file ".$name.": ".$!;
    open(TCO, ">.\\".$tc_dir."\\".$new_name)
or die "Could not open file ".$new_name.": ".$!;
```

```
    while ($line = <TCI>) {
$line =~ s/$ls/$rs/g;
print TCO $line;
    }
    close(TCI)
or die "Could not close file ".$name.": ".$!;
    close(TCO)
or die "Could not close file ".$new_name.": ".$!;
    print ".";
}

print "\n";


# Make command file for test case execution
print "Making command file for test case execution: '".$test_coms."'\n";
open(TCT, ">".$test_coms)
    or die "Could not open file ".$test_coms.": ".$!;

print TCT "# Command file for test case execution\n\n";
print TCT "Define-Symbol-Time Undefined\n";

foreach $name (@tc_names) {
    $name =~ s/$o_pref/$n_pref/;
    print TCT "Define-Root original\n";
    print TCT "Verify-MSC .\\".$tc_dir."\\".$name."\n";

}
print TCT "Exit no";

close(TCT)
    or die "Could not close file ".$test_coms.": ".$!;

# Launch right side validator and execute command
# for test case execution
print "Executing test cases";
$time=time();
open(RSV, "starter.exe ".$test_coms."|".$rs.$vl_postfix."|")
    or die "Could not launch ".$ls.$vl_postfix.": ".$!;

while (<RSV>) {
```

```
# Analyse test case execution
    if (/MSC (.+) loaded./) {
  $name=$1;
        print ".";
    }
    if (/Symbol coverage :\s+(.+)/) {
  $test_coverage=$1;
    }
    if (/\*\* MSC $name verified \*\*/) {
        $tc_exec_res{$name}="YES";
        $no_ver++;
    }
    if (/\*\* MSC $name NOT VERIFIED \*\*/) {
        $tc_exec_res{$name}="NO";
        $no_not++;
    }


}
print "\n";

close(RSV)
    or die "Could not close ".$rs.$vl_postfix.": ".$!;

$time=time()-$time;
# Print report

print "\nRESULT\n\n";

print $no_tc." test cases was generated and placed in '.\\".$tc_dir."\\'\n";

print "First: ".$tc_pref."_Tree_".$first_tc."\n";
print "Last:  ".$tc_pref."_Tree_".$last_tc."\n\n";

if ($g_res ne "") {
    print "".$g_res."\n";
}
else {
    $mins=int($gtime/60);
    $gtime = $gtime - $mins*60;
    print " Time used: ".$mins.":".$gtime."\n\n";
}
print "".($no_ver+$no_not)." test cases was executed:\n";
```

```
print "  Symbol coverage was ".$test_coverage."%\n";
print "  ".$no_ver." was verified\n";
print "  ".$no_not." was NOT verified\n\n";

$mins = int($time/60);
$time = $time - $mins*60;

print "  Execution time: ".$mins.":".$time."\n\n";

foreach $name (sort keys(%tc_exec_res)) {
    print $name." : ".$tc_exec_res{$name}."\n";
}
```

A.2 Abadi/Lamport Testing, premiss 1

```
######################################################################
#                                                                    #
# Abadi/Lamport testing                                              #
# Premiss 1                                                          #
#                                                                    #
# Verifying                                                          #
#   A /\ (/\Gj) => Ak                                                #
# by generating test cases from A /\ (/\Gj) and executing them       #
# against Ak                                                         #
#                                                                    #
# Left side: A /\ (/\Gj)                                             #
# Right side: Ak                                                     #
#                                                                    #
######################################################################

# Declarations
my($tc_dir,
   $gen_coms,
   $test_coms,
   $ls,
   $rs,
   $vl_postfix,
   $tc_pref,
   $mins,
   $coverage,
   $no_tc,
   @tc_names,
```

```
   $name,
   $new_name,
   $tc_name,
   $o_pref,
   $n_pref,
   $first_tc,
   $last_tc,
   $test_coverage,
   %tc_exec_res,
   $no_not,
   $no_ver,
   $err,
   $has_err,
   @inputs,
   @outputs,
   $no_inputs,
   $no_outputs,
   $line,
   %inputsh,
   %outputsh,
   $chan,
   @lines,
   $no_lines,
   $i,
   $g_res,
   $time,
   $gtime
   );

# Predifined values
$tc_dir="test_cases";
$gen_coms="testcasegen.com";
$test_coms="testcasetest.com";
$vl_postfix="_vlb.exe";
$tc_pref="TC";
$mins=10;
$coverage=100;
$o_pref="Tree";
$n_pref="Conv";
$err="err";
$no_inputs=0;
$no_outputs=0;
```

```
# Test case counters
$no_tc=0;
$first_tc=0;
$last_tc=0;
$no_not=0;
$no_ver=0;

$test_coverage=0;

# Get validator prefixes
print STDERR "Left side: ";
$ls=<STDIN>;
chomp($ls);
print STDERR "Right side: ";
$rs=<STDIN>;
chomp($rs);

print STDERR "Input channels: ";
$line = <STDIN>;
chomp($line);
@inputs=split(/ +/,$line);

foreach $chan (@inputs) {
    $no_inputs++;
    $inputsh{$chan}=1;
}

print STDERR "Output channels: ";
$line=<STDIN>;
chomp($line);
@outputs=split(/ +/,$line);

foreach $chan (@outputs) {
    $no_outputs++;
    $outputsh{$chan}=1;
}

print "\n";
print "\n";
print "ABADI/LAMPORT TESTING, 1. PREMISS\n\n";
print "of ".$ls." => ".$rs."\n\n";
```

```
# Make test case direcroty if not exists
if (! -e $tc_dir) {
    print "Making directory for test cases: '.\\".$tc_dir."\\'\n";
    mkdir ($tc_dir, 0777)
or die "Could not make test case directory";
}

# Make command file for test case generation
print "Making command file for test case generation: '".$gen_coms."'\n";
open(TCG, ">".$gen_coms)
    or die "Could not open file ".$gen_coms.": ".$!;
print TCG "# Command file for test case generation\n\n";
print TCG "Define-Symbol-Time Undefined\n";
print TCG "Define-MSC-Test-Case-Directory .\\".$tc_dir."\n";
print TCG "Tree-Walk ".$mins." ".$coverage."\n";
print TCG "Save-Reports-As-MSC-Test-Cases TreeWalk ".$tc_pref."\n";
print TCG "Exit no";

close(TCG) or die "Could not close file ".$gen_coms.": ".$!;

# Launch left side validator and execute commands for
# test case generation
print "Generating test cases";

$gtime=time();
$g_res="";


open(LSV, "starter.exe ".$gen_coms."|".$ls.$vl_postfix."|")
    or die "Could not launch ".$ls.$vl_postfix.": ".$!;


# Count test cases and extract test case names
while (<LSV>) {
    if (/Test report \#(\d+) of length (\d+) added./) {
print ".";
    }
    if (/MSC test case is saved in file '\.\\$tc_dir\\(.+)'./) {
$tc_names[$no_tc]=$1;
$no_tc++;
    }
    if (/States: \d*/) {
$g_res=$_;
```

```
    }


}
print "\n";

close(LSV)
    or die "Could not close ".$ls.$vl_postfix.": ".$!;

$gtime=time()-$gtime;


if ($tc_names[0] =~ /$tc_pref_Tree_(\d+).mpr/) {
$first_tc=$1;
}
if ($tc_names[$no_tc-1] =~ /$tc_pref_Tree_(\d+).mpr/) {
$last_tc=$1;
}

# Filtering out err messages, fault input and input after err
# Converting test cases so right side will be able to read them
print "Filtering and converting test cases";
foreach $name (@tc_names) {
    $has_err = 0;
    $no_lines=0;
    if ($name =~ /$tc_pref_$o_pref_(\d+).mpr/) {
$tc_name = $tc_pref."_".$o_pref."_".$1;
    }
    $new_name = $name;
    $new_name =~ s/$o_pref/$n_pref/;

    open(TCI, ".\\".$tc_dir."\\".$name)
or die "Could not open file ".$name.": ".$!;

    while (<TCI>) {
if (/msc (.+);/) {
    $lines[$no_lines]=$_;
    $no_lines++;
}
elsif (/endmsc;/) {
    $lines[$no_lines]=$_;
    $no_lines++;
}
```

```
elsif (/(.+) : instancehead;/) {
    if ($1 eq $ls) {
$lines[$no_lines]=$rs." : instancehead;\n";
$no_lines++;
    }
    elsif ($inputsh{$1}==1 || $outputsh{$1}==1) {
$lines[$no_lines]=$_;
$no_lines++;
    }
}
elsif (/.*$err.*/) {
    if (!($has_err==1)) {
$has_err = 1;
$no_lines-=2;
# invariant: fault message was last message
    }
}
elsif (/(.+) : out (.+) to (.+);/) {
    if ($1 eq $ls) {
if ($inputsh{$3}==1) {
    # input to rs
    $lines[$no_lines]=$rs." : in ".$2." from ".$3.";\n";
    $no_lines++;
}
if ($outputsh{$3}==1 && !$has_err) {
    # output from rs
    $lines[$no_lines]=$rs." : out ".$2." to ".$3.";\n";
    $no_lines++;
}
    }
    elsif ($inputsh{$1}==1) {
# If not $1==$ls, then $3==$ls
# input to rs
$lines[$no_lines]=$1." : out ".$2." to ".$rs.";\n";
$no_lines++;

    }
    elsif ($outputsh{$1}==1 && !$has_err) {
# If not $1==$ls, then $3==$ls
# output from rs
$lines[$no_lines]=$1." : in ".$2." from ".$rs.";\n";
$no_lines++;
    }
```

```
}
elsif (/(.+) : in (.+) from (.+);/) {
    if ($1 eq $ls) {
if ($inputsh{$3}==1) {
    # input to rs
    $lines[$no_lines]=$rs." : in ".$2." from ".$3.";\n";
    $no_lines++;
}
if ($outputsh{$3}==1 && !$has_err) {
    # output from rs
    $lines[$no_lines]=$rs." : out ".$2." to ".$3.";\n";
    $no_lines++;

}
    }
    elsif ($inputsh{$1}==1) {
# input to rs
# If not $1==$ls, then $3==$ls
        $lines[$no_lines]=$1." : out ".$2." to ".$rs.";\n";
$no_lines++;

    }
    elsif ($outputsh{$1}==1 && !$has_err) {
# output from rs
# If not $1==$ls, then $3==$ls
$lines[$no_lines]=$1." : in ".$2." from ".$rs.";\n";
$no_lines++;
    }

}
    }

    close(TCI)
or die "Could not close file ".$name.": ".$!;

    open(TCO, ">.\\".$tc_dir."\\".$new_name)
or die "Could not open file ".$new_name.": ".$!;

    for ($i=0; $i<$no_lines; $i++) {
print TCO $lines[$i];

    }
```

```
    close(TCO)
or die "Could not close file ".$new_name.": ".$!;

    print ".";
}

print "\n";

# Make command file for test case execution
print "Making command file for test case execution: '".$test_coms."'\n";
open(TCT, ">".$test_coms)
    or die "Could not open file ".$test_coms.": ".$!;

print TCT "# Command file for test case execution\n\n";
print TCT "Define-Symbol-Time Undefined\n";

foreach $name (@tc_names) {
    if ($name =~ /$tc_pref_$o_pref_(\d+).mpr/) {
$tc_name = $tc_pref."_".$o_pref."_".$1;
    }
    $name =~ s/$o_pref/$n_pref/;
    print TCT "Define-Root original\n";
    print TCT "Verify-MSC .\\".$tc_dir."\\".$name."\n";


}
print TCT "Exit no";

close(TCT)
    or die "Could not close file ".$test_coms.": ".$!;

# Launch right side validator and execute command
# for test case execution
print "Executing test cases";
$time=time();
open(RSV, "starter.exe ".$test_coms."|".$rs.$vl_postfix."|")
    or die "Could not launch ".$ls.$vl_postfix.": ".$!;

while (<RSV>) {

# Analyse test case execution
    if (/MSC (.+) loaded./) {
```

```
  $name=$1;
        print ".";
    }
    if (/Symbol coverage :\s+(.+)/) {
  $test_coverage=$1;
    }
    if (/\*\* MSC $name verified \*\*/) {
        $tc_exec_res{$name}="YES";
        $no_ver++;
    }
    if (/\*\* MSC $name NOT VERIFIED \*\*/) {
        $tc_exec_res{$name}="NO";
        $no_not++;
    }


}
print "\n";

close(RSV)
    or die "Could not close ".$rs.$vl_postfix.": ".$!;

$time=time()-$time;

# Print report

print "\nRESULT\n\n";

print $no_tc." test cases was generated and placed in '.\\".$tc_dir."\\'\n";

print "First: ".$tc_pref."_Tree_".$first_tc."\n";
print "Last:  ".$tc_pref."_Tree_".$last_tc."\n\n";

if ($g_res ne "") {
    print "".$g_res."\n";
}
else {
    $mins=int($gtime/60);
    $gtime = $gtime - $mins*60;
    print " Time used: ".$mins.":".$gtime."\n\n";
}
```

```
print "".($no_ver+$no_not)." test cases was executed:\n";
print "  Symbol coverage was ".$test_coverage."%\n";
print "  ".$no_ver." was verified\n";
print "  ".$no_not." was NOT verified\n\n";

$mins = int($time/60);
$time = $time - $mins*60;

print "  Execution time: ".$mins.":".$time."\n\n";

foreach $name (sort keys(%tc_exec_res)) {
    print $name." : ".$tc_exec_res{$name}."\n";
}
```

A.3 Abadi/Lamport Testing, premiss 2

```
#########################################################################
#                                                                       #
# Abadi/Lamport testing                                                 #
# Premiss 2                                                             #
#                                                                       #
# Verifying                                                             #
#   A /\ (/\Gj) => G                                                    #
# by generating test cases from A /\ (/\Gj) and executing them          #
# against G                                                             #
#                                                                       #
# Left side: A /\ (/\Gj)                                                #
# Right side: G                                                         #
#                                                                       #
#########################################################################

# Declarations
my($tc_dir,
   $gen_coms,
   $test_coms,
   $ls,
   $rs,
   $vl_postfix,
   $tc_pref,
   $mins,
   $coverage,
   $no_tc,
```

```
   @tc_names,
   $name,
   $new_name,
   $tc_name,
   $o_pref,
   $n_pref,
   $first_tc,
   $last_tc,
   $test_coverage,
   %tc_exec_res,
   $no_not,
   $no_ver,
   $err,
   $has_err,
   $no_lines,
   @lines,
   $i,
   $g_res,
   $time,
   $gtime);

# Predifined values
$tc_dir="test_cases";
$gen_coms="testcasegen.com";
$test_coms="testcasetest.com";
$vl_postfix="_vlb.exe";
$tc_pref="TC";
$mins=10;
$coverage=100;
$o_pref="Tree";
$n_pref="Conv";
$err="err";

# Test case counters
$no_tc=0;
$first_tc=0;
$last_tc=0;
$no_not=0;
$no_ver=0;

$test_coverage=0;

# Get validator prefixes
```

```
print STDERR "Left side: ";
$ls=<STDIN>;
chomp($ls);
print STDERR "Right side: ";
$rs=<STDIN>;
chomp($rs);
print "\n";
print "ABADI/LAMPORT TESTING, 2. PREMISS\n\n";
print "of ".$ls." => ".$rs."\n\n";

# Make test case direcroty if not exists
if (! -e $tc_dir) {
    print "Making directory for test cases: '.\\".$tc_dir."\\'\n";
    mkdir ($tc_dir, 0777)
or die "Could not make test case directory";
}

# Make command file for test case generation
print "Making command file for test case generation: '".$gen_coms."'\n";
open(TCG, ">".$gen_coms)
    or die "Could not open file ".$gen_coms.": ".$!;
print TCG "# Command file for test case generation\n\n";
print TCG "Define-Symbol-Time Undefined\n";
print TCG "Define-MSC-Test-Case-Directory .\\".$tc_dir."\n";
print TCG "Tree-Walk ".$mins." ".$coverage."\n";
print TCG "Save-Reports-As-MSC-Test-Cases TreeWalk ".$tc_pref."\n";
print TCG "Exit no";

close(TCG) or die "Could not close file ".$gen_coms.": ".$!;

# Launch left side validator and execute commands for
# test case generation
print "Generating test cases";

$gtime=time();
$g_res="";


open(LSV, "starter.exe ".$gen_coms."|".$ls.$vl_postfix."|")
    or die "Could not launch ".$ls.$vl_postfix.": ".$!;


# Count test cases and extract test case names
```

```
while (<LSV>) {
    if (/Test report \#(\d+) of length (\d+) added./) {
print ".";
    }
    if (/MSC test case is saved in file '\.\\$tc_dir\\(.+)'./) {
$tc_names[$no_tc]=$1;
$no_tc++;
    }
    if (/States: \d*/) {
$g_res=$_;
    }


}
print "\n";

close(LSV)
    or die "Could not close ".$ls.$vl_postfix.": ".$!;

$gtime=time()-$gtime;


if ($tc_names[0] =~ /$tc_pref_Tree_(\d+).mpr/) {
$first_tc=$1;
}
if ($tc_names[$no_tc-1] =~ /$tc_pref_Tree_(\d+).mpr/) {
$last_tc=$1;
}

# Filtering out err messages, fault input and input after err
# Converting test cases so right side will be able to read them
print "Filtering and converting test cases";
foreach $name (@tc_names) {
    $has_err = 0;
    if ($name =~ /$tc_pref_$o_pref_(\d+).mpr/) {
$tc_name = $tc_pref."_".$o_pref."_".$1;
    }
    $new_name = $name;
    $new_name =~ s/$o_pref/$n_pref/;

    open(TCI, ".\\".$tc_dir."\\".$name)
or die "Could not open file ".$name.": ".$!;
```

```
    $no_lines=0;

    while (<TCI>) {
if (/.*$err.*/) {
    if (!$has_err) {
$has_err = 1;
$no_lines-=2;
    }
}
elsif (!($has_err && (/.*: out .* to $ls/ or /$ls : in .*/))) {
    s/$ls/$rs/g;
    $lines[$no_lines]=$_;
    $no_lines++;
}
    }
    close(TCI)
or die "Could not close file ".$name.": ".$!;

    open(TCO, ">.\\".$tc_dir."\\".$new_name)
or die "Could not open file ".$new_name.": ".$!;

    for ($i=0; $i<$no_lines; $i++) {
print TCO $lines[$i];

    }

    close(TCO)
or die "Could not close file ".$new_name.": ".$!;

    print ".";
}

print "\n";

# Make command file for test case execution
print "Making command file for test case execution: '".$test_coms."'\n";
open(TCT, ">".$test_coms)
    or die "Could not open file ".$test_coms.": ".$!;

print TCT "# Command file for test case execution\n\n";
print TCG "Define-Symbol-Time Undefined\n";

foreach $name (@tc_names) {
```

```
    if ($name =~ /$tc_pref_$o_pref_(\d+).mpr/) {
$tc_name = $tc_pref."_".$o_pref."_".$1;
    }
    $name =~ s/$o_pref/$n_pref/;
    print TCT "Define-Root original\n";
    print TCT "Verify-MSC .\\".$tc_dir."\\".$name."\n";

}
print TCT "Exit no";

close(TCT)
    or die "Could not close file ".$test_coms.": ".$!;

# Launch right side validator and execute command
# for test case execution
print "Executing test cases";
$time=time();
open(RSV, "starter.exe ".$test_coms."|".$rs.$vl_postfix."|")
    or die "Could not launch ".$ls.$vl_postfix.": ".$!;

while (<RSV>) {

# Analyse test case execution
    if (/MSC (.+) loaded./) {
  $name=$1;
        print ".";
    }
    if (/Symbol coverage :\s+(.+)/) {
  $test_coverage=$1;
    }
    if (/\*\* MSC $name verified \*\*/) {
        $tc_exec_res{$name}="YES";
        $no_ver++;
    }
    if (/\*\* MSC $name NOT VERIFIED \*\*/) {
        $tc_exec_res{$name}="NO";
        $no_not++;
    }


}
print "\n";
```

```
close(RSV)
    or die "Could not close ".$rs.$vl_postfix.": ".$!;

$time=time()-$time;

# Print report

print "\nRESULT\n\n";

print $no_tc." test cases was generated and placed in '.\\".$tc_dir."\\'\n";

print "First: ".$tc_pref."_Tree_".$first_tc."\n";
print "Last:  ".$tc_pref."_Tree_".$last_tc."\n\n";

if ($g_res ne "") {
    print "".$g_res."\n";
}
else {
    $mins=int($gtime/60);
    $gtime = $gtime - $mins*60;
    print " Time used: ".$mins.":".$gtime."\n\n";
}


print "".($no_ver+$no_not)." test cases was executed:\n";
print "  Symbol coverage was ".$test_coverage."%\n";
print "  ".$no_ver." was verified\n";
print "  ".$no_not." was NOT verified\n\n";

$mins = int($time/60);
$time = $time - $mins*60;

print "  Execution time: ".$mins.":".$time."\n\n";

foreach $name (sort keys(%tc_exec_res)) {
    print $name." : ".$tc_exec_res{$name}."\n";
}
```

A.4 Starter

```
#include <stdio.h>

int main(int argc, char *argv[]) {

  if (argc >= 2) {
    printf("Include-File %s\n",argv[1]);
    printf("\n");
  }


  return 0;
}
```

A.5 Command File, Test Case Generation

```
# Command file for test case generation

Define-Symbol-Time Undefined
Define-MSC-Test-Case-Directory .\test_cases
Tree-Walk 10 100
Save-Reports-As-MSC-Test-Cases TreeWalk TC
Exit no
```

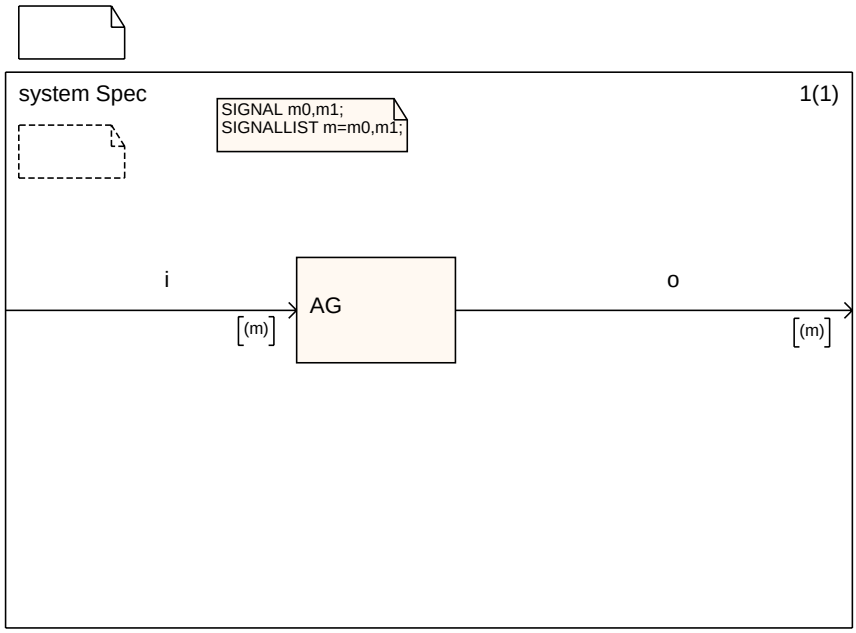
A.6 Command File, Test Case Execution

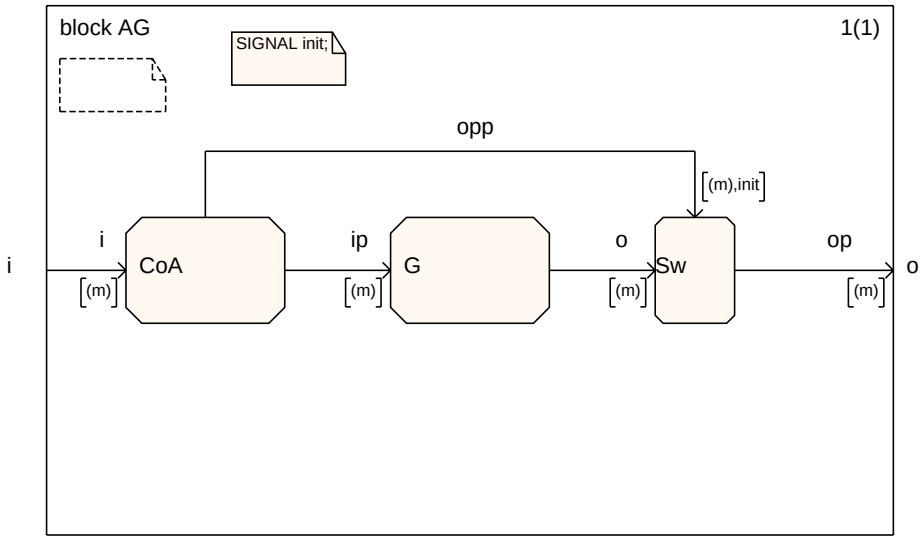
```
# Command file for test case execution

Define-Root original
Verify-MSC .\test_cases\TC_Conv_00013.mpr
Define-Root original
Verify-MSC .\test_cases\TC_Conv_00014.mpr
Define-Root original
Verify-MSC .\test_cases\TC_Conv_00015.mpr
Define-Root original
Verify-MSC .\test_cases\TC_Conv_00016.mpr
Exit no
```

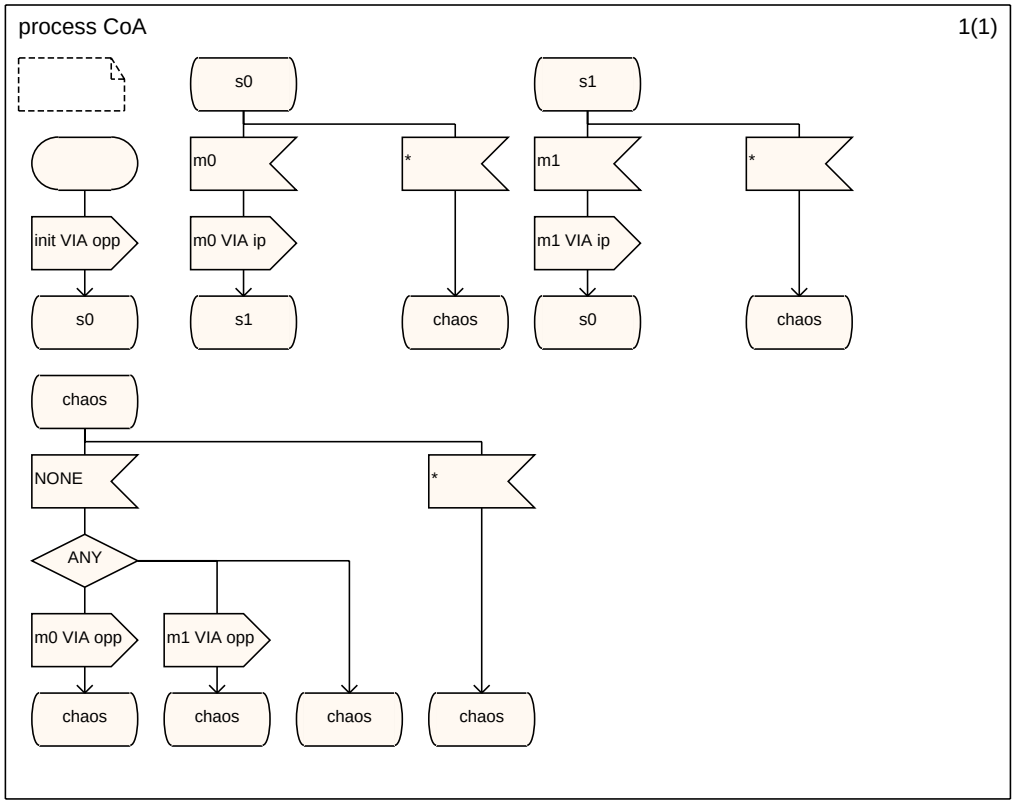
Appendix B

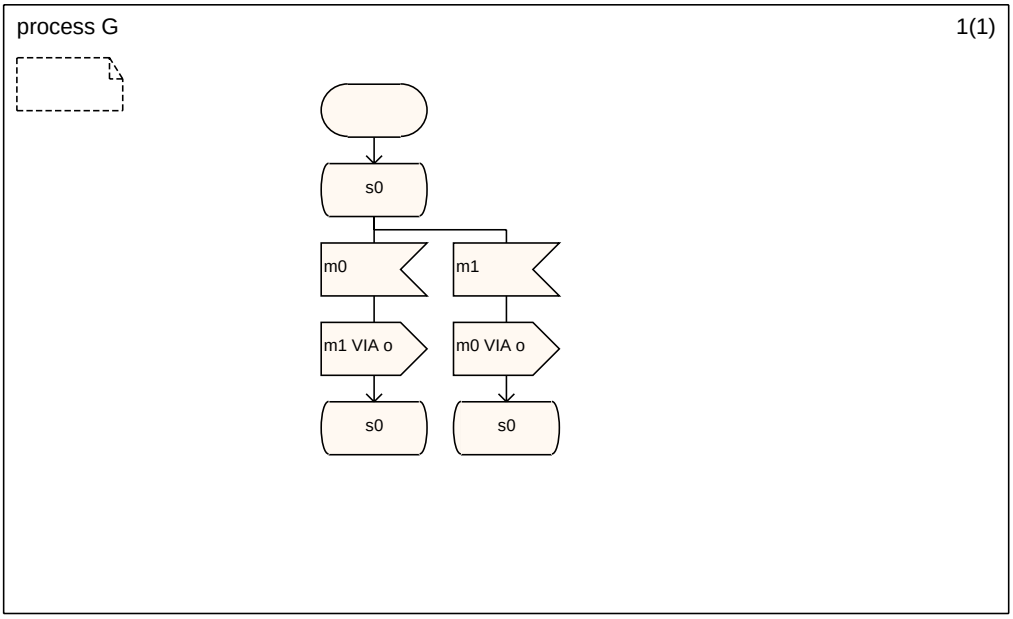
Example

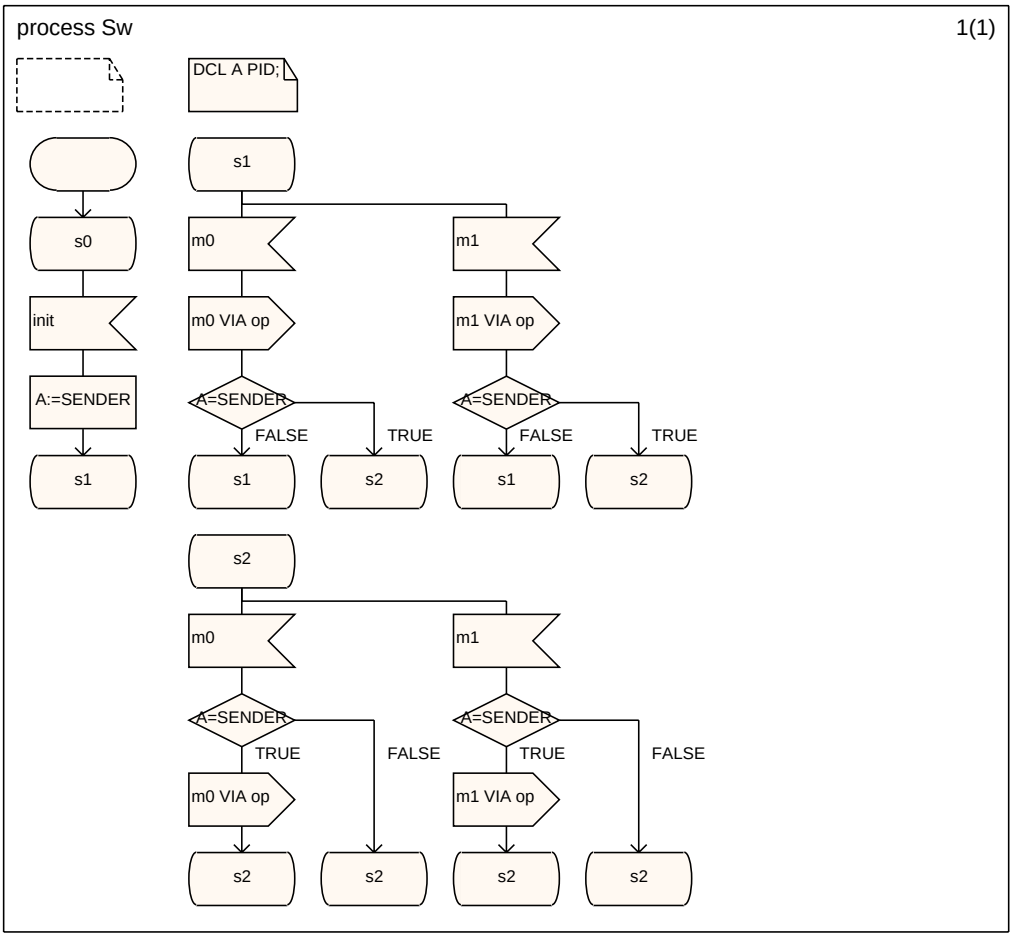
In this appendix the full example decomposition number 8 is presented.

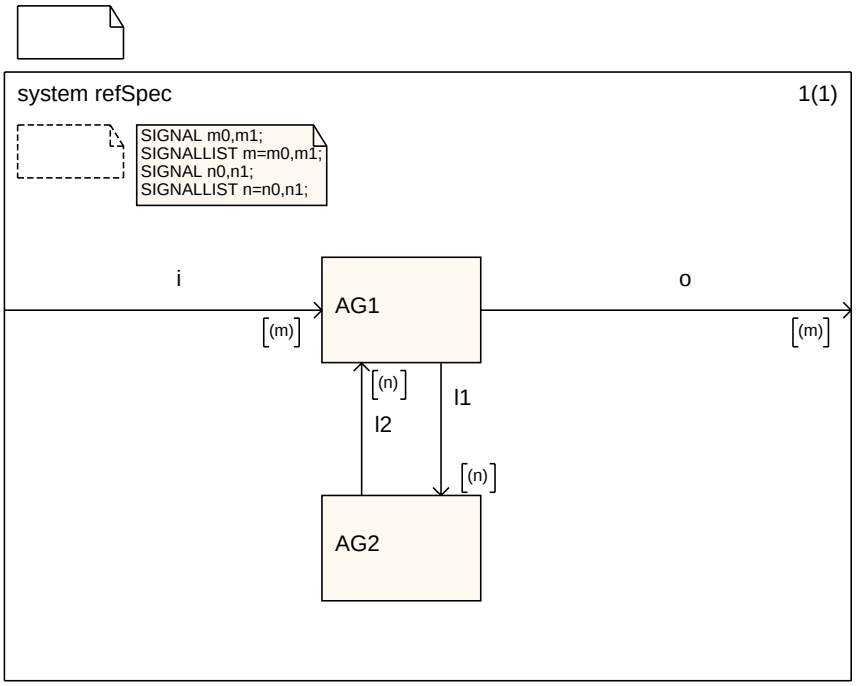


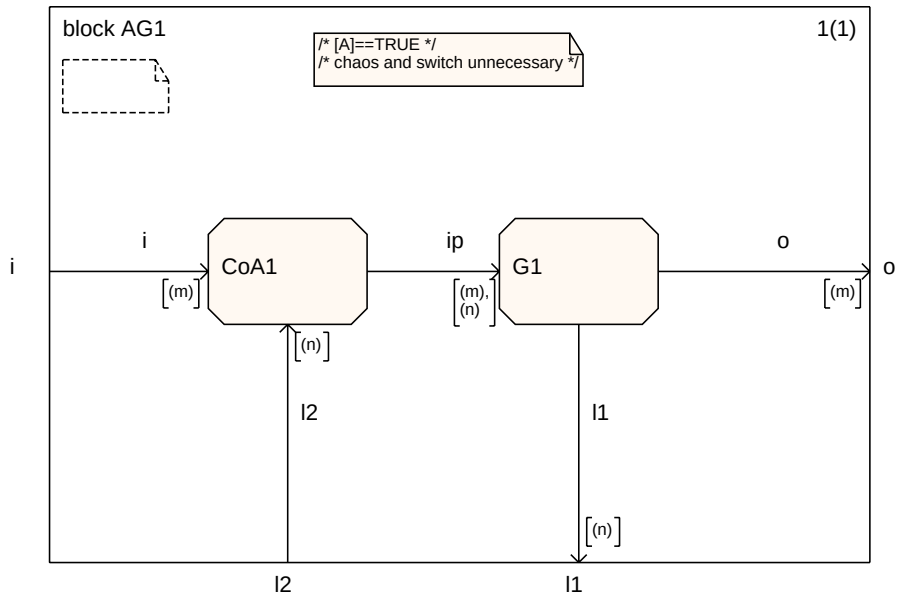

block AG
1(1)
SIGNAL init;
opp
(m),init
i
i
CoA
ip
G
o
Sw
op
o
(m)
(m)
(m)
(m)

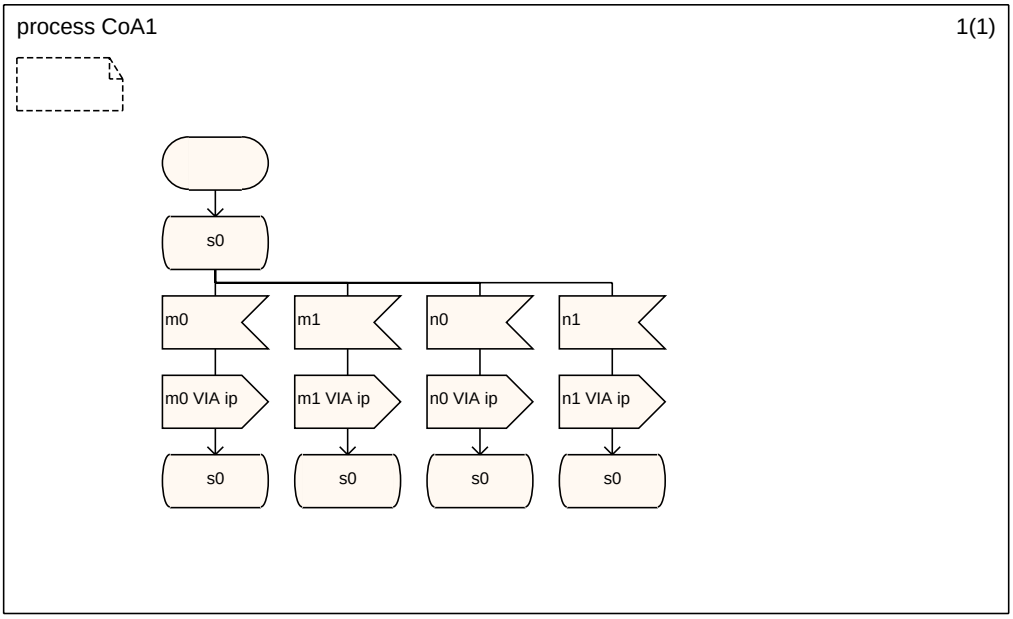

process CoA
1(1)
s0
s1
m0
*
m1
*
init VIA opp
m0 VIA ip
m1 VIA ip
s0
s1
chaos
s0
chaos
chaos
NONE
*
ANY
m0 VIA opp
m1 VIA opp
chaos
chaos
chaos
chaos

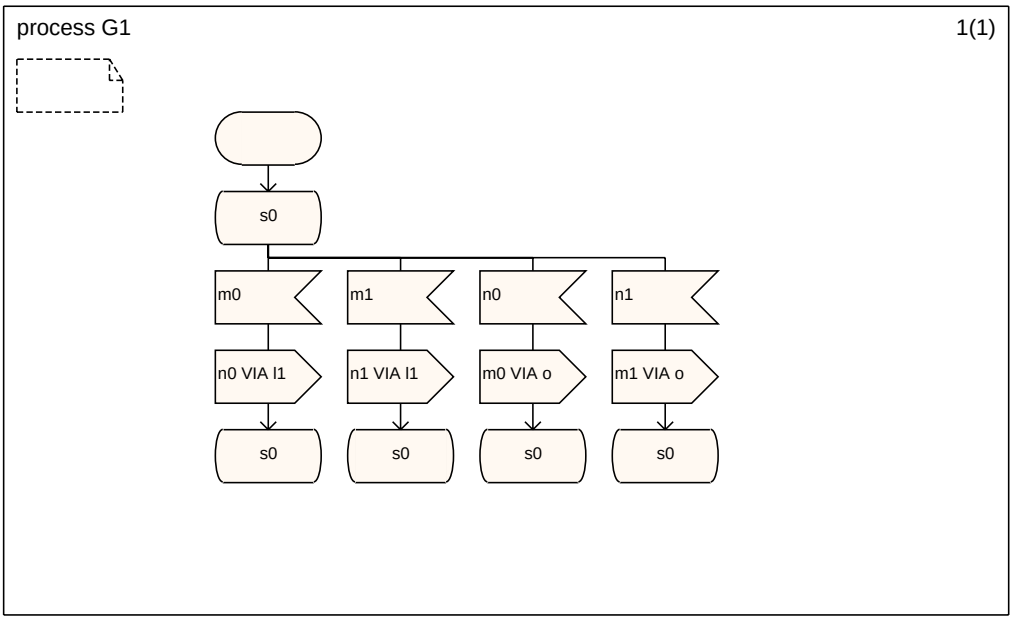

process G
1(1)
s0
m0
m1
m1 VIA o
m0 VIA o
s0
s0

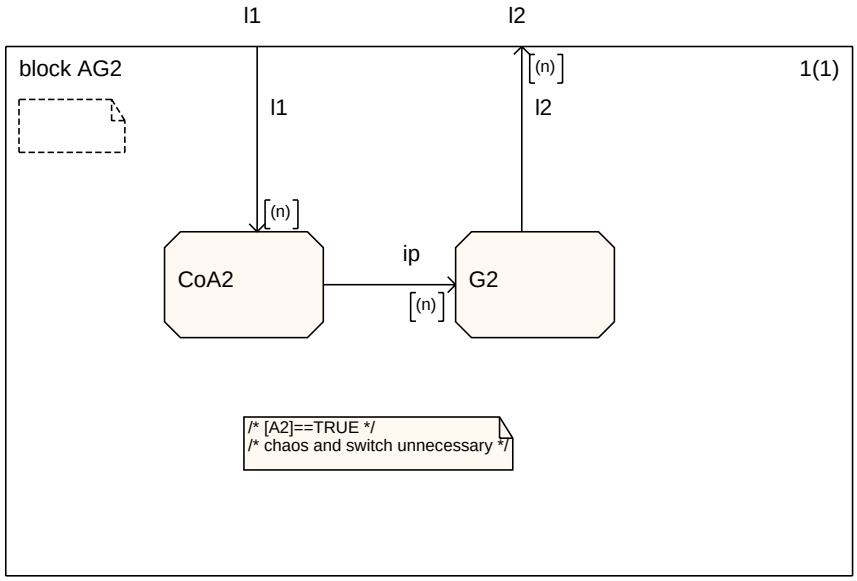

process Sw
1(1)
DCL A PID;
s0
init
A:=SENDER
s1
s1
m0
m1
m0 VIA op
m1 VIA op
A=SENDER
A=SENDER
FALSE
TRUE
FALSE
TRUE
s1
s2
s1
s2
s2
m0
m1
A=SENDER
A=SENDER
TRUE
FALSE
TRUE
FALSE
m0 VIA op
m1 VIA op
s2
s2
s2
s2

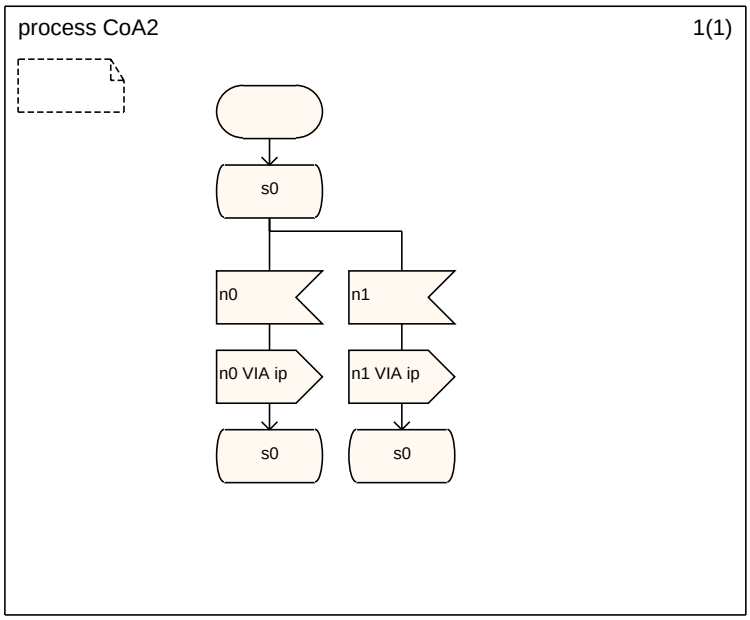

system refSpec
1(1)
SIGNAL m0,m1;
SIGNALLIST m=m0,m1;
SIGNAL n0,n1;
SIGNALLIST n=n0,n1;
i
AG1
o
(m)
(m)
(n)
l1
l2
(n)
AG2

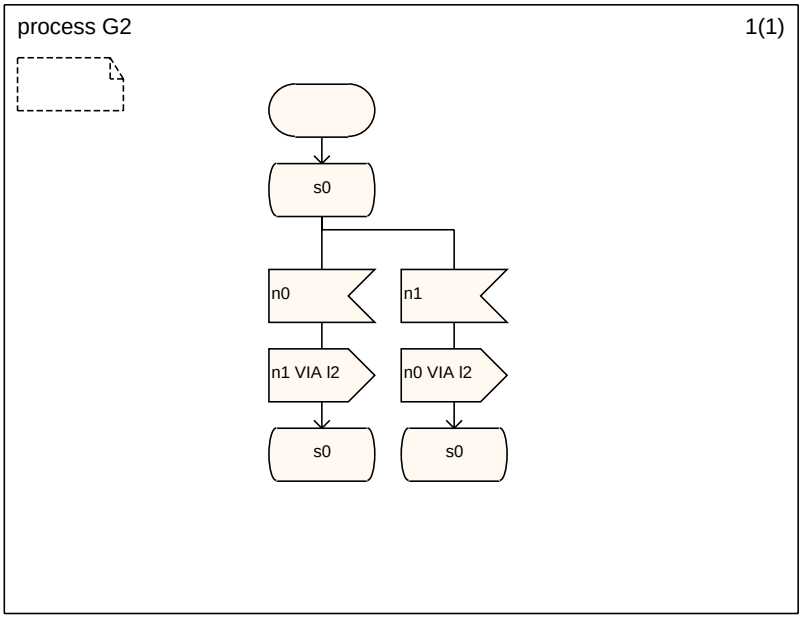

block AG1
1(1)
/* [A]==TRUE */
/* chaos and switch unnecessary */
i
i
CoA1
ip
G1
o
o
(m)
(m),
(n)
(m)
(n)
l2
l1
(n)
l2
l1

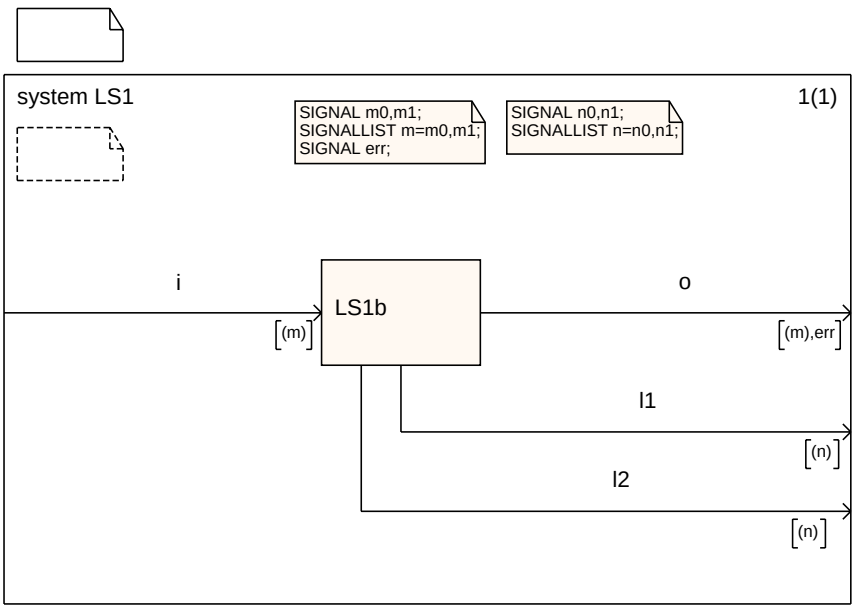

process CoA1
1(1)
s0
m0
m1
n0
n1
m0 VIA ip
m1 VIA ip
n0 VIA ip
n1 VIA ip
s0
s0
s0
s0

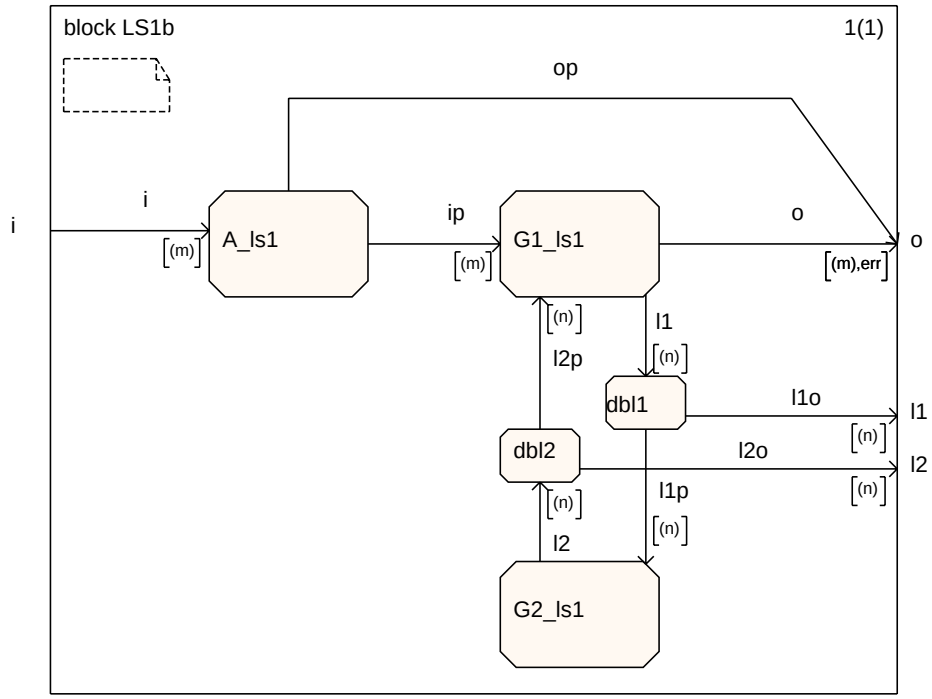

process G1
1(1)
s0
m0
m1
n0
n1
n0 VIA l1
n1 VIA l1
m0 VIA o
m1 VIA o
s0
s0
s0
s0

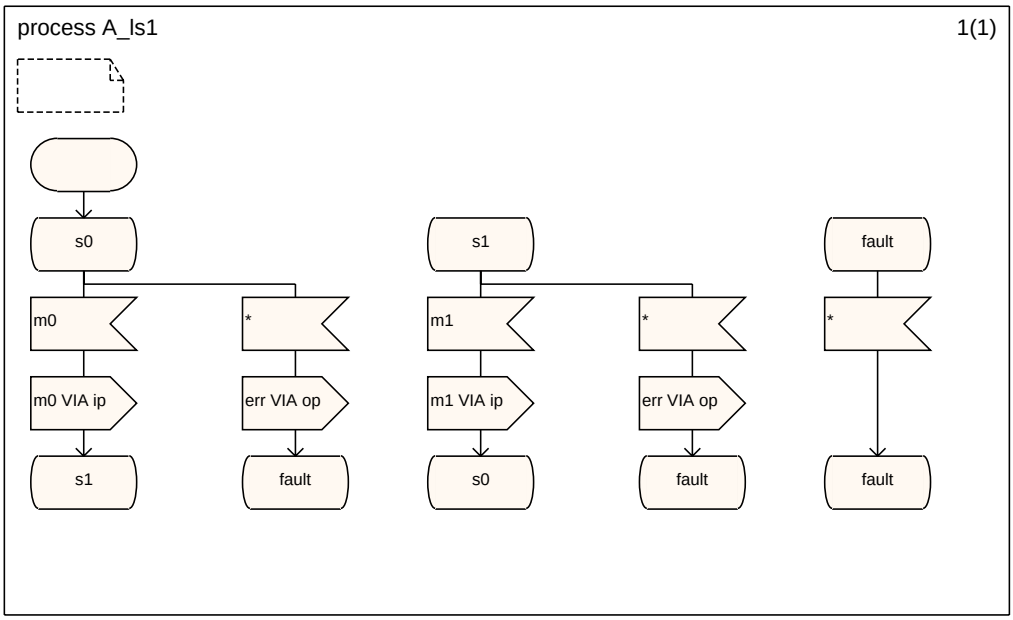

l1
l2
block AG2
1(1)
(n)
l1
l2
(n)
CoA2
ip
G2
(n)
/* [A2]==TRUE */
/* chaos and switch unnecessary */

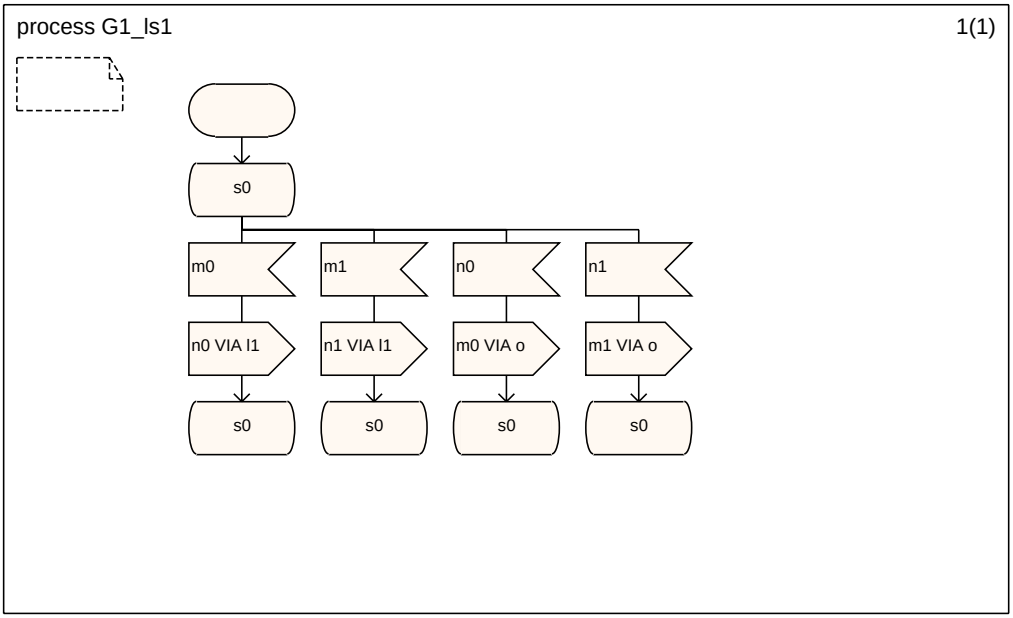

process CoA2
1(1)
s0
n0
n1
n0 VIA ip
n1 VIA ip
s0
s0

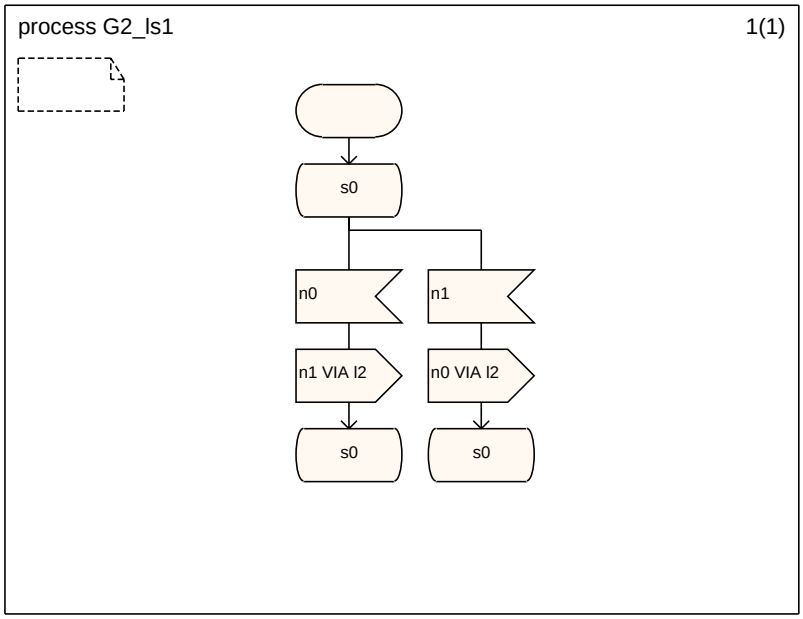

process G2
1(1)
s0
n0
n1
n1 VIA l2
n0 VIA l2
s0
s0

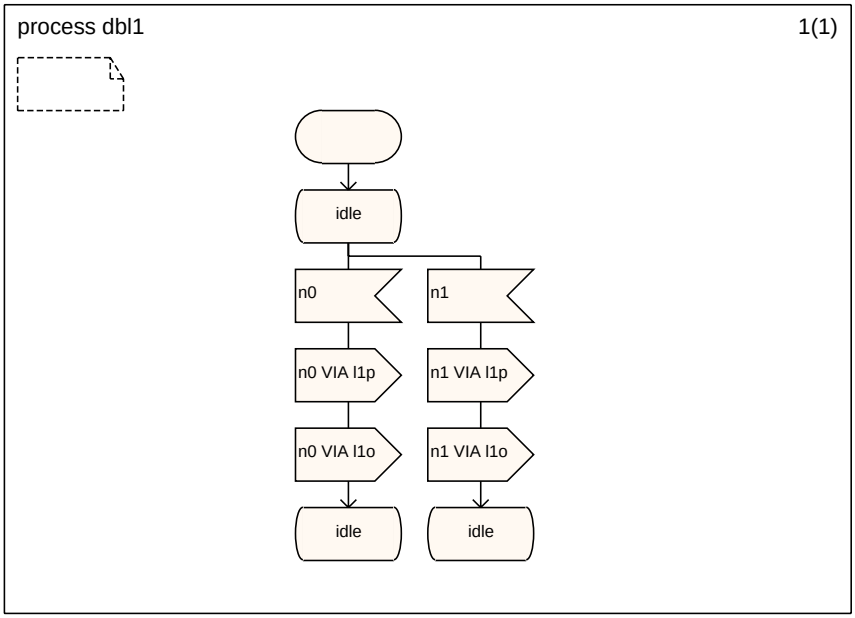

system LS1
1(1)
SIGNAL m0,m1;
SIGNALLIST m=m0,m1;
SIGNAL err;
SIGNAL n0,n1;
SIGNALLIST n=n0,n1;
i
LS1b
o
(m)
(m),err
l1
(n)
l2
(n)

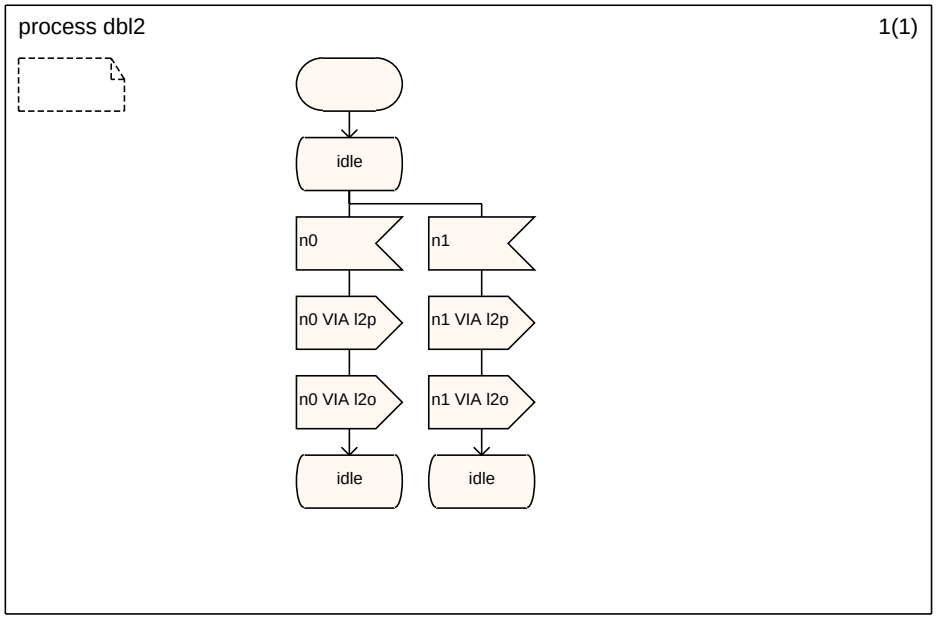

block LS1b
1(1)
op
i
i
A_ls1
(m)
ip
G1_ls1
(m)
o
o
(m),err
(n)
l2p
l1
(n)
dbl1
l1o
l1
(n)
dbl2
l2o
l2
(n)
l1p
(n)
l2
G2_ls1

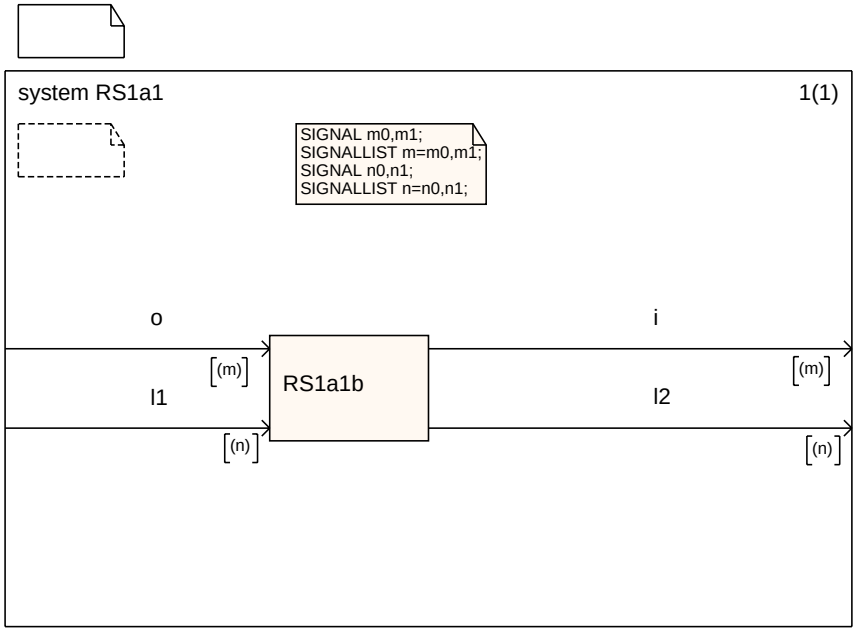

process A_ls1
1(1)
s0
m0
m0 VIA ip
s1
*
err VIA op
fault
s1
m1
m1 VIA ip
s0
*
err VIA op
fault
fault
*
fault

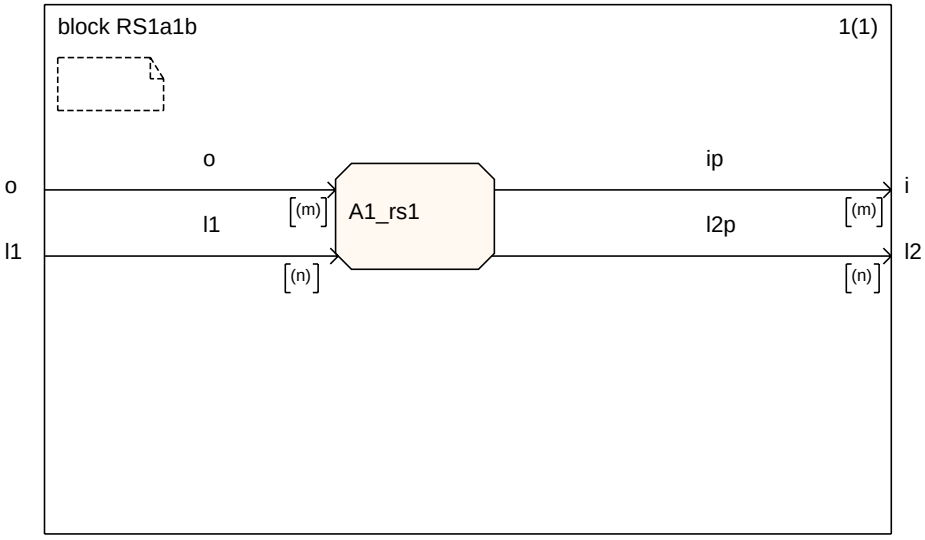

process G1_ls1
1(1)
s0
m0
m1
n0
n1
n0 VIA l1
n1 VIA l1
m0 VIA o
m1 VIA o
s0
s0
s0
s0

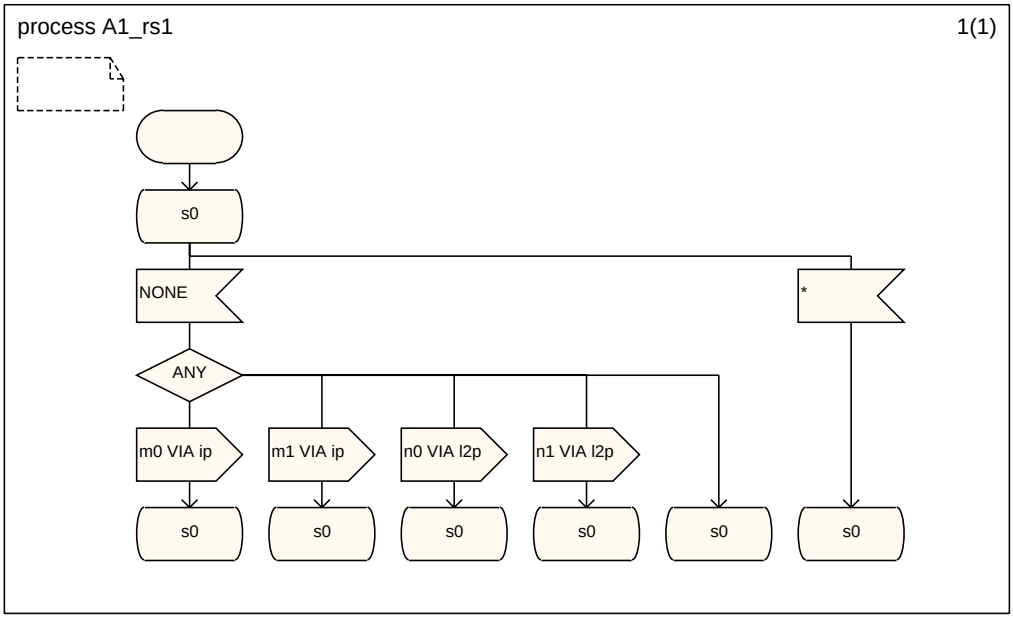

process G2_ls1
1(1)
s0
n0
n1
n1 VIA l2
n0 VIA l2
s0
s0

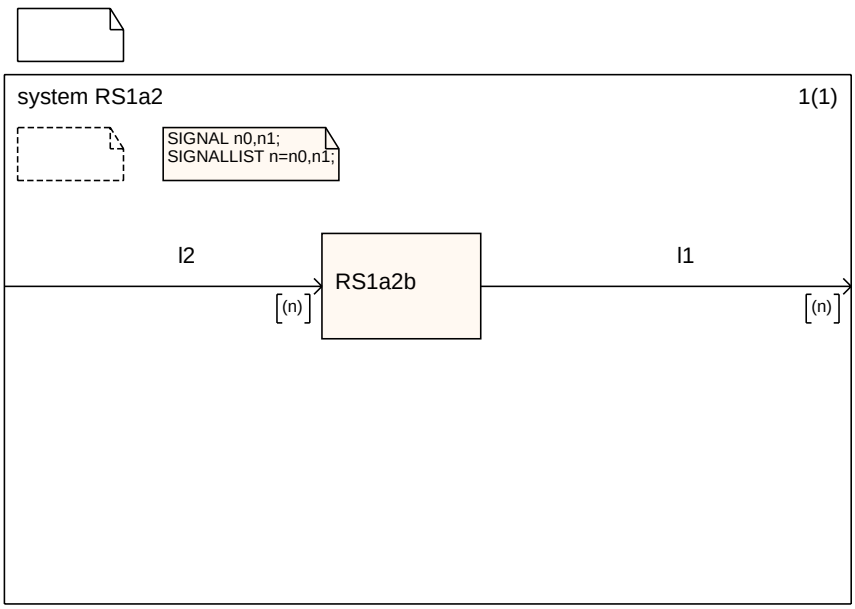

process dbl1
1(1)
idle
n0
n1
n0 VIA l1p
n1 VIA l1p
n0 VIA l1o
n1 VIA l1o
idle
idle

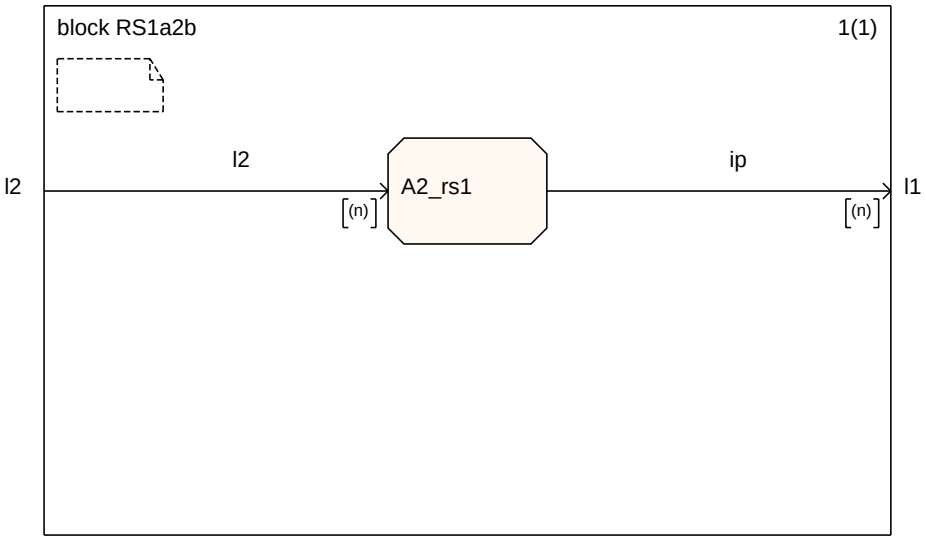

process dbl2
1(1)
idle
n0
n1
n0 VIA l2p
n1 VIA l2p
n0 VIA l2o
n1 VIA l2o
idle
idle

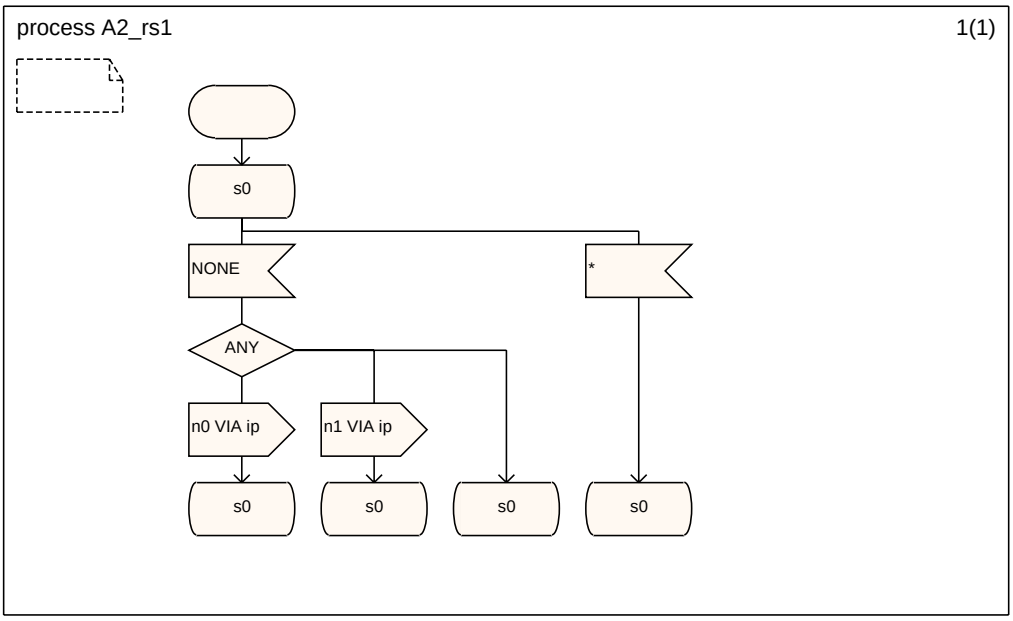

system RS1a1
1(1)
SIGNAL m0,m1;
SIGNALLIST m=m0,m1;
SIGNAL n0,n1;
SIGNALLIST n=n0,n1;
o
i
(m)
(m)
RS1a1b
l1
l2
(n)
(n)

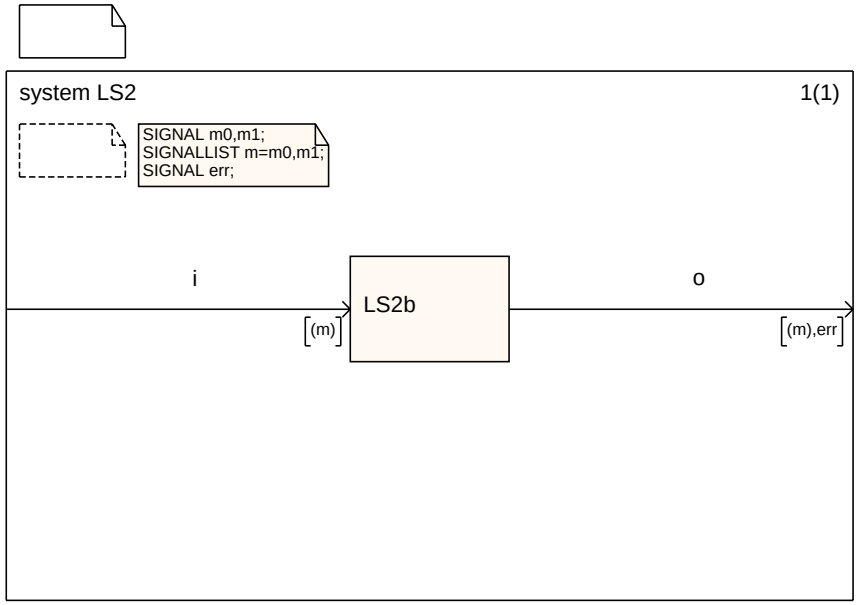

block RS1a1b
1(1)
o
ip
o
i
(m)
(m)
A1_rs1
l1
l2p
l1
l2
(n)
(n)

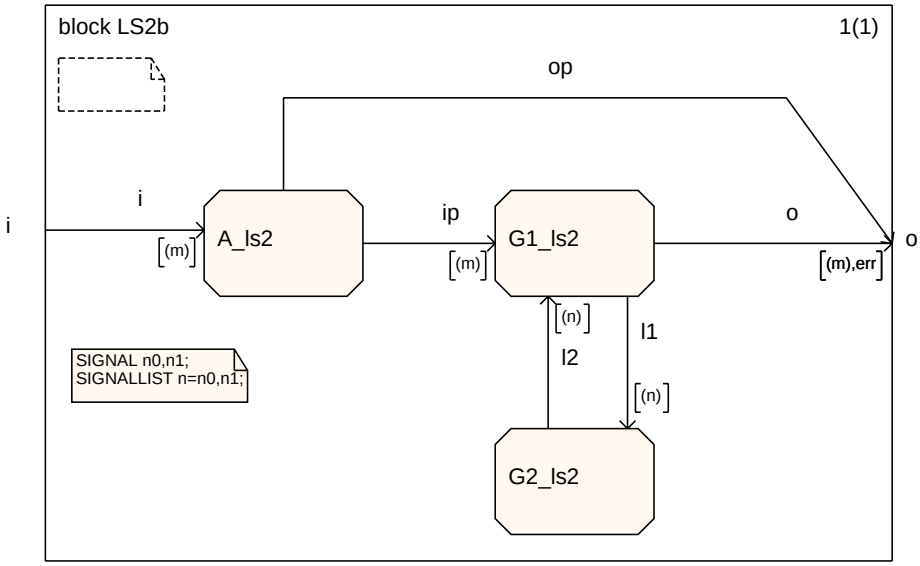

process A1_rs1
1(1)
s0
NONE
*
ANY
m0 VIA ip
m1 VIA ip
n0 VIA l2p
n1 VIA l2p
s0
s0
s0
s0
s0
s0

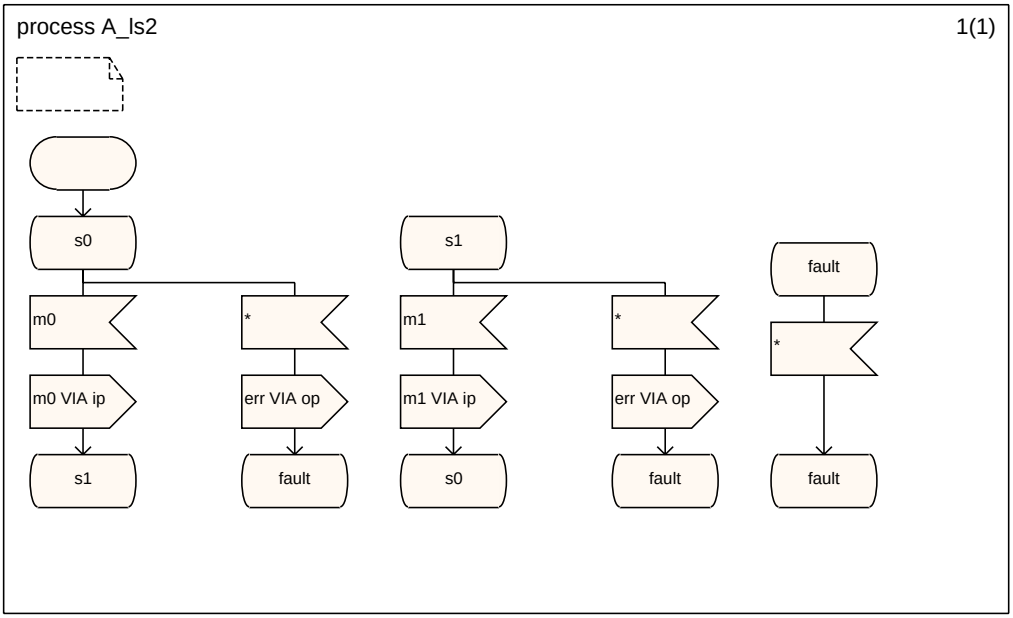

system RS1a2
1(1)
SIGNAL n0,n1;
SIGNALLIST n=n0,n1;
l2
RS1a2b
l1
(n)
(n)

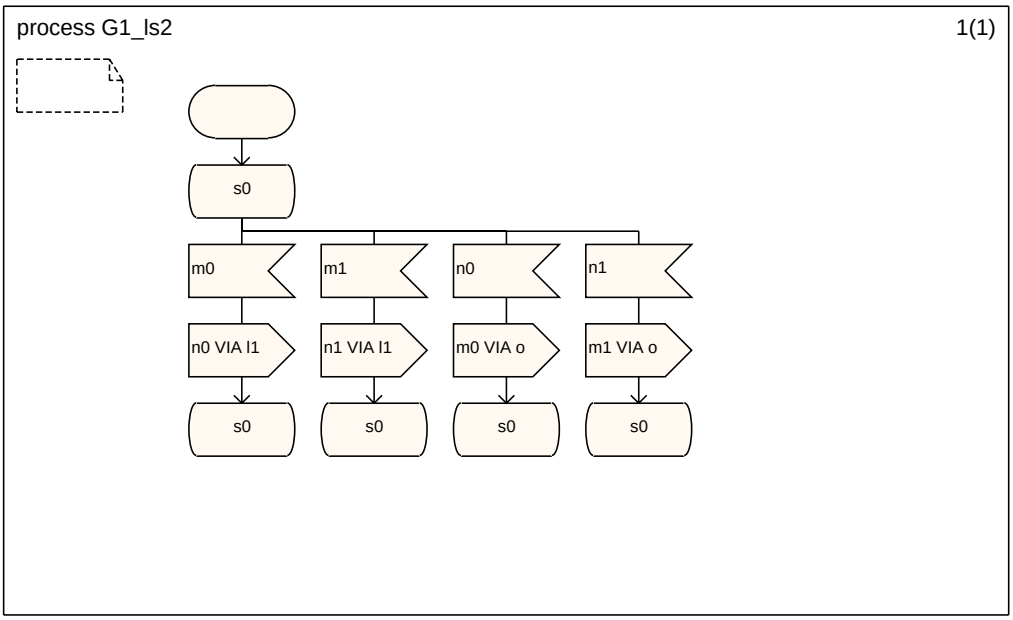

block RS1a2b
1(1)
l2
l2
A2_rs1
ip
l1
(n)
(n)

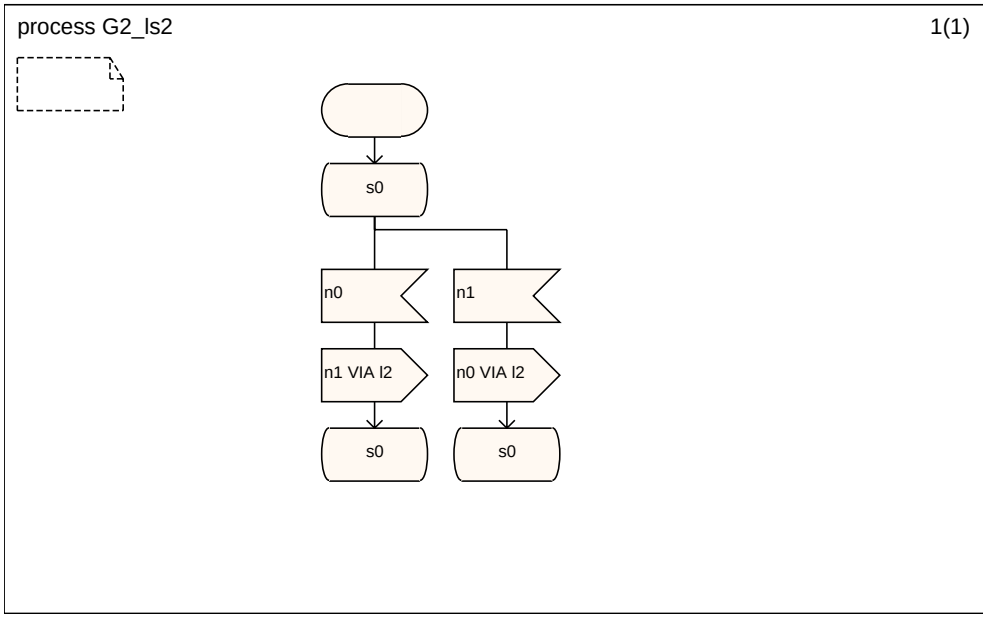

process A2_rs1
1(1)
s0
NONE
*
ANY
n0 VIA ip
n1 VIA ip
s0
s0
s0
s0

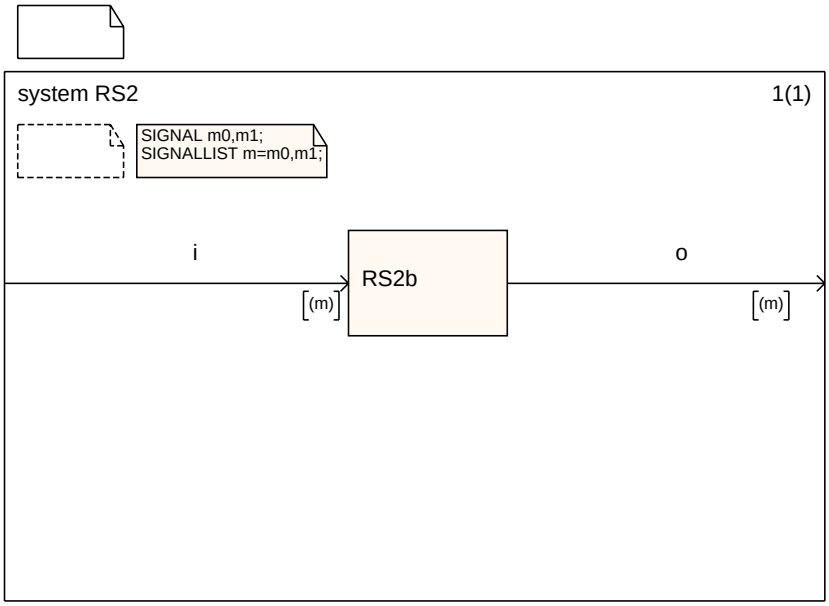

system LS2
1(1)
SIGNAL m0,m1;
SIGNALLIST m=m0,m1;
SIGNAL err;
i
LS2b
o
(m)
(m),err

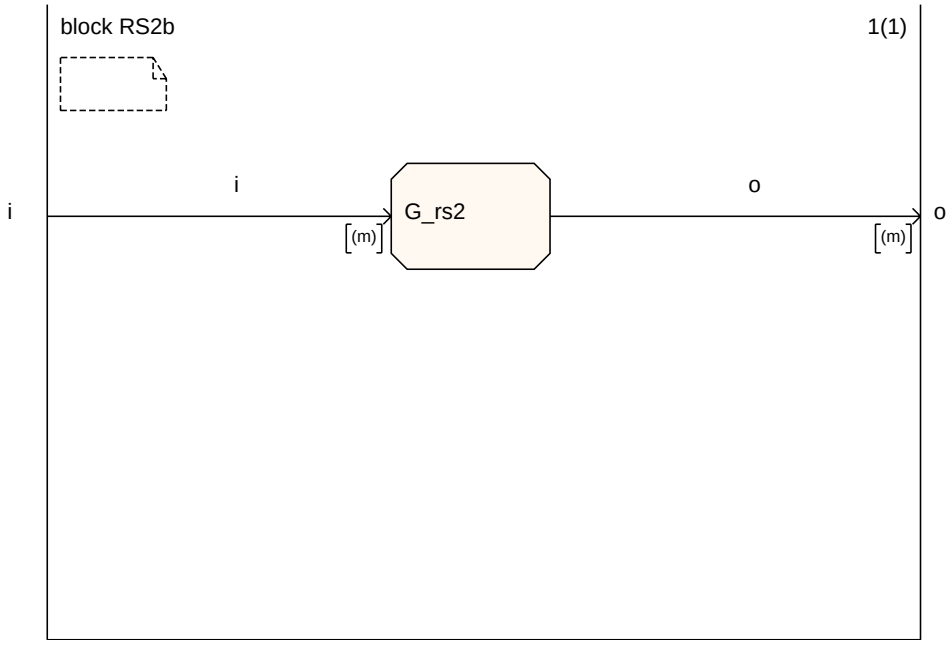

block LS2b
1(1)
op
i
i
A_ls2
ip
G1_ls2
o
o
(m)
(m)
(m),err
(n)
l1
l2
(n)
SIGNAL n0,n1;
SIGNALLIST n=n0,n1;
G2_ls2

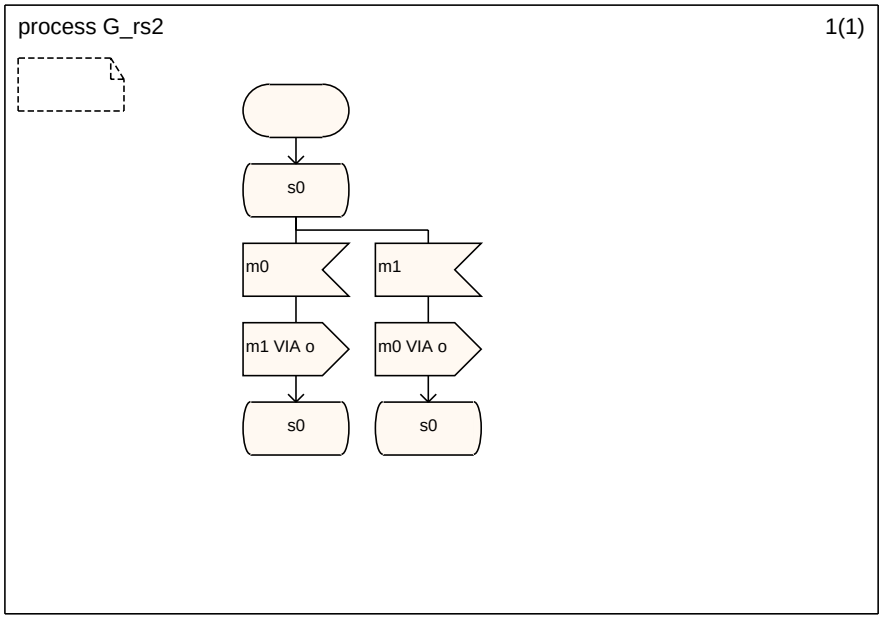

process A_ls2
1(1)
s0
m0
m0 VIA ip
s1
*
err VIA op
fault
s1
m1
m1 VIA ip
s0
*
err VIA op
fault
fault
*
fault


process G1_ls2
1(1)
s0
m0
n0 VIA l1
s0
m1
n1 VIA l1
s0
n0
m0 VIA o
s0
n1
m1 VIA o
s0


process G2_ls2
1(1)
s0
n0
n1
n1 VIA l2
n0 VIA l2
s0
s0


system RS2
1(1)
SIGNAL m0,m1;
SIGNALLIST m=m0,m1;
i
RS2b
o
(m)
(m)


block RS2b
1(1)
i
i
G_rs2
o
o
(m)
(m)


process G_rs2
1(1)
s0
m0
m1
m1 VIA o
m0 VIA o
s0
s0

Appendix C

Test Results

This appendix contains raw results from testing of the tools. Table C.1 provides the number of states in the examples as a measure of size. Tables C.2-C.4 present the results from the tests, and tables C.5 and C.6 show the number of messages in the generated test cases, as a measure of length.

Numer of State

Ex.	**1**	**2**†	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
A	1	2	4	2	1	1	1	2	2	2	1	1	1	2	1
G	1	3	4	2	1	1	1	1	1	4	1	1	1	1	1
A_1	1	2	4	2	1	1	1	1	1	2	1	1	1	2	2
G_1	2	2	4	2	2	1	1	1	1	2	2	2	2	1	1
A_2	1	1	2	1	1	1	1	1	1	1	-	1	1	1	1
G_2	1	2	2	1	2	1	1	1	2	3	1	1	3	2	1
Sum	7	15	20	10	8	6	6	7	8	14	6	7	9	9	7

† Decomposition has three components, but component 2 and 3 are identical.

Table C.1: Size of examples by number of states

		Ex 1	**Ex 2**†	**Ex 3**	**Ex 4**	**Ex 5**
Direct	gen	6:47	6:26	10:00	10:00	0:00
	test	0:38	0:07	1:37	0:08	0:04
	no	10	15	14	10	4
	cov	95,24	70.00	96,08	97,62	100,00
AL 1.1	gen	0:00	0:01	0:05	0:01	0:01
	test	0:02	0:05	0:05	0:04	0:04
	no	2	6	5	2	4
	cov	100.00	100.00	100.00	100.00	100.00
AL 1.2	gen	0:01	0:01	0:04	0:00	0:01
	test	0:04	0:05	0:05	0:05	0:05
	no	1	6	4	3	4
	cov	100.00	100.00	100.00	100.00	100.00
AL 2	gen	0:01	0:01	0:05	0:01	0:01
	test	0:04	0:05	0:05	0:04	0:04
	no	2	6	5	4	4
	cov	100.00	100.00	90.91	100.00	100.00
	D val	Y	Y	Y	N	Y
	AL val	Y	Y	Y	N,1	Y
	group 1	1	1	1	5	2
	group 2	1	1	1	1	3

Table C.2: Results from testing examples 1-5

† Decomposition has three components, but components 2 and 3 are identical.

		Ex 6	**Ex 7**	**Ex 8**	**Ex 9**	**Ex 10**
Direct	gen	0:05	0:04	0:00	0:00	6:22
	test	0:04	0:04	0:04	0:04	0:44
	no	3	3	2	2	10
	cov	83.36	100.00	61.82	61.82	88.73
AL 1.1	gen	0:03	0:03	0:01	0:00	0:00
	test	0:05	0:05	0:04	0:04	0:04
	no	3	3	2	2	2
	cov	100.00	100.00	100.00	100.00	100.00
AL 1.2	gen	0:03	0:03	0:01	0:01	0:01
	test	0:03	0:04	0:04	0:04	0:03
	no	3	3	2	2	1
	cov	100.00	100.00	100.00	100.00	100.00
AL 2	gen	0:04	0:03	0:00	0:00	0:00
	test	0:03	0:04	0:03	0:04	0:03
	no	3	3	2	2	2
	cov	100.00	100.00	100.00	62.50	100.00
	D val	Y	N	Y	N	N
	AL val	Y	N,2	Y	N,2	N,2
	group 1	2	4	3	6	4
	group 2	3	3	2	2	1

Table C.3: Results from testing examples 6-10

		Ex 11†	**Ex 12**‡	**Ex 13**‡	**Ex 14**	**Ex 15**
Direct	gen	0:34	0:18	0:12	0:41	6:50
	test	0:04	0:04	0:03	0:07	0:07
	no	4	1	1	5	10
	cov	100.00	43.75	43,75	96.36	100.00
AL 1.1	gen	0:35	2:01	0:13	0:00	0:00
	test	0:08	0:31	0:06	0:04	0:04
	no	5	5	1	2	2
	cov	100.00	100.00	100.00	100.00	100.00
AL 1.2	gen		2:01	0:12	0:00	0:00
	test		0:13	0:05	0:04	0:04
	no		5	1	2	2
	cov		100.00	100.00	100.00	100.00
AL 2	gen	0:35	0:18	0:12	0:00	0:00
	test	0:05	0:04	0:04	0:04	0:04
	no	4	1	1	2	2
	cov	100.00	45.45	45.45	100.00	100.00
	D val	N	Y	Y	Y	N
	AL val	N,2	Y	Y	Y	N,1
	group 1	4	1	2	1	5
	group 2	3	3	3	1	1

Table C.4: Results from testing examples 11-15

† The assumption of component 2 in the decomposition was not tested because the component has no input channels.
‡ The low symbol coverage is probably because of an endless loop of message passing between the components in the decomposition that dominated input of messages from the environment.

Explanation of Tables C.2-C.4

Direct: testing with Direct testing tool

AL 1.x: testing permiss AL 1.x of the Abadi/Lamport testing tool

AL 2: testing of premiss AL 2 of the Abadi/Lamport testing tool

gen: time used for generating test cases, minutes and seconds

test: time used for executing test cases, minutes and seconds

no: number of test cases (empty test cases not counted)

cov: symbol coverage when executing test cases, percentage

D val: validation with Direct testing tool

- Y validated
- N not validated

AL val: validation with Abadi/Lamport testing tool

- Y validated
- N,1 not validated by premiss AL 1.x
- N,2 not validated by premiss AL 2

group 1: Grouping 1 (section 8.2)

group 2: Grouping 2 (section 8.2)

Number of messages in test cases

Ex 1	Direct	2, 3, 83, 84, 84, 86, 164, 164, 167, 167
	AL 1.1	0, 2, 6
	AL 1.2	0, 0, 2
	AL 2	0, 2, 3
Ex 2	Direct	1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 3
	AL 1.1	0, 4, 4, 4, 4, 5, 5
	AL 1.2	0, 2, 2, 3, 3, 4, 4
	AL 1.3	0, 2, 2, 3, 3, 4, 4
	AL 2	0, 2, 2, 2, 2, 3, 3
Ex 3	Direct	2, 3, 4, 5, 86, 86, 167, 168, 168, 170, 170, 170, 244, 249
	AL 1.1	4, 4, 5, 7, 8
	AL 1.2	0, 2, 2, 3, 3
	AL 2	2, 3, 3, 4, 5
Ex 4	Direct	3, 81, 82, 82, 83, 83, 83, 83, 84, 84
	AL 1.1	0, 0, 0, 0, 1, 3
	AL 1.2	0, 0, 0, 1, 2, 3
	AL 2	0, 0, 1, 1, 2, 4
Ex 5	Direct	2, 2, 4, 4
	AL 1.1	2, 2, 4, 4
	AL 1.2	2, 2, 4, 4
	AL 2	2, 2, 4, 4
Ex 6	Direct	1, 2, 2
	AL 1.1	3, 4, 4
	AL 1.2	2, 2, 3
	AL 2	1, 2, 2
Ex 7	Direct	2, 2, 2
	AL 1.1	4, 4, 4
	AL 1.2	2, 2, 2
	AL 2	2, 2, 2
Ex 8	Direct	2, 2
	AL 1.1	0, 4, 8
	AL 1.2	0, 1, 4
	AL 2	0, 2, 4

Table C.5: Length of test cases by number of messages

Number of messages in test cases

Ex 9	Direct	2, 4
	AL 1.1	0, 2, 4
	AL 1.2	0, 1, 4
	AL 2	0, 2, 4
Ex 10	Direct	9, 84, 84, 85, 85, 165, 165, 166, 168, 168
	AL 1.1	0, 1, 15
	AL 1.2	0, 0, 6
	AL 2	0, 1, 9
Ex 11	Direct	0, 2, 2, 2, 2
	AL 1.1	82, 84, 84, 85, 85
	AL 1.2	82, 82, 82, 83, 83
	AL 2	0, 2, 2, 2, 2
Ex 12	Direct	0, 0, 0, 0, 0, 1
	AL 1.1	69, 99, 198, 199, 199
	AL 1.2	98, 98, 198, 198, 198
	AL 2	0, 0, 0, 0, 0, 1
Ex 13	Direct	20
	AL 1.1	143
	AL 1.2	123
	AL 2	41
Ex 14	Direct	4, 50, 50, 51, 52
	AL 1.1	0, 2, 4
	AL 1.2	0, 1, 4
	AL 2	0, 2, 4
Ex 15	Direct	4, 83, 83, 84 ,85, 164, 164, 165, 165, 167
	AL 1.1	4, 4
	AL 1.2	2, 2
	AL 2	2, 2

Table C.6: Length of test cases by number of messages