ACTA UNIVERSITATIS UPSALIENSIS
Uppsala Dissertations from the Faculty of Science and Technology
86

Jan Henry Nyström

Analysing Fault Tolerance for ERLANG Applications

Dissertation presented at Uppsala University to be publicly examined in Sal 2446, Polacksbacken, Uppsala, Wednesday, June 3, 2009 at 13:00 for the degree of Doctor of Philosophy. The examination will be conducted in English.

Abstract
Nyström, J H. 2009. Analysing Fault Tolerance for ERLANG Applications. Acta Universitatis Upsaliensis. *Uppsala Dissertations from the Faculty of Science and Technology* 86. 178 pp. Uppsala. ISBN 978-91-554-7532-1.

ERLANG is a concurrent functional language, well suited for distributed, highly concurrent and fault tolerant software. An important part of ERLANG is its support for failure recovery. Fault tolerance is provided by organising the processes of an ERLANG application into tree structures. In these structures, parent processes monitor failures of their children and are responsible for their restart. Libraries support the creation of such structures during system initialisation.

A technique to automatically analyse that the process structure of an ERLANG application from the source code is presented. The analysis exposes shortcomings in the fault tolerance properties of the application. First, the process structure is extracted through static analysis of the initialisation code of the application. Thereafter, analysis of the process structure checks two important properties of the fault handling mechanism: 1) that it will recover from any process failure, 2) that it will not hide persistent errors.

The technique has been implemented in a tool, and applied it to several OTP library applications and to a subsystem of a commercial system the AXD 301 ATM switch.

The static analysis of the ERLANG source code is achieved through symbolic evaluation. The evaluation is performed according to an abstraction of ERLANG's actual semantics. The actual semantics is formalised for a nontrivial part of the language and it is proven that the abstraction of the semantics simulates the actual semantics.

Keywords: formal methods, symbolic evaluation, fault tolerance, erlang

Jan Henry Nyström, Division of Computer Systems, Box 337, Uppsala University, SE-751 05 Uppsala, Sweden

ISSN 1104-2516
ISBN 978-91-554-7532-1

urn:nbn:se:uu:diva-101975 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-101975)

Printed in Sweden by Universitetstryckeriet, Uppsala 2009.

Distributor: Uppsala University Library, Box 510, SE-751 20 Uppsala
www.uu.se, acta@ub.uu.se

to my family and friends, they know who they are

Contents

1. Introduction

This introductory chapter presents the background, goals and contributions of this thesis as well as a description of the design decisions made along the way towards achieving the goals. Finally, an overview of the thesis is given.

1.1 Background

The increasingly pervasive dependence on computers in modern society makes it is increasingly important that these systems are dependable. Not only do we wish that the highly critical systems, such as flight control systems and control software for (nuclear) power plants [Rushby 1994], behave dependably but also other systems where great monetary value is at stake.

An example of systems that are not highly critical, yet it is important that they are dependable, are telecommunication systems. Such systems are expected to provide continuous service around the clock, year in and year out. Telecommunication systems, such as telephone switches, are in many cases expected to have less than half a minute of down time in a year [Kuhn 1997], even including time for upgrades and service.

By a dependable system is meant a system that we can depend on to behave in an appropriate manner, both during normal circumstances and when some form of fault has occurred, either in the software or the hardware. The notion of dependability may also include that the system behaves appropriately even under workloads exceeding the largest workload the system can handle, perhaps with downgraded performance.

Four means of achieving dependability are *fault prevention*, *fault tolerance*, *fault removal* and *fault forecasting* [Laprie 1995]. The notion of *fault prevention* is closely connected to general system engineering. *Fault forecasting* deals with the evaluation of the system by predicting the likelihood of a failure and its effect.

Fault tolerance involves three stages, first the system must *detect* that there is an error, whereupon it must *diagnose* the cause of the error in order to perform the correct *error recovery*. The different types of *error recovery* that can be performed are: *backward recovery* where the system is restored to an earlier error free state; *forward recovery* where the erroneous state is transformed into a new error free state, from which the system can continue to operate (though

the continued operation may be degraded in a mode); *compensation* where the erroneous state contains enough redundancy to calculate an error free state.

The purpose of *fault removal* is to minimise the number of remaining errors in the design and realisation of the system, by *verification* of properties derived from the specification, called verification conditions in [Cheheyl et al. 1981]. Should the properties not hold, the reason is *diagnosed* and the system is *corrected*. After the system has been corrected, *regression verification* is performed to ensure that the correction did not break any previously verified properties. The verification may either be a verification in the sense used in the field of formal methods [Rushby 1994, Clarke and Wing 1996], or a validation through code inspection [Mills et al. 1987] and testing [Beizer 1990]. In the case of formal verification, this constitutes proof of partial program correctness that can either be performed on the actual system or a model of the system.

These four methods can be combined to create dependable system as follows. Through adhering to good software engineering principles we ensure *fault prevention*. To further minimise the possible errors in the construction of the system we verify the system against its specification according to *fault removal*. To ensure that the verification conditions, derived from the specification and used in *fault removal* are sufficient, we use *fault forecasting* to determine how likely and damaging errors are. In the firm conviction that faults may yet occur, if nothing else through hardware failure, we use *fault tolerance* to handle the faults occurring during system operation.

One might say that constructing a dependable system is an iterative process, since to ensure that the system is fault tolerant we have to add error detection, diagnosis and recovery to the system. This fault handling subsystem must in turn be fault tolerant. In order to simplify the design of fault tolerant software, as well as perform *fault prevention* and *fault removal*, it is desirable to have libraries supporting the implementation of fault tolerance. Of course, there remains the task of ensuring that the libraries are fault tolerant and that the library facilities are properly used.

ERLANG is a language which has been designed to facilitate the implementation of distributed and fault tolerant software. The language has been used successfully in several commercial applications [Blau et al. 1999, Mullaparthi 2005, Stenman 2006] which place rigorous demands on dependability, in that they must have the high availability normally demanded of telecommunication applications. ERLANG, which is concurrent as well as distributed, supports error detection through an exception construct and the ability to *link* processes. These *links* are bidirectional: should one of the linked processes fail, the other process linked to it will be informed. The diagnosis and recovery from an error is supported by system libraries realising design patterns common in concurrent distributed systems.

To perform *fault removal* on an ERLANG system one can, and indeed should, always use testing and statistical testing [Thévenod-Fosse et al. 1995] to measure the success of the *fault removal*. However, testing the fault tolerance aspects of the system is always tricky and testers have to depend on some form of *fault injection* [Powell et al. 1995]. Should the system be distributed the matters are even worse, since testing in a distributed system [Schütz 1995] is difficult and resource intensive.

A complementary technique to testing the system is the use of formal analysis of the fault tolerance properties of the system, such as whether the system will recover from a failure in one of its processes. When trying to analyse the system it is preferable, on two accounts, to analyse a model of the system as opposed to the source code of the system. The analysis can be clearer since the majority of the system not dealing with the fault handling can be abstracted away, and it helps to keep the analysis tractable. For large systems it is impracticable and error prone to construct a model by hand. On the other hand is it hard to automatically extract a model of something as intricate as a large distributed system. In the case of ERLANG, there is a natural solution to these problems: the model of the libraries supporting fault handling are manually constructed, and the model extraction of the remainder of the system is performed automatically. The effort of modelling the supporting libraries by hand, amortised over all the different systems analysed, is quite acceptable.

1.2 Problem

The main problem we address in this thesis is:

> Given a specific implementation of an ERLANG application, automatically determine the fault tolerance properties of that application.

To succeed with this task we are confronted with a number of questions:

- Which fault model should the analysis consider in the context of ERLANG applications, i.e, what type of faults should it be able to handle, and what should handling entail?
- Which properties concerning fault tolerance in ERLANG applications are of interest? An example property is: if there is a persistent fault, how many times does the system try to recover before failing.
- Which fault tolerance properties of an ERLANG application can be automatically determined? The assumptions we make about the system structure and design will greatly influence what properties we can determine. It is therefore vital that it is clearly stated what assumptions are made, and to motivate the choice in view of the alternative sets of assumptions that could be made.

- What is the appropriate formal model to use when reasoning about ERLANG applications? The ERLANG application executes in a concurrent and possibly distributed environment. The application can itself consist of a number of processes which can be spread over several ERLANG-nodes. In order to be able to reason formally about an ERLANG application we must have a formal model of its environment.
  We have to decide what type of formal semantics to use as a model of ERLANG and its environment. The choice of semantics will determine what precision we can attain, and how hard it will be to reason about the correctness of our analysis.
- What analysis techniques should be used to determine the fault tolerance properties of ERLANG applications? The possible techniques have different properties with respect to the computational resources they require, as well as their difference in strengths when it comes to showing properties.
- How should the fault tolerance properties of an ERLANG application best be presented? In order for the results of the analyses to be of use they have to be presented in a clear way, so that a user of the analysis results can understand why the application has this property. It likewise important to present the places where the analysis can not determine what the properties are, so that even partial results have a potential use.

In this thesis I present techniques to extract the fault tolerance model of an ERLANG system automatically, as well as manually constructed model of the supporting libraries. I will further formulate relevant fault handling properties and present an automated analysis of the fault handling models, with example applications to commercial distributed dependable systems.

We will continue by presenting an outline of the areas we will touch on in order to answer these questions. First we will look at fault tolerance, explaining the basic concepts and establishing the terminology used in this thesis. Then the ERLANG language with supported libraries and design principles are positioned within the field of fault tolerance. The chapter ends with a detailed listing of the contributions made by the author in this thesis.

1.3 Fault Tolerance

The aspect of dependability examined in this thesis is the behaviour of the system when some fault has occurred, known as fault tolerance. What is the appropriate response to a fault differs between systems and what type of fault has actually occurred. Let us begin by looking at some examples of the different approaches used to provide fault tolerance:

- The simplest form of fault treatment is to simply stop the system, but few industrial or embedded systems can allow the system to stop.
- In order to allow a *graceful degradation* [Herlihy and Wing 1991], one may let unaffected parts of the system continue once the affected

parts are stopped. To actually determine what other parts of a system are affected by a fault in one component or subsystem is a lively research topic in its own right, see [Sampath et al. 1995]. This can be helped by grouping functionality into components together with error detection mechanisms forming *self-checking component* [Yau and Cheung 1975, Laprie et al. 1990], which enables clearly defined *error containment areas* [Siewiorik and Swartz 1982].

- In some systems one can not allow the system to continue after a fault, but the system, or what it controls, must be left in a safe state. This is called *fail-safe* fault tolerance [Gärtner 1999]. An example of such a system is the Ariane 5 ground control system [Dega 1996], which could handle the failure of one component but stopped in a safe state when two components failed. The term *fail-safe* is also used differently in [Laprie et al. 1990], where it denotes a system where only benign faults may occur.
- The Ariane 5 ground control system, also exhibits an example of *redundancy*, where there are two or more components to perform the same task. When one of the components suffers a fault it is simply stopped and it is up to the remaining components to perform the task. Ideally the fault should not be detectable by an outside observer.
  The redundant parts may be different physical objects, such as chips or computers, or software entities, such as databases or sub-systems. The redundant parts may even be the design or algorithm implemented, where if a fault is detected one changes to a different implementation, possibly even an earlier version of the same part. This change may impair the performance of the system, and we have yet again a case of *graceful degradation*. The use of redundant components can also be used to achieve detection of errors where all the components perform the assigned task and their results are compared for discrepancies (using comparators and voters). Another form of organisation of redundant components is the method of *fail-stop processors* [Schlichting and Schneider 1983] where components that detect internal faults simply stop in a way detectable from the outside, i.e., by the other fail-stop processors and the higher system levels. One commonly used example of this use of redundancy is N-version programming [Aviziensis 1985].
- An interesting notion is that of self-stabilising [Schneider 1993a, Dijkstra 1974] systems which recover from any perturbations of their internal state although the error persists until the correct internal state has been restored. This would mean that the system can handle any transient fault. It is however hard to construct or indeed verify such a system [Theel and Gärtner 1998]. One implementation of a self-stabilising algorithm in a commercial system is the SunSCALR framework for fault tolerance and load balancing, as described by [Singhai et al. 1998].

I will proceed to define some of the basic terminology and the fault model we will assume together with the computational model.

The most basic notion is that of a fault pertaining to the lowest form of abstraction, e.g., a memory cell that always returns the value 0 [Jalote 1994] or a division by zero. The fault may lead to an error which is an incorrect internal state of the system, which in turn may lead to a failure. A failure is the state where the system fails to fulfil its specification.

We will in this thesis use slightly different terminology, fault has the usual meaning but an error is when the system, or any of its components, fails to fulfil its specification. A failure occurs when a process terminates abnormally, and when the root process of a supervision structure fails we will say that the supervision structure fails. The reason for this choice of terminology is that it conforms to the terminology prevalent in the ERLANG community.

Fault tolerance contains two necessary parts, error *detection* and *recovery*, even if it is a moot point whether the decision just to fail constitutes error recovery. There are three distinct forms of recovery [Laprie 1995]:

Backward recovery where the system is restored to an earlier error free state, which may necessitate that check point states are saved. This recovery method has the advantage that we know that, if the detection works correctly, there exists an error free previous state and we need not diagnose the error. The disadvantage is that we may lose information added since the last check point.

Forward recovery where the erroneous state is transformed into a new error free state, this does not require check points but it is likely that diagnosis of the error is necessary is to determine how to perform the state transformation. Removing one of the redundant components in N-version programming is an example of forward recovery.

Compensation where the erroneous state contains enough redundancy to calculate an error free state. There is one application of compensation that actually deviates from the normal detection and recovery scenario, *fault masking* where the recovery action is performed in each step without any detection. In systems where state is complex the cost of *fault masking* would probably be prohibitive.

It is easy to assume that the three forms of recovery are mutually exclusive, but in fact they can easily be combined in any fashion. One can imagine a system that will rely on *compensation* if there is enough redundancy to restore the state, then if that fails resorts to *forward recovery* unless the diagnosis fails, in which case it performs *backward recovery*.

1.3.1 Fault Models

There is a number of hierarchical taxonomies for fault models in which techniques for fault handling are organised according to the end result of the system [Schneider 1993b, Cristian 1991], common ones are:

crash fault model where the entire system simply stops;

fail-stop fault model which behaves in the same manner as *crash* fault model but the stopping of a subsystem is visible to the other parts of the system;

Byzantine fault model where the subsystem may behave in any way, including malevolently.

In [Gärtner 1999] is proposed an alternative way of classifying fault tolerance into four different forms, where the discriminating factor is which type of properties the system preserves. The categories of properties follow [Lamport 1977] those of safety and liveness. Informally, a safety property states that nothing bad will happen, and a liveness property states that something good will happen eventually.

The four forms of fault tolerance proposed by [Gärtner 1999], obtained by dividing them depending on what type of properties they preserve after a fault, are presented in Table 1.1. The *masking* fault tolerance, where we have progress in the system and the system never violates its specification, is the desired form. The *masking* fault handling, which is totally transparent to an external observer, is of course the hardest to achieve. The easiest form is of course *none* where no type of properties are preserved and is not so much fault tolerant as fault intolerant. The two intermediate forms are of more interest; *fail safe* fault tolerance was mentioned already in Section 1.1, and is quite common in safety critical systems where breaking the specification is much more serious than stopping altogether.

	live	not live
safe	masking	fail safe
not safe	non-masking	none

Table 1.1: *Gärtner's four forms of fault tolerance*

The final form of fault tolerance, *non-masking*, where the system, after violating the safety properties during fault handling, returns to normal execution is for us the most interesting, although it is claimed in [Gärtner 1999] that:

> For a long time nonmasking fault tolerance has been the "ugly duckling" in the field, as application scenarios for this type of fault tolerance are not readily visible (some are given by [Arora et al. 1996] and by [Singhai et al. 1998]).

I would like to differ on this point, there is a large class of soft real time systems performing a large number of similar tasks, where it is more important that the vast majority are handled correctly and in time, but where a few tasks may fail completely. An example of such an area is telephone switching

where one talks about the telecommunication four nines, i.e., 99.99% availability of the systems. These systems must be up and running continuously, and must serve the vast majority of calls correctly, but the call in ten million that fails, may do so completely, as long as it does not disturb the handling of the 9'999'999 other calls. Another area for which this holds most of the time is internet services, where we may be annoyed upon failing to load a page, but the unavailability of the server may prove disastrous to the provider.

1.3.2 Model of Distributed Computation

To be able to reason about fault tolerance properties of a concurrent, or indeed distributed, system one has to specify the computational model used. The computational model will have direct impact on the level of error detection that is possible.

Some of the aspects are: network topology, atomicity of actions, communication primitives available and how the individual processes are modelled. One important aspect is their notion of real time, i.e., if there exist any real time bounds on execution, message transmission and process response time.

If no assumptions are made concerning real time bounds the models are called *asynchronous* [Schneider 1993b, Lamport and Lynch 1990]. The *asynchronous* model is the weakest, i.e., that every system can be modelled as *asynchronous* or as [Schneider 1993b] also calls them: *non-assumption*. The result is that if you can show something using an *asynchronous* model it will hold under any other consistent assumptions. Its has been argued that the *asynchronous* model should also be good practise [Chandra and Toueg 1996], since for many real systems it is extremely difficult to state any assumptions concerning real time behaviour. There is however a price for using this model, it is not possible to detect a crashed process in an *asynchronous* model [Chandy and Misra 1986].

There are of course models between the *synchronous*, where all bounds are known, and the *asynchronous* where no bounds are known, these models are known as *partially synchronous* [Lynch 1996, Dolev et al. 1987].

1.3.3 Erlang's Computational and Fault Model

In the case of ERLANG, fault tolerance when a failure is reached is clearly *non-masking* and can be categorised as the *fail-stop* fault model. This is hardly surprising, as ERLANG was developed to implement telecommunication systems, and as I have argued, *non-masking* fault tolerance is suited to these systems.

The form of error recovery, or rather failure recovery, advocated by the ERLANG documentation is a form of *backward recovery*. The reset to an earlier state is performed by shutting down all the affected processes and restarting them, resetting them to the initial state with the exception of data stored in persistent storage such as the distributed database Mnesia [Mattsson et al. 1999].

When it comes to computational model, ERLANG has no fixed topology and the only objects for which atomicity is implemented are the messages which are received whole and in the same order as sent when originating from the same sender. The communication primitives consist of asynchronous point-to-point message passing.

The issue of synchrony is less easily decided, the obvious option is the *asynchronous* model, but that is however inappropriate since the ERLANG way of fault tolerance relies on the ability to determine that a process has crashed. The underlying ERLANG runtime system relies on a heart-beat mechanism to detect that another node can no longer be reached, so the appropriate synchrony is the *partially synchronous* model where there is a definite upper bound on the time before a process is notified that a linked process has failed.

1.4 ERLANG

ERLANG is a concurrent functional language, especially tailored for distributed and fault tolerant software, e.g., in telecommunication applications [Armstrong et al. 1996, Armstrong 2007].

Prominent features of ERLANG include support for light-weight processes, asynchronous message passing, and fault handling as integral parts of the language. The Open Telecom Platform (OTP) [Torstendahl 1997] provides a number of libraries which support program design patterns that commonly occur in concurrent distributed software. Examples of such patterns, called "behaviours" in OTP, are event handlers, generic servers, and finite state machines.

The ERLANG language supports implementation of failure recovery by a mechanism in which *links* can be created between processes. When a process fails, all processes linked to it are notified, and can react either by failing themselves (thereby informing other linked processes), or by initiating a recovery action such as restarting a copy of the failed process.

The *supervisor behaviour* in OTP is used to program processes which monitor a set of children. Using links, a supervisor is notified about failures of its children, and can then restart new copies of failed children, possibly after some cleanup operations. The "best practice" when designing ERLANG systems is to organise all processes of the ERLANG system into *supervision structures*, i.e., trees of processes in which parent processes supervise their children [Armstrong 2003, Armstrong 2007].

The failure recovery mechanisms in ERLANG and OTP can be seen as a way to catch exceptions caused by some anomalous condition in a process: this makes it possible to write clear code for each process, which is not obscured by defensive code. When using such a style of programming, it is important to ascertain that the global process structure of the system is set up in such a way that it recovers from arbitrary process failures. This can be done by ex-

tracting the process structure, and thereafter inspecting it to analyse the effect of any particular process failure, saying which processes will be affected and determining whether the process structure will be restored after recovery.

Currently, to obtain the process structure of an application, one must rely on external documentation or manual inspection of the source code. However, it is nontrivial to extract the process structure from source code since an ERLANG program is structured according to modules and functions, whereas process creation and communication may occur anywhere in the code, and since the created process structure is not unique; it may depend on the system environment and configuration parameters.

1.5 In this thesis

In this thesis, we present a technique for automatically detecting deficiencies in the failure recovery mechanism of ERLANG applications which are due to improperly designed supervision structures. The technique is structured into two phases.

- The set of possible process structures is extracted by static analysis of the source code. More precisely, we extract an overapproximation of the set of possible static parts of process structures by symbolically executing the initialisation code of the application. By "the static part" we mean the processes started when the application is started and which are to remain running (possibly restarted to handle failures) until the application terminates. The extraction assumes that the OTP libraries are used in the recommended way to set up the process structure; otherwise the precision will be poor.
- Each extracted process tree is analysed to determine the effect of process failures. We present a technique to determine the effect of a particular process failure on the entire process structure, which shows which processes will be terminated and restarted and whether the structure itself is restored to the situation before the failure.

In addition to providing sufficient information for analysis, the extracted process structure is also of independent interest; it can be presented to the designer for visual inspection, with the possibility to choose different views depending on the information sought. As an example, the parts of the application that are affected by an abnormal process termination can be visualised.

I have implemented the techniques in this thesis in a tool, which extracts sets of possible process static structures from source code, and which automatically checks the effects of a process failure in each process structure. The tool can also check that principles for the construction of "good" supervision structures are followed. If the principles are not followed strictly, the tool can check the effects of process failures; this is useful when analysing legacy code applications, which may not have been designed using current design

principles. We have applied the tool to several OTP-library applications and a subsystem of the AXD 301 ATM switch [Blau et al. 1999].

1.6 Contributions

The main contributions of this thesis are the following:

- A detailed presentation of fault tolerance methods advocated by the OTP documentation to be employed in ERLANG applications.
- A formulation of the key properties, regarding fault tolerance, of ERLANG applications, together with techniques for automated assessment of these properties.
- An automated method to extract the static part of the supervision structure from the source code of an ERLANG application, using symbolic execution.
- An operational semantics for the sequential part of CORE ERLANG, based on the semantics of [Carlsson 2001].
- A tool implementing the extraction and analysis of ERLANG applications.
- Application of the tool to non-trivial commercial applications.

1.7 Related work

We have not found any report of work that performs the same type of analysis as ours. This could partly be due to the fact that there are few other languages where failure recovery on the process level is supported by language and library primitives in the same way as in ERLANG.

Analogous to our extraction of process structures is the extraction of call-graphs of a program in inter-procedural program analysis (e.g., by Agrawal in [Agrawal 2000]). There is a relationship, since an argument to a process creation statement in ERLANG should be the function executed initially by the created function. A complication in ERLANG is that this function may be computed in arbitrary ways, making it hard to obtain precision in the analysis.

Our aim is also to some extent related to the area of fault analysis, where one is also interested in the potential effects of faults in a system component (e.g., Sampath et al. in [Sampath et al. 1995]). However, most work in this area assumes that suitable models of system components are given (e.g., as state machines), and does not address the extraction of these models from source code.

1.7.1 Model extraction

The extraction of analysable models (e.g., finite state machines) of concurrent system from source code has recently attracted much attention in the model checking community. The aim is to extract control skeletons from source code.

C

Holzmann extracts Promela models from C with threads in the tools FeaVer [Holzmann and Smith 2000] and AX [Holzmann 2000], This method relies on user defined abstractions, i.e. deciding what procedures and variables are unimportant, and consequently if the user definitions are incorrect executions that violates the properties under investigation might not be found.

Java and Concurrent ML

An approach similar to Holzmann's, which does not rely on user definitions, is applied by Corbett for JAVA [Gosling et al. 1996] in the Bandera project [Corbett 2000] where shape analysis is used to determine what variables are only accessible from one thread. A method for the derivation of a finite-state control skeleton from Concurrent ML [Reppy 1993] programmes, abstracting values to their types, is presented in [Nielson et al. 1998].

ERLANG

Arts and Earle has investigated translation of ERLANG programs into μCRL [Blom et al. 2001] models which can be model checked by the CÆSAR/ALDÉBARAN [Fernandez et al. 2000] tool set. Properties to proved are specified in alternation free modal μ-calculus and checked against state spaces generated by the μCRL tool set [Wouters 2001]. A problem of translating ERLANG into μCRL is that ERLANG requires process fairness, whereas the parallel composition of μCRL lacks this notion. The fairness assumptions of ERLANG must be explicitly stated in the correctness properties. In [Arts and Earle 2001] they investigate a simplified version of a resource locking mechanism in the AXD 301 ATM switch [Blau et al. 1999]. This has been continued in [Arts et al. 2004a, Arts et al. 2004b].

In [Leucker and Noll 2001] Leucker and Noll has continued this work and has implemented a prototype distributed model checker and proposes to employ the ELAN [Borovanský et al. 1998] environment for specifying and prototyping deduction systems as the next step. When using ELAN, they would have to specify the syntax and semantics of ERLANG in terms of abstract data types and term rewriting rules similar to that of [Noll 2001]. The use of rewriting rules enables them to to develop higher level specifications in the same way as in [Arts and Noll 2001]. The ELAN work is presented in [Noll 2003, Amiranashvili 2002]. This work with rewriting logics has been continued using the Maude system [Clavel et al. 2003] in [Neuhäußer and Noll 2007].

1.7.2 Analysis of ERLANG

The analysis of ERLANG programs has received increased attention in the last years. This can be seen in the steadily increasing number of papers on the subject.

Static Type Analysis

The first attempts made to analyse ERLANG was to derive type information. It has been the approach with the largest practical impact with the Dialyzer tool [Lindahl and Sagonas 2004] included in the OTP release.

In [Lindgren 1996] is developed a soft-typing system for ERLANG, where types are not declared but derived from the code. The constraint solver used was however deemed to be unsuitable for ERLANG. That was followed by a paper [Marlow and Wadler 1997] describing another soft-typing scheme, in which however not all legal ERLANG programs could be typed.

A third soft-typing scheme was presented in [Nyström 2003], where the type inference is based on data flow with optional user annotations, the intention being that the user would annotate all the interfaces, and warnings generated for all possible type clashes. The main drawback of this system is that it tends to generate too many false positives.

A slightly different approach is followed in [Lindahl and Sagonas 2004] where a data flow analysis is applied to the virtual machine byte code to derive the possible values of the live variables at each program point. This type information enables the tool to find: places that would raise exceptions, unreachable branches, and dead code. The work has be extended in [Lindahl and Sagonas 2006] to generate success typings. The success typings, unlike the other soft-typing schemata already described, allows for compositional bottom up type inference which has been shown to scale well in practice.

To provide the necessary information to provide optimisations using a new heap structure and allocation strategy, a variant of escape analysis is presented in [Carlsson et al. 2006].

Abstract Interpretation

Huch has developed an approach for the formal verification of ERLANG programmes using abstract interpretation and model checking. In [Huch 1999] an ERLANG system is viewed as a set of expression evaluations in the context of the identity of the processes executing the expressions and their message queues. The abstraction consists of truncating the terms in the expression at a predefined depth. It is mentioned in the paper how one could tailor the interpretation so that for selected terms the terms are either kept as they are or truncated at a greater depth. The language used to specify the desired properties is a linear temporal logic [Manna and Pnueli 1992], expressive enough to state most interesting properties.

The interpretation can only handle tail recursive programs and does not handle exceptions, links, nor process termination. This work has been extended in [Huch 2001] where he handles non tail recursive calls, through a technique of jumps which makes his approach much more realistic for real life programs. The work has been further extended in [Huch 2003], where a technique to automatically generate an abstracted finite graph representation of the possible

evaluations of a process. This graph is used to model check properties of the program.

A factor that hinders this method, in its current cast, from handling realistic applications is that the model checking does not scale to: dozen of processes and over 50'000 lines of source code of the AXD 301 subsystem analysed in Chapter 7.

Theorem Proving

The ERLANG Verification Tool(EVT) [Fredlund et al. 2003] is an interactive proof assistant with an embedding of the language in the proof rules. The specification logic used [Fredlund 2001] is a first order logic inspired by the modal μ-calculus [Kozen 1983] and extended with ERLANG-specific features. The logic is quite powerful, with both least and greatest fix points, allowing the formalisation of a wide range of behavioural properties. The strength of the specification language however leaves the verification problem undecidable, although normally a considerable parts of a proof can be automatically produced.

There have been extensions to allow the tool to reason about ERLANG code on an architectural level, where the specified behaviour of the OTP behaviours can be characterised by sets of transition rules [Arts and Noll 2001]. This enables the tool to reason about OTP behaviours without having to consider their concrete implementation.

There are two great differences between this work and my thesis. First that the proof of properties, using the proof assistant, are very much done by hand even if most of the tedious details are automated, whereas an analysis by my tool will performed automatically. When performing an analysis with my tool one may have to state an initial configuration, such as contents of database tables, but this constitutes only a small effort. Secondly, with the proof checker one can prove a wide range of properties, whereas if I were to examine anything else but the fault tolerance aspects of ERLANG, I would have to redesign the model extraction and the subsequent analysis.

Conditional Term Rewriting Systems

In [Giesl and Arts 2001] it is shown how techniques for analysing termination of conditional term rewriting systems, can be used to show properties for ERLANG programmes. The correctness properties of the query lookup protocol of the Mnesia distributed database is transformed into a termination problem, and then it is shown that, using refinement of dependency pairs techniques [Arts and Giesl 1997], the transformed properties can be proved without manual intervention. The same properties have also been shown to hold using EVT [Arts and Dam 1999].

Model Checking

Wiklander has in [Wiklander 1999] implemented a translation from a finite state subset of ERLANG to Promela, the specification language of the model checker SPIN [Holzmann 1991, Holzmann 1997]. The main difficulties were to translate the dynamic data structures lists and tuples to Promela constructs. Without a good translation of these basic data structures of ERLANG, it is hard to translate any program. Another major obstacle was, finding an effective encoding of ERLANG's receive statement using Promela's different style of message passing.

Another process algebraic model is presented in [Noll and Roy 2005] and then refined in [Roy et al. 2006]. In these two papers a subset of ERLANG is translated into asynchronous π-calculus with monadic communication. The model can be checked using existing tools for π-calculus. The advantage one gets from using π-calculus as opposed to μCRL used in [Arts et al. 2004a] is that name passing feature of π-calculus allows the direct representation of the sending of process ids between processes. The models deal with a subset of the language and do not handle distribution or concurrency at a level that is even near what is required to analyse real systems.

A distinctly new approach is examined in [Fredlund and Earle 2006], where model checking is achieved by running the program being checked in lock step with an automaton representing a safety property on a new run time system for ERLANG. The result is a system where everything is ERLANG. The system was only a prototype with the semantic model not fully completed, lacking elements of the semantics presented in [Claessen and Svensson 2005]. This has now been extended to support to the entire semantics, including the modeling of distributed ERLANG as described in [Fredlund and Svensson 2007]. It is the approach that has most similarities with the techniques used in this thesis.

1.7.3 Semantics

Formal semantics for various subsets of ERLANG have been presented, as well as rigorous but informal semantics, for the whole language.

A detailed and rigorous account of ERLANG language version 4.7.3 is presented in [Barklund and Virding 1999], where syntax, semantics and assumptions underlying the implementation are explained. There was an earlier suggestion for the development [Barklund 1999] with discussions regarding the existing semantics of the language.

The Formal Design Techniques group at the Swedish Institute of Computer Science has defined a succession of formal semantics for subsets of ERLANG starting with [Dam et al. 1998a, Dam et al. 1998b]. The latest installment of these semantics is presented in [Fredlund 2001], which is the most complete formal characterisation of ERLANG today. The semantics is a a layered small step operational semantics[Plotkin 1981]. The subset of ERLANG contains most of the language concepts except the notions of modules, distribution and

real time. Some parts of the runtime system integral to the language, such as the possibility to register symbolic names for process identities, are also missing. This semantics is more comprehensive than mine in that it handles the concurrency aspects of ERLANG which I do not formally state. On the other hand I handle modules, all value types and guard functions. The runtime system features, such as registering, that I formulate informally are not handled by Fredlund.

The semantics presented in [Fredlund 2001] has been further expanded in [Claessen and Svensson 2005], with a layer dealing with the distribution. The intention is that this will form a basis for future model checkers. This is already the case where the McErlang, described in [Fredlund and Earle 2006, Fredlund and Svensson 2007], base its distributed communication on this semantics.

Huch has defined a somewhat different approach for defining semantics in [Huch 1999], where he relies heavily on contextual information. The result of this is that he can only compare processes when they are parts of a closed system. The subset is also much smaller then Fredlund's.

There is also a definition of the semantics of the sequential fragment of CORE ERLANG [Carlsson 2001].

The analysis of [Carlsson et al. 2006] is based on a big-step operational semantics that only deals with one process at the time. The state is modelled as a set of shared terms.

1.8 Overview

The thesis is organised as follows:

Chapter 1: Introduction
: The introductory chapter presents the background and a formulation of the questions this thesis tries to address. The chapter then goes on to briefly present ERLANG and the contributions of this thesis, and finally a presentation of the related work is given.

Chapter 2: ERLANG
: This chapter presents the background of the language, as well as all the parts of the language relevant to this thesis. A presentation of the OTP behaviours and their intended use for supervision is also given.

Chapter 3: Analysis of Failure Recovery Through Supervision Structures
: This chapter presents and motivates the properties to be determined by the analysis and describes how the analysis is performed on the supervision structure of an ERLANG application.

Chapter 4: Semantics
: This chapter presents CORE ERLANG, an intermediate language used by the OTP ERLANG compiler, and then goes on to give a semantics for CORE ERLANG.

Chapter 5: Abstract Semantics
: This chapter presents an abstraction of the semantics presented in the previous chapter. This semantics constitutes the basis for the extraction of the supervision structure of an ERLANG application.

Chapter 6: Process Structure Extraction
: This chapter presents the symbolic execution of CORE ERLANG which is used to extract the supervision structure of an ERLANG application.

Chapter 7: Experiments
: This chapter presents the results of a number of analyses of OTP library and industrial applications, then a more detailed description of one of the analyses is given.

Chapter 8: Thesis Summary
: This chapter summarises the thesis and presents conclusions drawn from the thesis work and what directions future work can take.

1.9 Publications by the Author

Of the following publications it is only paper 9. that is directly connected to this thesis.

1. A Formalization of Service Independent Building Blocks, J. Nyström, B. Jonsson, *Proceedings of the Advanced Intelligent Networks'96 workshop, ed. T. Margaria*, 1996.
2. Creation of Dependent Features, J. Blom, R. Bol, B. Jonsson, J. Nyström, *Proceedings of RadioVetenskap och Kommunikation'96*, 1996.
3. A Case Study in Automated Detection of Service Interactions, G. Naeser, J. Nyström, B. Jonsson, *Proceedings of RadioVetenskap och Kommunikation'99*, 1999.
4. Building Tools for Creation and Analysis of Telephone Services, G. Naeser, J. Nyström, B. Jonsson, *Proceedings of RadioVetenskap och Kommunikation'99*, 1999.
5. On Modelling Feature Interactions in Telecommunications, B. Jonsson, T. Margaria, G. Naeser, J. Nyström, B. Steffen, *Proceedings of the Nordic Workshop on Programming Theory, eds. B. Victor and W. Yi*, 1999.
6. Incremental Requirement Specification for Evolving Systems, B. Jonsson, T. Margaria, G. Naeser, J. Nyström, B. Steffen, *Feature Interactions in*

Telecommunications and Software Systems VI, eds. M. Calder and E. Magill, ISO Press, 2000.
7. A formalisation of the ITU-T Intelligent Network standard, J. Nyström, *Licentiate thesis, Department of Information Technology, Uppsala University, Sweden*, 2000.
8. Incremental Requirement Specification for Evolving Systems, B. Jonsson, T. Margaria, G. Naeser, J. Nyström, B. Steffen, *Nordic Journal of Computing, Vol.8, No.1*, 2001.
9. Extracting the processes structure of Erlang applications, J. Nyström, B. Jonsson, *Proceedings of the Erlang Workshop in connection with Principles, Logics, and Implementations of high-level programming languages(PLI'01)*, 2001.
10. Evaluating Distributed Functional Languages for Telecommunications Software, J.H. Nyström, P.W. Trinder, D.J. King, *Proceedings of the* 2^{nd} *ACM SIGPLAN Erlang Workshop*, 2003.
11. Are High-level Languages suitable for Robust Telecoms Software?, J.H. Nyström, P.W. Trinder, D.J. King, *Proceedings of the* 24^{th} *International Conference on Computer Safety, Reliability and Security (SAFECOMP'05), Lecture Notes in Computer Science Vol.3688*, 2005.
12. Evaluating High-Level Distributed Language Constructs, J.H. Nyström, P.W. Trinder, D.J. King, *Proceedings of the* 12^{th} *ACM International Conference on Functional Programming (ICFP)*, 2007.
13. Priority Messaging made Easy, J.H. Nyström, *Proceedings of the* 6^{th} *ACM SIGPLAN Erlang Workshop*, 2007.
14. High-level Distribution for the Rapid Production of Robust Telecoms Software: comparing C++ and Erlang, J.H. Nyström, P.W. Trinder, D.J. King, *Concurrency and Computation: Practice & Experience. Vol.20, Issue 8*, 2008.

2. ERLANG

In this chapter the ERLANG language is described, together with the OTP libraries defining behaviours. The chapter begins with a section on the intentions and background of the language, and thereafter describes the language starting with its sequential parts and ending with its support for distributed systems. The OTP libraries are described one by one, as well as their intended use in system design.

2.1 History and Usage

The ERLANG language is primarily intended for the implementation of telecommunication software, this demanding reliable system with soft real time properties that can be upgraded while continuing to operate.

2.1.1 History

ERLANG stems from research, within Ericsson, on which language was best suited for telecommunication applications. The design of ERLANG started in the late 80's with an interpreter written in PROLOG [Armstrong et al. 1992a], that was first presented to the world in 1990 [Armstrong and Virding 1990][1]. Originally the language was a very simple concurrent functional language, with only those language features deemed necessary for the implementation of telecommunication applications. Some constructs normally associated with functional languages, such as functional values (closures), were not present in the early implementations.

The initial trials within Ericsson were successful [Ödling 1993], but indicated that the language needed to be much more efficient. In order to make the language more efficient and independent of PROLOG, a new implementation based on an interpreter written in C [Kernighan and Ritchie 1978] for an abstract machine JAM (Joe's Asbtract Machine) [Armstrong et al. 1992b]. The JAM was inspired by WAM [Warren 1983, Aït-Kaci 1990], an abstract machine for PROLOG [Sterling and Shapiro 1986]. The JAM, in comparison to WAM, had added primitives for concurrency and exception handling.

Before the release of the commercial version 4.1 in October 1993, distribution was added as an integral part of language [Wikström 1994].

[1] For a more detailed history of the language see [Däcker 2000, Armstrong 1997].

The development of ERLANG and its supporting libraries continued, with among other things a distributed real time database named Amnesia [Nilsson and Wikström 1996], which was renamed as Mnesia [Mattsson et al. 1999] when it became a product.

With the version 4.4, ERLANG matured a lot and functions were now first class objects, which could be passed as arguments to functions as well as returned from functions. A syntax for anonymous functions (or lambda expressions as they are often called) had been introduced. Other additions included records, list comprehensions, macros and the possibility to include files. The records are only syntactic sugar, which are expanded by a preprocessor stage.

In May 1996 the first prototype version of the Open telecom platform (OTP) [Torstendahl 1997] was delivered. The OTP system, was designed to support an open system approach of implementing telecommunication software, where different technologies, computers, languages, management systems etc. could cooperate. The OTP system consists of ERLANG with libraries for distributed application management, release handling, OS resource monitoring, alarm handling, the distributed database Mnesia, operation and maintenance interface and HTTP support. An important part is the libraries supporting common design patterns in telecommunication applications, called behaviours in OTP terminology.

The work to improve the language has not ceased. In 1992 Hausman started to work on an implementation that compiled directly to C instead of being interpreted [Hausman 1994], based on the abstract machine BEAM (Bogdan's Abstract Machine),[2] that replaced JAM in 1999. A multi threaded version of ERLANG [Hedqvist 1996] was implemented in 1998, and that year the OTP went Open Source enabling anyone to try new techniques in the context of ERLANG. This has now be turned into fully fledged SMP support [Lundin 2008].

Among the later additions to ERLANG is a syntax for manipulating binaries [Nyblom 2000], making it easy to implement protocol stacks and other low level bit handling software. This has now been extended to handling bit streams and binary list comprehensions [Gustafsson 2007].

2.1.2 Usage

The showpiece of ERLANG/OTP applications is the control logic of the AXD 301 [Blau et al. 1999] scalable backbone ATM switch, an application consisting of over 500'000 lines of ERLANG code. The system also contains approximately 280'000 lines of C and small amounts of JAVA code, Wiger [Wiger 2001] claims that the use of ERLANG instead of C/C++ resulted in a fourfold increase in productivity without any loss in quality. These findings has been born out by later studies [Nyström et al. 2008].

[2] For a survey of different abstract machines see [Diehl et al. 2000].

Other successful experiences reported are: the first implementation of GPRS (General Packet Radio Service), a high speed packet data service for GSM networks, demonstrated at CeBit 1998 [Granbohm and Wiklund 1999]; and the Intelligent Network Service Creation Environment by [Hinde 2000]. Among later successes can be noted the T-mobile's sms platform the Third Party Gateway [Mullaparthi 2005].

ERLANG/OTP has now established itself in many fields outside telecommunications as is evident from early adoptions[3] such as the Mail Robustifier [Millroth 1999]. More recently ERLANG has been used in: video on demand and interactive TV [Gonzáles 2007], ad serving platform for online games [Ippolito 2008], financial services [Stenman 2006], highly scalable distributed data stores [Schütt et al. 2008]

2.1.3 Alternative ERLANG Implementations

There have been several alternative implementations of ERLANG, the most notable is High Performance Erlang (HiPE) [Johansson et al. 2000], which native compiles ERLANG to both the SPARC [SPARC International 1994] and the x86 [Intel 2002] platforms. The HiPE system uses the OTP ERLANG runtime system and you can freely combine native compiled code with the byte compiled code. HiPE has been a part of OTP since the 8th OTP release.

An implementation of ERLANG compiled to C, wholly independent of OTP, is Gerl [Wong 1998]. The rational for this implementation was to be able to instrument the implementation with automatic monitoring of the system for the benefit of the developer. Another implementation is ETOS [Feely and Larosse 1998, Feely et al. 1999], which compiles ERLANG to Scheme [Dybvig 1996].

There has been work on changing ERLANG so that it would provide a secure execution environment for mobile code [Naeser 1997, Brown and Sahlin 1999]. The reason that they have focused on ERLANG is that the language supports writing of reliable systems, which is of great importance for secure mobile code, since many security attacks exploit weaknesses in fault handling [Dean et al. 1996].

2.2 Basic Language

A brief characterisation of the sequential part of ERLANG is: strict, call by value with static binding and dynamic typing [Peyton Jones 1987, Allen 1978]. What this means in more detail will be presented below.

[3] More examples earlier ERLANG/OTP applications can be found in [Däcker 2000, Armstrong 1996].

2.2.1 Data Types

The ERLANG language is dynamically typed, meaning that variables and parameters are not declared to have a specific type, instead the type is determined dynamically and type errors will be detected during runtime. An example of a type error is to call the built in function plus + with the arguments 'a' and 'b' which are atoms. Type errors detected during runtime generate exceptions.

There have been attempts to design a soft type system for ERLANG [Lindgren 1996, Marlow and Wadler 1997]. In a soft type system type declarations are optional, and where the type is not declared the type system tries to infer the type. Unlike in a statically typed language, the code will compile even if the type system can not determine the types for all variables and expressions, although if a type error is determined the compilation generates an error.

The generic name for an ERLANG value of any type is *term*, and parts of a term are called *subterms*.

2.2.1.1 Atomic Data Types

The atomic data types, that are not composed of other data types, are: numbers, atoms, process identifiers, ports, references, and the empty list.

Numbers consist of integers or floats. OTP ERLANG may use several computer words to represent large integers (so called bignums) and consequently can represent ridiculously large integers. The floats have the usual notation. The usual arithmetic operations are available on numbers as well as conversions between integers and floats.

Atoms are named constants. They are written as any alphanumeric sequence starting with a lower case letter and terminated by a non alphanumeric character. An atom may start with an upper case letter or contain non alphanumeric characters if it is enclosed in single quotes. Example atoms are: `erlang`, `first`, `listSofar` or `'$trangeAtom'`. The basic operations on atoms are comparisons and transformation to and from strings.

Process Identifiers can not be manipulated; they can only be converted to a printing format or compared. They are normally abbreviated *pids*, as in the built in function `is_pid/1` which tests whether its argument is of the type process identifier. The pids are created when a new process is spawned.

Ports or port identifiers are identifiers of external processes with which the ERLANG node can communicate. Port identifiers are similar to pids.

References are unique symbolic constants. References are created using the built in function `make_ref/0`, which does not take any arguments and returns a new unique reference on each call. References can only be compared, or transformed into a string for output.

2.2.1.2 Compound Data Types

Compound data types are made up of parts that may in their turn be either atomic or compound. ERLANG has the following compound data types: tuples, lists, functions, and binaries.

Tuples are of form $\{t_1,\ldots,t_n\}$ where the arity of the tuple is n and $t_1,\ldots,t_n$ are the subterms of the tuple. The subterms of a tuple may be of any type, including tuple. A tuple has a fixed arity, and there are functions which determine the arity and select subterms of the tuple. Tuples are used much as records are used in other languages.

Lists are used to store sequences of varying length. A list $[t_1,\ldots,t_n]$ of length n, composed of the subterms t_1 to t_n, can also be written as $[t_1,\ldots,t_i \mid [t_{i+1},\ldots,t_n]]$ where the part after the | is called the tail of the list. Normally one distinguishes between the first element of the list and the rest of the list. A list, like a tuple, may contain data of any type as subterms. It is correct in ERLANG to construct a "cons" cell (i.e., a head and tail pair) where the tail is not a list, e.g., $[1 \mid bar]$, but this is not a proper list, and trying to use the built in function `length` to calculate its length will result in an exception. In other words, the result of cons is not always a proper list. There are built in functions to select elements from a list and a system library `lists` with a wide variety of different utility functions on lists.

Strings in ERLANG is not a proper data type on its own, but is implemented as lists of integers. They however have their own syntax of form `"` *String* `"`, where *String* is any sequence of alphanumeric characters and escape sequences, e.g., `\n` for a newline.

Functions as a data type (or funs as they are usually called) were added to ERLANG in version 4.4 and have the form

$$\begin{array}{l}\texttt{fun(}p_{11}\texttt{, ..., }p_{n1}\texttt{) -> }b_1\texttt{;}\\ \qquad\vdots\\ \texttt{(}p_{11}\texttt{, ..., }p_{n1}\texttt{) -> }b_n\\ \texttt{end}\end{array}$$

where the p_{ij}:s are patterns as described below in Section 2.2.2 and the b_j:s are any ERLANG expressions. The fun's clauses can also have guards as described below in Section 2.2.4. Funs can be tested if they are of the type fun, compared and applied to arguments. An example fun is the identity function `fun(X) -> X end`. ERLANG has static binding, which means that any variable in the body of a fun must either be a formal parameter, bound to a value within the fun or already be bound in the context in which the fun is constructed.

Binaries are the latest addition among the compound data types and were introduced in ERLANG version 5.0. A binary is a sequence of octets, of form «$t_1,\ldots,t_n$». Each of the segments $t_1,\ldots,t_n$ are of form *value* `:` *size* `/` *type* −

sign − *endian* − `unit` : *no*, where all parts except *value* is optional. The *type* which is either integer, float or binary shows what type the value has. The *size* multiplied with the *no* gives how many bits the segment will occupy, though the sum of all segment lengths has to be a multiple of eight. The part *endian*, which can either be `little` or `big`, decides if the most significant octet comes first or last. Finally the *sign*, which is either `signed` or `unsigned`, shows if the segment is signed.

2.2.2 Matching

An important part of the language is matching, which is used to compare and select parts of compound data structures and to determine which clause of a function, case, or receive expression is applicable. An explicit match is of form *pattern* = *expression*, where *pattern* is a constant term with possible occurrences of variables, this construct can be used to construct, e.g., `Cons = [Head |Tail]`. A variable is any alphanumeric sequence started by an upper case letter or '_' and terminated by any non alphanumeric character. A special variable '_' is used for parts of a pattern whose value is unimportant; the anonymous variable '_' always remains unbound.

In a match the variables in the pattern will be bound to the corresponding parts of the term resulting from the evaluation of the expression matched against. For example, in `{example, X} = {example, 1}` the variable `X` will be bound to 1. Should the term not match the pattern, an exception will be raised. ERLANG allows nonlinear patterns, i.e., patterns where one variable occurs more than once. In a nonlinear pattern all occurrences of a variable must be bound to the same value. For example, the matching `{X, X} = {1, 1}` will succeed whereas `{X, X} = {1, 2}` results in an exception. This is because the second occurence is considered already bound and it is treated exactly as variables bound before the match, i.e., as if it was the bound value itself.

From ERLANG version 4.8 and, on matchings inside patterns are allowed (or an *alias* as this special case of matching is also known), so $p_1 = p_2$ is a pattern if both p_1 and p_2 are patterns. In order for an alias to match a term, both patterns must match the term, e.g., in `{X = {1, Y}, Z} = {{1, 2}, 3}`, the variable `X` is bound to `{1, 2}`, and `Y` and `Z` to 2 and 3, respectively.

2.2.3 Modules

ERLANG functions are grouped into modules, where each module has a private name space for functions. An example module is presented in Figure 2.1, where two simple functions are defined. The module starts with the declaration of the name of the module, which in this case is `example`.

Functions that are supposed to be called by other functions outside the defining module must be declared exported. Export declarations of functions

are on the form `-export(` *FunctionList* `)`, where *FunctionList* is a list of *FunctionName/Arity* pairs. One must state the arity of a function when declaring exported functions, since in ERLANG two different functions can have the same name if they have different arity, e.g., `reverse` in the `example` module.

```
-module(example).
-author('JanHenryNystrom@gmail.com').
-export([reverse/1,
         sort_reverse/1,
         lazy_reverse/1
        ]).

reverse(List) -> reverse(List, []).

reverse([], Reversed) ->
    Reversed;
reverse([H | T], Reversed) ->
    reverse(T, [H | Reversed]).

sort_reverse(List) ->
    Sorted  = lists:sort(List),
    reverse(Sorted, []).

lazy_reverse(List) when length(List) < 10 ->
    reverse(List);
lazy_reverse(List) ->
    List.
```

Figure 2.1: A example ERLANG module.

When calling a function defined in another module, the module in which the function is defined must be prepended to the function name, e.g., the `sort` function defined in the system library module `lists` is called as: `lists:sort(List)`.

A module can also contain definitions of macros, records. These are only defined in the module where they occur. Function definitions are described in the next section.

2.2.4 Functions

A function definition consists of a sequence of function clauses separated by semicolons and terminated by a period. Each function clause is of form *FunctionName* `(` $p_{1k}, \ldots, p_{nk}$ `)` `when` *Guard* `->` b_k, where *FunctionName* is the

name of the function, the p_{ik}:s are patterns and the b_k:s is any ERLANG expression. The guard part `when` *Guard* of the clause is optional. When the function is called, the first clause for which the formal parameter patterns match the actual arguments, and the guard expression evaluates to `true`, is evaluated. As an example, a call to `reverse([a, b], [])` would result in the second clause of `reverse/2` being evaluated with the variables `H`, `T`, `Reversed` bound to `a`, `b` and `[]` respectively. If there is no clause for which the actual arguments match the parameter patterns and the guard evaluates to `true`, then an exception is raised.

ERLANG is call by value, which means that when a function call is evaluated the arguments are evaluated to constant terms before the function is applied. The order in which the arguments are evaluated is not defined. For example in the call `f(q(r(X)), s(X))`, the calls to `q`, `r` and `s` would be evaluated before the call to `f`, but it is not known whether `q` and `r` are evaluated before `s` or not. If all the functions in the previous example are side effect free, the result will not depend on the order of evaluation of the arguments.

The guards in the clauses consist of the keyword `when` and a comma separated sequence of simple tests, all of which have to evaluate to `true` in order for the guard to evaluate to `true`. Simple tests are built in functions such as type tests and simple arithmetic operations. Should any of the functions in the test cause an exception, it is caught and the guard evaluates to false, e.g., when evaluating the call `lazy_reverse(horse)` the guard in the first clause would evaluate to `false` since the built in function `length/1` would return an exception caused by the fact that the atom `horse` is not a list.

In ERLANG version 4.9 disjunctions in guards were added. In Figure 2.2 is shown first an example of a disjunction in a guard and thereafter an equivalent way of writing the same function without disjunctions.

```
%With disjunction
f(X) when atom(X); number(X); list(X) -> p(X).

%Without disjunction
f(X) when atom(X) -> p(X);
f(X) when number(X) -> p(X);
f(X) when list(X) -> p(X).
```

Figure 2.2: Example of disjunctions in guards.

The guard expressions were further extended with boolean expressions in version 4.9, and non-strict versions of `and` and `or`, named `andalso` and `orelese` respectively, were added in version 5.1. The non-strict versions do not evaluate the second argument if the result can be decided after the evaluation of the first argument, e.g., if the first argument to `andalso` evaluates to

`false` then the call `andalso` can not evaluate to `true` and consequently evaluates to `false` without evaluating the second argument.

The body of a function clause consists of a comma separated sequence of expressions which are evaluated sequentially in order, and the returned value of the function is the value of the last evaluated expression.

2.2.5 Conditionals

The two language constructs used for conditional evaluation is the `case` and the `if`, where the `if` is really a special case of the `case` construct.

The `case` is on the form: `case` *Expression* `of` *ClauseSequence* `end`, where each case clause in the *ClauseSequence* is of form *Pattern* `when` *Guard* `->` *ClauseBody*. The "`when` *Guard*" part is, as in function definitions, optional, and the clauses are separated by semicolons. An example `case` statement is shown in Figure 2.3, together with the equivalent `if` statement.

```
case f(X) of
  Y when list(Y) -> true;
  _  -> false
end
```

```
Y = f(X),
if
  list(Y) -> true;
  true -> false
end
```

Figure 2.3: Case and If statements.

The `if` statement is just a sequence of clauses with only guard tests, without the `when` keywords, preceded by the `if` keyword and terminated by the `end` keyword.

As with function definitions the *ClauseBody* is a comma separated sequence of expressions that are evaluated in order, with the result of the statement being the result of the last expression evaluated. Should no clause be selected in the conditional, an exception will be raised.

2.2.6 Variable Scopes

A variable is defined from the point in the function where it was bound and to the end of the function. A variable can only be bound by a matching operation, either explicit matching, parameter pattern or clause pattern. Since ERLANG is a single assignment language, once the variable is bound it retains that value throughout the function definition.

For a variable bound in a conditional statement to be well defined in the following statements in the function, it has to be bound in all clauses of the conditional statement. In figure Figure 2.4 an the variable Z is unsafe and will generate a compiler error.

```
case f(X) of
  [Y]    -> Z = true, used;
  [a, b] -> not_used;
  _      -> Z = false, used
end,
q(Z)
```

Figure 2.4: Example of a case with an unsafe variable.

It is a deprecated usage to bind a variable in all clauses of a conditional and later in the function use that variable, if is preferable to return the value from the last expression of the conditional clauses instead. In Figure 2.5 a deprecated use is shown on the left and the preferred usage on the right.

```
case f(X) of                        Z = case f(X) of
  [Y] -> Z = true, not_used;              [Y] -> true;
  _   -> Z = false, not_used              _   -> false
end,                                    end,
q(Z)                                q(Z)
```

Figure 2.5: A deprecated use of variable bindings and the correct alternative.

2.2.7 List Comprehensions

List comprehensions which were added in ERLANG version 4.4, are a succinct and elegant way of performing filtering and mapping on lists. The general format of the comprehensions are: `[` *Expression* `||` *QualifierSequence* `]`, where the *Expression* can be any expression and the *QualifierSequence* consists of a comma separated sequence of generators and filters. The generators are on the form *Pattern* `<-` *Expression* where the *Expression* must evaluate to a list. The filters are predicates (functions that evaluate to a boolean) or boolean expressions. The result of a list comprehension is the list of all possible values of *Expression* given the bindings generated by the generators for which all the filters are `true`. An example list comprehensions is `[{X, Y} || X <- [1,2,3], Y <- [a,b]]` which calculates the Cartesian product of the two lists resulting in `[{1,a},{1,b},{2,a},{2,b},{3,a},{3,b}]`.

For list comprehensions there a few special scoping rules:

- All variables occurring in the pattern of a generator are "fresh"', i.e., the previous definition of the variable will be shadowed by the one in the generator pattern.

- Variables defined previously to the list comprehension may be used anywhere except in the generator patterns, they will of course have the already defined value.
- No bindings made in the list comprehension are visible outside the comprehension, e.g, the `X` and `Y` variables used in the calculation of the Cartesian product above will not be available outside the list comprehension.

2.2.8 Exceptions

There are two constructs in the ERLANG language for the handling of exceptions, the `throw/1` built in function and the `catch` statement. Exceptions can be raised either by the ERLANG runtime system caused by, for example type errors, or as a result of incorrect arguments to a built in function, or using the `throw/1` built in function. An exception alters the flow of control and breaks out of the current evaluation until it reaches a `catch` statement or reaches the process' initial function call whereupon the process fails.

The value of the exception is the argument to the built in function `throw/1` or if it is a runtime error of the form `{'EXIT',` *Reason* `}`, where the *Reason* would typically be that a match failed or that a built in has been called with the wrong type argument. For example, evaluating `a = 1` would result in the exception `{{badmatch, 1},`*History*`}` being raised, where *History* is a truncated list of all function calls that have been made previously in the evaluation.

A `catch` statement is on the form: `catch` *Expression*, where the result is either the value of the exception if one is raised during the evaluation of the *Expression*, or otherwise the value of the *Expression*. A typical usage of the catch construct is shown in Figure 2.6, where the call to the function `dangerous` may result in an exception being raised, in which case we report the error and perform some clean up, or a `success` which is returned.

```
case catch dangerous(X) of
  {'EXIT', Reason} ->
    report_error(Reason),
    clean_up();
  success -> success
end
```

Figure 2.6: Example of the catch construct.

2.3 Concurrency, Distribution and Fault Detection

The concurrency in ERLANG is explicit whereas the distribution is semitransparent. Processes are created and handled explicitly, but the distribution on different nodes has to be explicitly handled only when creating nodes, processes and connecting nodes. The concurrent process operations, such as the sending of messages, is handled in the same manner irrespective on which nodes the involved processes reside, but it is possible to determine from the pid on which node a particular process resides.

2.3.1 Communication

Communication in ERLANG is asynchronous via message passing. The non-blocking function `!`, called as "`Pid ! Message`", transmits the message `Message` (which may be any ERLANG term) to the mailbox of the process with process identifier `Pid`. The receiver's mailbox preserves the order of messages sent from the same process.

A process accesses its mailbox through the `receive` statement, which matches a number of clauses against the messages in the mailbox queue. The first message in the queue which matches some clause will be received. If there is no such message, the receive statement blocks; blocking may be avoided by an optional timeout clause with a specified waiting time.

The `receive` statement is of form: `receive` p_1 `when` $guard_1$ `->` $body_1$ `;` $\ldots p_n$ `when` $guard_n$ `->` $body_n$ `after` *timeout* `->` $body_{n+1}$, where the guard is optional as with function and conditional clauses. The *timeout* part of the receive statement may be omitted which has the same effect as giving `infinity` as the timeout. Namely that the process will wait until it receives a message matching one of the clause patterns, however long that may take.

2.3.2 Process Handling

A process is created by calling a function in the `spawn` family of functions, with arguments specifying what function and with what arguments the created process should execute. Optional arguments determine on which ERLANG node the process should be created. The `spawn` functions return the process identifier (pid) of the created process.

A process can terminate normally by returning from the last function call, or abnormally by not catching an exception. We use the term *failure* to denote abnormal termination. A process can force another process to terminate normally or abnormally by calling the built in function `exit(Pid, Reason)`, where `Pid` is the pid of the terminated process.

There is also a mechanism for giving names to processes, which can be used instead of pids. This mechanism is called registration, and is performed by a call to the built in function `register/2` with the name as the first argument

and the pid as the second. Only atoms can be used as registered names of a process, and a process can only have one registered name. The name can be used instead of the pid in most cases within the ERLANG node, e.g., sending a message to a process registered as `X_server` can be made through the expression `X_server!`*`message`*.

2.3.3 Failure Detection

The basis for failure handling is that *links* can be created between processes. A process creates a link by a call to the built in function `link/1` with the pid of the other process as an argument. Alternatively, the `spawn_link` functions behave in the same manner as the `spawn` functions, but also link the parent and child processes when creating the child. The link between processes can be destroyed by one of the connected processes by calling the built in function `unlink/1` with the process identity of the linked process as argument.

If a process terminates abnormally, by an uncaught exception, all processes linked to it will be informed by a special type of message, called a *signal*. Unless the boolean process flag `trap_exit` is true, the signalled process will also terminate abnormally and its linked processes are informed in the same way, i.e., the failure spreads. If the process flag `trap_exit` is true, the signalled process is not terminated, and may use the received information to take some recovery action. We will say that a process *traps exit* when the process flag `trap_exit` is true; this is analogous on an interprocess level to the catching of exceptions on the intraprocess level. Supervisor processes, described in Section 2.4.2, use this mechanism to monitor termination of their child processes, and to restart them, possibly after performing some cleaning up actions.

2.4 Behaviours

OTP behaviours are support libraries used to implement common design patterns in distributed fault tolerant hardware. I have called this section behaviours, in the terminology of OTP, as opposed to design pattern [Gamma et al. 1998] since the definitions of the behaviours lacks the abstraction of implementation detail and regimented form of presentation to be called design patterns proper. The behaviours are rather library realisations supporting design patterns.

A process that is written using OTP behaviours is structured into a generic part provided in a library module and a callback module written by the application developer . The process is started with a call to the *Behaviour*`:start(`*Module*`,` *Args*`)`, whereupon a process is created and the *Module*'s `init` function is called with *Args* to initialise the behaviour. If the initialisation succeeds, the call returns `{ok,`*Pid*`}` and the process continues

with its execution relying on the callback module to define its actions. If the initialisation fails, the process fails and the call will return `error`.

There is for most behaviours also a `start_link` family of functions that will use `spawn_link` to create the processes; in fact, for the supervisor behaviour there is only the linking version. The start functions also exist in a version where there is a name argument under which the created processes is registered. When registering the processes one can choose whether one wants it registered locally, as described in Section 2.3.2, or globally using the `global` module, the latter is a kernel utility module that enables registering of symbolic names of processes on all ERLANG nodes connected.

There are two exceptions from these principles: the application and generic event handler behaviours, that do not have a callback module and as a consequence does not call an `init` function on creation.

An important design pattern in ERLANG/OTP is the supervision structure which consists of a tree of supervisors monitoring their children through links. The children processes normally execute one of the OTP behaviours or are connected to their supervisor via a `supervisor_bridge`. The supervisor bridge is a behaviour that allows non behaviour processes to be managed by supervisor.

The six behaviours `application`, `supervisor`, `supervisor_bridge`, `gen_event`, `gen_fsm` and `gen_server` will be described below, each in its own section.

2.4.1 Application

Applications are packagings of system components, and have a number of resources such as modules, registered names and processes. The processes can be loaded, started and stopped together and it can be checked that the needed resources are available when loading the application.

Associated with an application is a resource file which declares the resources needed by the application, such as the names that will be registered by the application, and which other applications have to be running before the application is started. One important declaration in the resource file is what function should be called in order to actually start the application, returning the `{ok,`*Pid*`}` if successful and `error` otherwise.

When starting a process with the call `application:start(`*App*`)` a system process called the application controller will locate the application's resource file and determine if the required applications are already started. Should any of the required applications be missing the call will return `error`. If the required applications are present, the application controller will call the start function declared in the application's resource file.

An application can be one of three different types: `permanent`, `transient` or `temporary`. The type decides what happens to the application when it terminates, in a similar way to the children of a

supervisor, see Section 2.4.2. The application controller acts as a supervisor for applications.

2.4.2 Supervisor

Supervisors are used to structure applications for failure recovery. A supervisor is a process which has a number of children which it monitors through links. When a child fails, one or more children are restarted, possibly after first shutting down one or several children. A child is shut down using the `exit/2` built in function with reason `shutdown`; if the child being shut down is also a supervisor, this reason tells supervisors to shut down their children before terminating.

The callback module's `init` function determines what children will be statically started by the supervisor whenever it starts or is restarted. Note that children of a supervisor can also be added later, but we will not regard such children as part of the static process structure. The value returned by the `init/1` function should either be of form `{ok,{{`*RestartStrategy*`,`*MaxR*`,`*MaxT*`},[`*Childspec*`]}}` or `ignore` indicating that the supervisor should not start and the result of the call to `supervisor:start_link` is `ignore`. Each *Childspec* is of form `{`*Id*`,{`*M*`,`*F*`,`*A*`},`*Restart*`,`*Shutdown*`,`*Type*`,`*Modules*`}`, where:

Id is the identity used to refer to the specific child, if for example the function `restart_child` is used to restart a child explicitly.

{M,F,A} declares how the child should be started (and restarted). The function *F* in module *M* with arguments *A* is called to start the function, and should return either `{ok,` *Pid*`}`, `ignore`, or `error`. The `ok` answer returns the `Pid` of the spawned child, whereas both `ignore` and `error` indicates that no child has been spawned. On an `error` the supervisor shuts down the spawned children and fails, otherwise the failure to start the child is ignored and the supervisor continues to start the other children.

Restart indicates to the supervisor how a failure of normal termination of the child should be dealt with. There are three restart strategies:

`permanent`: the child is always restarted if it terminates,

`transient`: it is restarted only if it fails, or

`temporary`: it is not restarted.

Shutdown is the time in milliseconds allowed for the child to perform its shutdown before being exited with the untrappable `kill` reason. The time allowed can also be either `brutal_kill` in which case it is exited with reason `kill` in the first instance, or the other extreme `infinity`

where the supervisor will simply wait until it receives the message indicating that the child has terminated.

Type is either `worker` or `supervisor`, indicating the rôle of the child. It is only children declared as supervisors that are allowed to have the *Shutdown* parameter set to `infinity`, should this be contravened the application will fail to start.

Modules should be the modules used by the behaviour. The *Modules* parameter is used by the SASL's release handler during code replacement.

The precise reaction of the supervisor to the failure of a child, which it should restart according to the child's restart strategy, is determined by the supervisor's *RestartStrategy* which is either:

`one_for_one:` only the terminated child is restarted,

`one_for_all:` all children are shut down and then restarted, or

`one_for_rest:` all children started after the terminated child are shutdown and restarted.

There is a limit to the number of times a supervisor will restart its children, given by two numbers *maxR* and *maxT*. If more than *maxR* restarts are made within *maxT* seconds the supervisor fails. This mechanism allows the system to attempt a more global recovery action if restarting only the children of a particular supervisor is insufficient.

2.4.3 Generic Event Handler

The generic event handlers are managers for a number of event handlers, each of which can be added and removed dynamically. The event manager applies every present event handler, via a call to the callback module, to a received call or notification.

2.4.4 Generic Finite State Machine

The generic finite state machines are used to write finite state machines, where the callback module for each state has a function which describes the transitions made on events.

2.4.5 Generic Server

The generic servers provide a simple way of writing the server part of client server applications, where the `gen_server` module handles debugging and termination of the parent.

2.4.6 Supervisor Bridge

A supervisor bridge enables a subsystem, not originally intended to be part of a supervision hierarchy, to be connected to a supervision hierarchy. The process having the `supervisor_bridge` behaviour behaves as a bridge between the supervision tree and the subsystem.

3. Analysis of Failure Recovery Through Supervision Structures

In this chapter I discuss how supervision structures are used in ERLANG to construct fault tolerant concurrent applications and what properties these structures must have in order to correctly implement fault tolerance through failure recovery. I will then investigate how these properties can be violated, and how the violations can be detected by automatic analysis. I will also present programming conventions that help to avoid these problems. Finally is presented how to detect certain violations of the specification expressed in the application's resource file.

3.1 Supervision structures

One of the chief methods of building fault tolerant systems in ERLANG/OTP is using the supervisor behaviour to build supervision structures. These structures are trees of processes, in which the children are connected to their parents by links. The leaves of the tree are "worker processes" which perform the actual function of the system, and the non-leaf parts are supervisors that create and monitor their children.

The monitoring is on the level of process failures, which would be far too coarse grained in a language less process centric than ERLANG. In ERLANG the light weight processes make it feasible to have a separate process for each task. A task can be quite small, such as handling one party in a telephone call.

The objective of the fault handling mechanism of the supervision structure is to restart the process that has failed. Processes can have interdependencies, and therefore more than one process may have to be restarted in order for the system to be able to perform its function. Should the supervision structure have to restart other processes than the failed one, these processes are first shut down and are replaced by new processes. The reason that a new process is created rather than resetting the old one, is that information that the process had may have become obsolete, e.g., the identities of restarted ports. So the intention of the supervision structure is to restart the rôle that the process had, not the process.

A system built according to the design principles of OTP will set up the supervision structure during the initial start phase, using the OTP behaviours. Later during the execution of the system there may be a number of processes

created to perform specific tasks, that terminate when the task is performed, These processes will be called dynamic processes as opposed to the static processes that make up the supervision structure and persist throughout the operation of the system. In this thesis it is assumed that the systems under inspection are meant for continuous execution, and it is unlikely that the supervision structure changes other than to accommodate for failures. Example of such continuously running system is a telephone switch or a web server.

3.1.1 The Extracted Supervision Structure

The supervision structure that the analysis extracts will be a tree of nodes, each node being a process. In the tree when *node B* is the direct decendant of *node A*, that will denote that *node A* spawned *node B*. We will refer to them as parent and child respectively.

Processes can be linked, which we denote in the structure by a dotted line between the nodes representing the processes.

With each node in the tree will be associated a number of properties that the anlysis has determined, with the supervisors having additional properties that the other nodes do not have. All the nodes have the following properties:

Links: The processes it is linked to.

Trapping exits: Whether the process has the process flag `trap_exit` set to `true`.

Children: Processes it has started.

Side effects: This is the list of all side effects that the processes has performed as part of its initialisation. Examples are interactions with the ETS tables, setting up links to other processes, or sending messages.

Supervisors have these additional properties:

The Restart Strategy: This determines what actions should be taken if a child terminates. The four strategies are `one_for_one`, `all_for_one`, `rest_for_one`, `simple_one_for_one`. For the purpose of the analysis there is no difference between the `one_for_one` and the `simple_one_for_one` so they will be both be denoted as `one_for_one`.

The maximum Restarts: The maximum number of restarts allowed before the supervisor restarts, this is an integer.

The maximum Time: The time span in seconds which the maximum number of restarts should not be exceeded, this is an integer.

Child Specifications: A specification of each child the supervisor starts. Each children has four properties:

{M, F, A}: How the child is started.

Restart: The restart type of the child which specifies in what scenarios the child should be restarted if it terminates. The types are `permanent`, `transient`, and `temporary`.

Shutdown: The property specifies how long the child may spend shutting down once it has been informed by the supervisor to do so. This is either one of the atoms `infinity`, `brutal_kill` or an integer larger than zero.

Type: This is either `worker` or `supervisor`.

An example supervision structure is shown in Figure 3.1. In the figure supervisors are depicted by rectangles, and all other processes by circles. Solid arrows indicate parent-child relationships and the dashed lines show links between processes. Within each process is either a registered name or a process id of the form `<number>`. Associated boxes show values of parameters and flags.

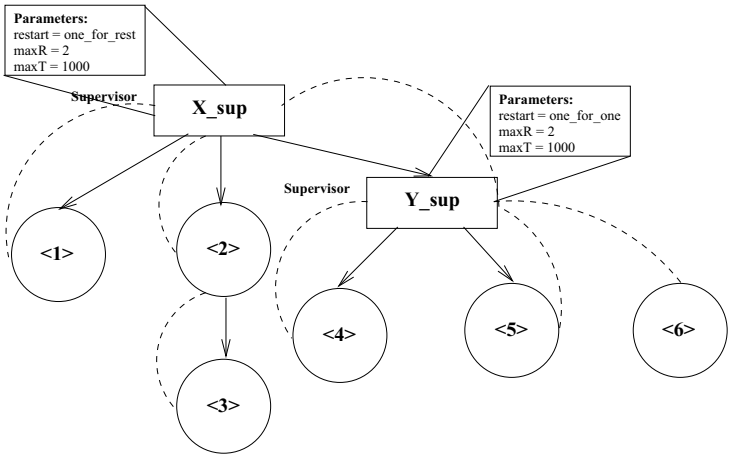


Figure 3.1: Supervision structure example.

3.2 The Effect of Failures on the Supervision Structure

The effect of a process failure on the supervision structure can be summarised as follows:

(1) All processes linked to the failed process are notified by signals. If in Figure 3.1 the process `<2>` fails, the supervisor `X_sup` and process `<3>` are informed.

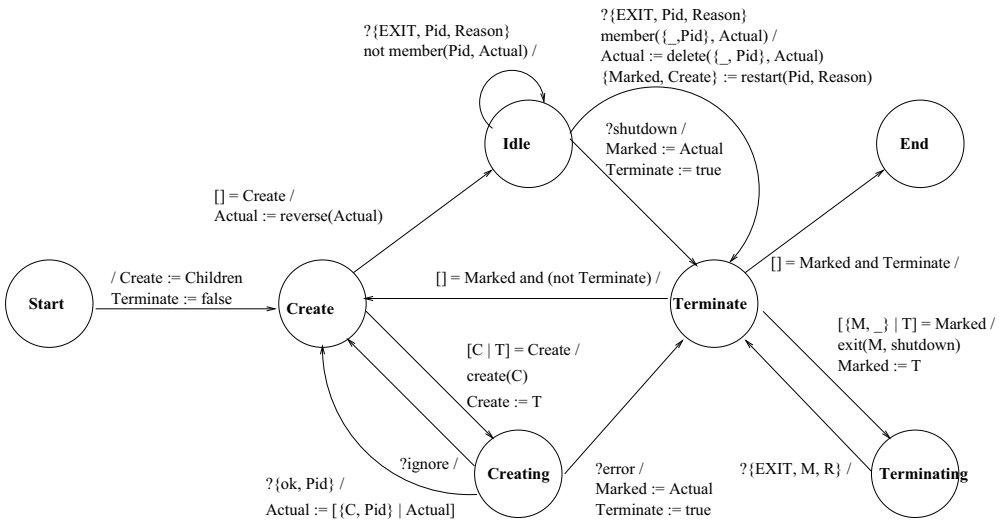
(2) The different responses by notified processes:
   (a) Any notified process that does not trap exits will fail and the effect spreads. If in Figure 3.1 the process <3> fails, the process <2> fails.
   (b) When a supervisor is informed that a child has failed, it will shut down some of its other children. What children are shut down depends on the supervisor's restart strategy, as described in Section 2.4.2. A supervisor shuts down the children in the inverse order of their creation and waits for a child to terminate before shutting down the next. If the child has restart strategy `temporary`, then no action is taken. If in Figure 3.1 the process <1> were to fail, the supervisor `X_sup` would shut down first supervisor `Y_sup` and then process <2>.
   (c) If a supervisor is informed that a child has failed, and thereby the allowed maximum rate of failures has been exceeded, then the supervisor will shut down all its children and fail. If in Figure 3.1 the process <1> were to fail more than twice during one second, then supervisor `X_sup` would fail.
   (d) If a supervisor is informed that a process, which is not a child of the supervisor, has failed the supervisor ignores this. If in Figure 3.1 the supervisor `Y_sup` is informed that processes <6> has failed, this will be ignored.

(3) A shutdown message from the supervisor results in:
   (a) Since shutting down is achieved using `exit` the process will fail and the effect will spread according to (2a). If in Figure 3.1 the supervisor `X_sup` would shut down the process <2>, then the process <3> would fail.
   (b) If a shutdown child is a supervisor this will cause it to shut down its children in turn, shut downs propagating down the supervision structure. If in Figure 3.1 then supervisor `X_sup` would shut down the supervisor `Y_sup`, then the supervisor `Y_sup` would shut down processes <5> and <4>.

(4) When a supervisor has shut down all the children it is required to shut down it will restart them, unless it was being shut down itself, in which case it terminates. If in Figure 3.1 then the supervisor `X_sup` would have shut down the supervisor `Y_sup` because it had been informed that processes <2> had failed, then it would restart first process <2> and then supervisor `Y_sup`.

An informal, but more detailed, presentation of the model of supervisors used here is found in Figure 3.2. The figure is a state transition graph describing the different phases of the supervisor with guards, actions and communication on the edges between the states.

The guards, and all the other actions, are in a pseudo ERLANG syntax, of form: *Guard / Actions*. Each guard is either a "`receive`" from the message buffer of the form `?{ok, Pid}` meaning that there is a message matching




```
create({Id, {M, F, A}, Restart, Shutdown, Type, Modules}) ->
  apply(M,F, A).

restart(Pid, Reason) ->
  {Id, MFA, Restart, Shutdown, Type, Modules} =
    get_child(Pid, Actual),
  case {Restart, Reason, RestartStrategy} of
    {temporary, _, _} -> {[], []};
    {transient, normal, _} -> {[], []};
    {_, _, one_for_one} -> {[{C, Pid}], [C]};
    {_, _, one_for_all} -> {Actual, Children};
    {_, _, one_for_rest} -> {select_after(Pid, Actual),
                             select_after(C, Children)}
  end.

select_after(X, [X | T]) -> T;
select_after(X, [_ | T]) -> select_after(X, T).

get_child(Pid, Actual) ->
  hd([Child || {Pid1, Child} <- Actual, Pid1 == Pid]).
```

Figure 3.2: The Behaviour Model of the Supervisors.

the ERLANG term `{ok, Pid}` in the message buffer, or a boolean expression that may contain explicit matches. If more than one guard is satisfied, the one with the message earliest in the queue is chosen.

In the graph we use global variables that are assigned their value (`Create := Children`), or bound by matches (`[C | T] = Create`). The variable `Children` is a list of tuples returned by the supervisor's `init` function (see Section 2.4.2). The variable `Create` is a list of tuples, on the same format as `Children`, describing all the children that remain to be created, and `Actual` is a list keeping track of the pids of the created children. `Marked` is a list of processes to be shut down, and the `Terminate` is a boolean variable that indicates whether the supervisor is shutting down itself.

In the definitions of auxiliary functions, which are written in ERLANG, the global variables of the graph occur as free variables. The free variable `RestartStrategy` in the `restart/2` function is the restart strategy of the supervisor itself.

The supervisor process starts in the `Start` state and setting all children to be created goes to the `Create` state. In the `Create` state either all children have been created, whereupon it goes to the `Idle`, or spawns a child and goes to the `Creating` state.

In the `Creating` state there are three distinct alternatives, the child can fail in a way that allows continued start of the children, which it indicates by responding `ignore` to the supervisor. The child created can also fail in an intolerable manner indicated by the `error` message, in which case the `Terminate` variable is set to `true`, the list of processes to be terminated `Marked` is set to the `Actual` processes, and the supervisor goes to the `Terminate` state. Should the creation of the child go well, indicated by `{ok, Pid}`, the pid is noted in the `Actual` list and the supervisor returns to the `Create` state.

From the `Terminate` state the supervisor will iterate over the list of `Marked` processes shutting them down using the `exit/2` built in function, waiting in the `Terminating` state for each to actually terminate as determined by the `{'EXIT', Pid, Reason}` message. When all the processes marked have terminated, the supervisor either goes to `End` if the `Terminate` variable is true, or goes to the `Create` state otherwise.

In the `Idle` state the supervisor waits for a message indicating that either a process has failed or for a shutdown message from its parent. If the supervisor gets the `shutdown` message it marks all the actual processes for termination and sets the `Terminate` variable before going to the `Terminate` state. Should the supervisor receive a processes failure notification `{'EXIT', Pid, Reason}` and the process `Pid` is a child, the supervisor removes process `Pid` from the `Actual` list, and depending on the restart strategy of the supervisor and child, sets the list for terminate `Marked` and restarts `Create` before going to the `Terminate` state. Should the failed process

`Pid` not be a child of the supervisor, this message is simply ignored and the supervisor stays in the `Idle` state.

3.3 Essential Properties

The primary function of the supervision structures is: that when a process fails, repair the supervision structure in such a way that normal functionality can be delivered by the system. This would in the nomenclature of Gärtner (see Section 1.3.1) be *non-masking* fault tolerance.

Equally important is that the fault tolerance mechanism does not conceal a persistent error. If a persistent error is present in a subsystem and the fault handling, through restarts, were to perform repeated restarts without any restriction, then the system would find itself in a livelock. The system would neither fail nor deliver its intended functionality.

These two properties of repair and non concealment can be formulated as:

> Property `P1` (repair): Whenever a process that takes part in the supervision structure fails, the supervision structure returns to the process structure prior to the failure after a reasonable delay.
>
> Property `P2` (non concealment): When the cause of a failure is not transient or sufficiently infrequent to let the application function acceptably, only a small number of recoveries should occur before the supervision structure fails.

Below we present more details on what these two properties entail, with examples of violations of properties `P1` and `P2` in the supervision structure shown in Figure 3.3.

3.3.1 Analysis of the Repair Property

The repair property, or simply `P1` for short, implies that in the recovery actions caused by a failure, **(a)** each processes that fails is replaced by an equivalent "restarted" process in the structure, and **(b)** each process replaced by a restarted process has indeed terminated. We now claim that, in fact, properties **(a)** and **(b)** are sufficient to guarantee property `P1`, under the two extra assumptions that **(c)** a non-supervisor that is linked to a failed process does not trap exits, and **(d)** the initialisation of a restarted process creates the same structure as the process it replaces. Property **(c)** mirrors the recommendation that only supervisors trap exit. Property **(d)** is true if the initialisation of a process is not dependent on configuration parameters that change during system operation; it holds in all but one of the applications that we have analysed as described in Chapter 7.

To show the claim assume that a supervision structure satisfies properties **(a)** and **(b)**. We then prove property `P1` by structural induction over the supervision structure, starting from its leaves. For a leaf process, the property is immediate. Consider a non-leaf process p, and assume that the substructures rooted at the children of p satisfy `P1`. If p fails, then it will be restarted (by **(a)**), and (when reinitialising, by **(d)**) recreate the entire substructure rooted at p, thus establishing `P1`. If p does not fail, then if no child fails, the connections between p and its children are preserved. If a child fails, then if p is a supervisor, it will restore its children and its connections to them, otherwise (by **(c)**) p must fail, and we are back to the previous case.

The possible violations of property `P1` are the following:

case 1: A process is terminated but not restarted, because it is not connected by a sequence of links to a supervisor, directly or in several steps, where no intermediate process traps exit.

In Figure 3.3 the process `<3>` might not be restarted if it fails since process `X_serv` traps exit. From the information in the Figure 3.3, we can not determine if process `<3>` is restarted since this depends on the dynamic execution of `X_serv`.

case 2: A process is restarted without having terminated first, because its parent is a non-supervisor and it is either not linked to its parent or traps exit. In Figure 3.3 the process `<3>` might be restarted without having terminated if `X_serv` fails, since it traps exit.

Another reason is if a supervisor has less time to shut down its children, than the combined shutdown times of the children. In this case, when time expires for the supervisor all remaining children will be left unterminated. In Figure 3.3 processes `<1>` and `<2>` have the combined shutdown time of 4 seconds whereas the supervisor `Y_sup` has 2 seconds. Even if the `<2>` were linked to `Y_sup`, process `<1>` might not be terminated when `Y_sup` is shut down.

Should the supervisor with limited shut down time be deadlocked by a child the remaining children would remain. In Figure 3.3, when shutting down `Y_sup` process `<2>` will deadlock `Y_sup`, as described below, and `Y_sup` consequently will not shut down `<1>`.

case 3: A supervisor is deadlocked, when trying to shut down a child which is not linked to the supervisor, since the supervisor is never informed that the child has terminated. In Figure 3.3 the supervisor `Y_sup` is deadlocked when trying to shut down process `<2>`.

Another reason for a supervisor to deadlock is that a child with shut down time `infinity` that traps exit does not terminate, e.g., a deadlocked supervisor.

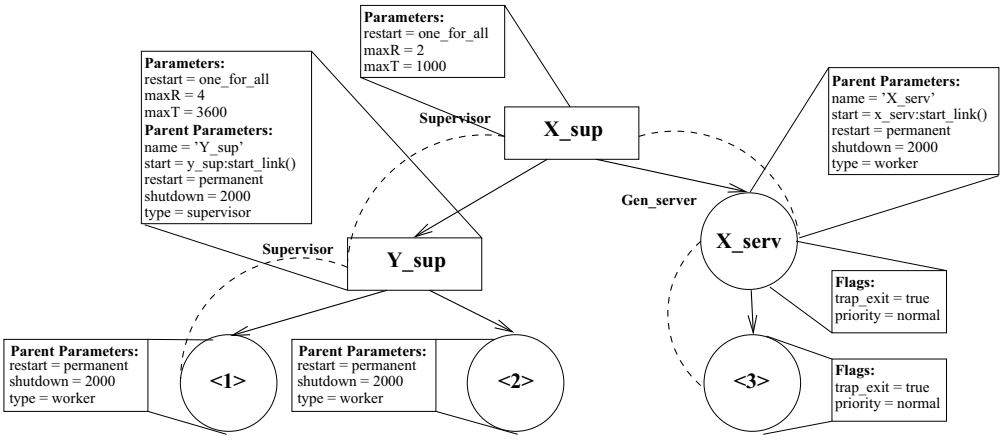


Figure 3.3: Faulty supervision structure example.

In order to determine if property `P1` is violated by an application our tool extracts a set of supervision structures as described in Chapter 6, and for each process records what the effects are of a failure.

The tool will simulate the steps (1) to (4) described earlier in Section 3.3. As a first step the tool records processes that will fail by following links via processes that do no trap exit (2a). As a second step, for each supervisor with a failed child, it records what children are shut down (2d). Which children are shut down is decided by the supervisor's parameters, and for each shut down child we have to repeat steps (3a) followed by (2a) or (2d). As a final step (4), the children that would be restarted by the supervisor are recorded.

Note that a supervisor can fail for two reasons. First it can have a maximal restart frequency of 0, and then the steps (2c) and (1) are performed recursively up the supervision structure until reaching a supervisor that does not fail or until the entire supervision structure has failed. Secondly the number of restarts can exceed the highest allowed restart frequency; in this case both, when the supervisor fails (2c) and when it restarts (2d,4), its children must be recorded.

When all effects of the failure have been applied the tool can determine if property `P1` has been violated by checking that properties **(a)** and **(b)** are satisfied for each process. For example, if a process has failed but not restarted, then property `P1` is violated according to **case 1**.

3.3.2 Analysis of the Non Concealment Property

First I restate the property of non concealment:

> Property P2 (non concealment): When the cause of a failure is not transient or sufficiently infrequent to let the application function acceptably, only a small number of recoveries should occur before the supervision structure fails.

A supervision structure can violate Property P2 in two ways: **(a)** a substructure fails and is restarted too many times before the supervision structure itself (represented by its root processes) fails, **(b)** the supervision structure never fails although there are an unbounded number of repeated failures in the structure.

In order to reason about these possible violations, we have to define the maximum rate of failures (MRF for short) that can occur in a structure before the structure itself fails. In order to calculate the MRF of the supervision structure, we have to calculate the MRF of the individual processes in the structure. The MRF for a process can be calculated from the parameters of the supervisors higher in the structure, as follows.

The MRF for a process with supervisors $s_1, \ldots, s_n$ above in the process structure, where the supervisors are numbered from the top down is given by:

$$\begin{aligned} \text{MRF} &= \mathit{Restarts} \ / \ \mathit{TimeUnit} \\ \mathit{Restarts} &= (\prod_{i=1}^{n} (\mathit{MaxR}(s_i)+1)) \quad 1 \\ \mathit{TimeUnit} &= \min_{i=1}^{n} \mathit{MaxT}(s_i) \end{aligned}$$

where $\mathit{MaxR}(s_i)$ is the parameter *maxR* for the i^{th} supervisor and analogously for *maxT*.

The total number of allowed failures (*Restarts*) is the product of the number of allowed failures plus one (so that the supervisor fails) for each level of the supervision structure, except for the highest level where only the allowed number can occur since otherwise the structure fails.

The minimum time for the failure to occur is bounded by the minimum for all levels, since a failures does not propagate up a level if the time span in which they occur exceeds the minimum of any level.

As an example, in Figure 3.3, process <1> has supervisors X_sup, Y_sup above it, $\mathit{Restarts} = ((2+1) \ \ (4+1)) \quad 1$ and $\mathit{TimeUnit} = \min(1000, 3600)$ so the MRF is 14 failures in 1000 milliseconds.

In order to determine the MRF for a supervision structure one must take the maximum MRF for any process taking part in the supervision structure.

Violations of type (a) occur if the MRF of a supervision structure is too high. The meaning of "too high", of course, varies from application to application. We suggest that a threshold for this number shall be provided for each application by designers or by company coding principles.

Violations of type (b) occur if the structure contains a supervisor for which the maximum time it takes to restart one child is larger than the least time between failures needed to cause failure of the supervisor itself.

The maximal time to restart a child is the combined shut down time of the other children that are shut down and the start times of all the terminated children. We can only determine the shut down times[1], but if this is already too large (in effect assuming a start time of 0) we are certain that the combined time to shut down and restart is too large. The least time between failures needed is simply supervisor parameter `maxT` divided by `maxR`.

In Figure 3.3 the shutdown time of `Y_sup` is 2 seconds but its supervisor `X_sup` will fail only if more than 2 restarts are performed within 1 second. If the shutdown of `Y_sup` actually takes 2 seconds then, even if the process `X_serv` fails immediately upon each restart, the supervisor `X_sup` will never fail. This is a typical violation of (b).

When designing an application one wants to determine parameters of the entire supervision structure as well as over individual processes; restarts and shutdown times can be calculated automatically. Interesting parameters include: (a) the maximum restart frequency for any process before the application restarts; (b) the largest shutdown time allowed for any process; (c) the largest shutdown time allowed for the entire application, i.e., the combined shut down times of all the parts.

3.4 Design Conventions

There exist a number of coding conventions designed to prevent violation of properties `P1` and `P2`, which catch a subset of these violations. The tool checks whether the conventions are followed:

All processes should create their children using `spawn_link`, rather than `spawn`, and children should not `unlink` from their parents. If a process is started with `spawn` rather than `spawn_link` it can fail before it has time to link to its parent, in which case its parent will not be informed.

The max restart frequency of intermediate supervisors should be 0 in order to minimise the MRF of leaf processes in the supervision structure.

Only supervisors should have shutdown time infinity and all non-supervisor children of supervisors should have shutdown set to a limited time. This ensures that the supervisor has time to terminate its

[1]There is an OTP library that allows the designer to set an upper limit on the time it takes to start and initialise a process, if that was used throughout we could give an upper limit on the startup time and as a consequence on the restart time as a whole.

children and that no child can indefinitely block the shutdowns. This eliminates one of the causes of violation of property `P1` in **case 3**.

3.5 Adherence to Specification

Each application has an associated resource file which declares resources needed by the application, such as the names that will be registered by processes of the application. The *application behavior* library module will perform certain checks and code/application loading in association with the resource file, but it is the responsibility of a user programmed callback module to actually start and initialise the appropriate processes. The resource file can thus be viewed as a specification, which should be compared against the process structure that is actually created by the user programmed module; there is no support in OTP for checking that the application's actual process structure agrees with the specification. An example is the registered names in the application's resource file: should the actual application not register any of the names, or a registered name not be mentioned, this can be viewed as a specification violation.

If the analysis cannot find important parts of the specification's process structure, this is in many cases a sign that the coding of the application does not follow the recommended coding style. One example is when initialisation code is not performed within the `init` callback-function, as recommended, but rather in some other function which is executed after `init`; another example is when an OTP behavior process is not registered in the way supported by OTP, but rather by calling an ERLANG built in function.

4. Semantics

In this chapter we present a semantics for CORE ERLANG. The semantics will be used for the extraction of the supervision structure. The extraction is made on CORE ERLANG derived from source code of the ERLANG application via the OTP compiler. After a presentation of the intermediate language CORE ERLANG, a formal semantics for single processes is given. Chapter 5 presents an abstract version of the semantics and a proof of abstraction.

4.1 Overview

To make the presentation as accessible as possible we have chosen to first present the semantics in this chapter, then the abstraction of the semantics with the proof of abstraction in the following chapter and finally how the actual extraction can be implemented based on the abstraction is presented in Chapter 6. In fact we arrived at the final formulation of the semantics, abstraction and the implementation in the opposite order. This does not mean that we decided on the semantics after we had the implementation of the process structure extraction, since the semantics is given by the ERLANG definition together with the OTP implementation.

We first implemented the symbolic evaluator, and the reason was that it was vital to strike the correct balance of abstraction in the evaluator to get tractable evaluation times, whilst retaining sufficient information for further analysis. After we had implemented an abstraction that suited our purposes we had to give a definition of the ERLANG semantics that would allow us to show the abstraction with greatest clarity, while of course being correct. By correct we mean a conservative overapproximation of the original semantics, i.e., the actual evaluation should be a member of the set of possible evaluations the symbolic evaluation computes.

Thus the aim of the semantics is to serve as a basis for the extraction of an ERLANG application's supervision structure. The extracted structure will be used for the analyses of fault handling properties, as described in the previous chapter. The supervision structures, described in Section 2.4, consist of the involved processes and their parameters. The structures are setup using the `init` functions of the callback modules written to implement the specifics of the behaviour, together with the general parts from the library modules. For

the general library part of the behaviours, manually constructed models are used.

More specifically, what we want to capture from the setup of the supervision structure is: what processes are created, what OTP behaviour does the processes realise and what are its parameters associated with the behaviour, which processes are linked to each other, and finally what other side effects have occurred during the setup.

When building a model of the supervision structure of an ERLANG application, it would be preferable to fully automate the construction, for reasons of speed and to minimise the risk of errors. It has however proved to be impracticable in this case, since the complexity and size of the systems make that impossible.

To deal with the complexity, which to a great degree stems from the handling of special cases in the behaviours' initialisation, we have used manually constructed models of the behaviours' initialisations. The result of this is that we get a two part strategy for extracting the supervision structure from an ERLANG application, first is the automatic part which extracts the setup of individual processes via their behaviour's `init` function, i.e., the code called in the callback module. The second part is the use of a manually constructed model of the behaviour's library source code that builds the supervision structure from these individual processes.

For this two part extraction strategy to work, we restrict the extraction to the static parts of the supervision structure that are implemented using behaviours. This is less of a restriction than it might first appear, since the structures that provide fault tolerance will most likely be static (except when they are shutting down and restarting processes in order to repair the system after a failure). If the system is designed according to OTP design principles the processes will be implemented using behaviours.

The supervisors in the supervision structure will start their children one after the other and only start the next child after the previous child has reported that it has started and initialised successfully, via a message sent to the supervisor. This ordered starting of children and synchronisations between the children and supervisors, provides a total ordering on the initialisations. Having a total ordering of the actions performed in the creation and initialisation of the supervision structure means that we do not need to explicitly handle the concurrency, or indeed the possible distribution, of the system. The execution consists of the initialisation of the supervisor and the successive creation and initialisation of its children; this is repeated recursively down the supervision structure.

The two semantics presented below are of the intermediate CORE ERLANG language with approximations made of the communications with other processes that are not part of the behaviours' library. The semantics are operational semantics [Plotkin 1981] where each construct is defined through what changes it makes to the program state, i.e., the construct is seen as an opera-

tion on the program state. The first semantics is a concrete semantics of CORE ERLANG and the second semantics is the abstracted semantics actually used in the extraction. The abstracted semantics is presented by describing how it differs from the concrete, in order that it should be clear how the abstract semantics relates to the concrete.

The operational semantics of [Fredlund 2001, Huch 1999] both describe ERLANG and in Fredlund's case it is aimed at proof rather than implementation of an interpreter and is organised more like a natural semantics although it is a small step operational semantics.

Other possible alternative types of semantics would be denotational semantics [Stoy 1977, Schmidt 1986], where a language construct is given meaning by translation into a mathematical formalism; or axiomatic semantics [Hoare 1969], where a set of axioms and proof rules over the syntax of the language provides the definition. More on semantics for programming languages can be found in in textbooks such as [Winskel 1993, Nielson and Nielson 1992].

4.2 Core Erlang

CORE ERLANG is an intermediate language that is used internally in the ERLANG/OTP compiler. The language was devised with the intent of decoupling the parsing and preprocessing of the front-end from the code generation of the back-end of the compiler. The reason for this separation is partly to enable other implementations but also to make use of optimisation techniques commonly used in other functional languages which are hard to implement both for the rather complex and redundant structure of ERLANG and for the imperative instruction sets of the abstract machines. The optimisation techniques include algebraic transformations, specialisations, and advanced inlining.

The development of CORE ERLANG has been a cooperation between the OTP development team at Ericsson and the High Performance Erlang group at Uppsala University and is described in [Carlsson et al. 2000, Carlsson 2001]. The CORE ERLANG described in this chapter should not be confused with the core fragments of ERLANG used in [Huch 1999, Huch 2001, Dam et al. 1998a, Dam et al. 1998b, Arts and Dam 1999] which are truly fragments of ERLANG as opposed to CORE ERLANG, which is a separate language into which the whole of ERLANG can be translated.

The CORE ERLANG language bears many similar traits to ERLANG although there are some notable differences, such as the `letrec` construct which allows the definition of local (possibly recursive) functions. All the constructs are described informally in the remainder of this section. The following syntactical conventions will be used: all alphanumeric `terminal` and *non terminal* symbols are identified by their type faces. All non alphanu-

meric symbols in the grammars are terminals with the exception of the meta-character | which is quoted '|' when occurring as a terminal grammatical term.

4.2.1 Modules

As with ERLANG, the grouping unit in CORE ERLANG is the module where a number of functions are defined. The general format of a CORE ERLANG module is shown in Figure 4.1. An example CORE ERLANG module is shown in Figure 4.7, it is the result of compiling the example ERLANG module in Figure 2.1 into CORE ERLANG.

$$
\begin{array}{rcl}
\mathit{module} & ::= & \texttt{module}\ \mathit{atom}\ \texttt{[}\mathit{fname}_{e1},\ldots,\mathit{fname}_{ek}\texttt{]} \\
 & & \texttt{attributes}\ \texttt{[}\mathit{atom}_1\texttt{=}\mathit{const}_1,\ldots,\mathit{atom}_l\texttt{=}\mathit{const}_l\texttt{]} \\
 & & \mathit{fname}_1\texttt{=}\mathit{fun}_1 \\
 & & \quad\vdots \\
 & & \mathit{fname}_m\texttt{=}\mathit{fun}_m \\
 & & \texttt{end} \\
 & & \\
\mathit{fname} & ::= & \mathit{atom}\ /\ \mathit{integer} \\
\mathit{fun} & ::= & \texttt{fun}\,\texttt{(}\mathit{var}_1,\ldots,\mathit{var}_2\texttt{)}\ \texttt{->}\ \mathit{exprs} \\
\mathit{var} & ::= & \mathit{VariableName}
\end{array}
$$

Figure 4.1: CORE ERLANG module syntax.

A module begins with the `atom` module followed by the name of the module and a list consisting of the names of the exported functions. Each function name (*fname*) consists of the name of the function and its arity. The meaning of the following attributes are implementation dependent and they play no rôle in our analysis; an example attribute is that of the author found in the example module in Figure 4.7. After the attributes follow the function definitions, if an exported function is not matched by any definition a compile time error will occur. Each function definition consists of the function name equated to a nameless function, or fun as they are called in ERLANG terminology.

The funs in CORE ERLANG only have variables as arguments and one single clause; if the originating ERLANG function consisted of more complex matching expressions or several clauses, this is expressed using a conditional in the expression body of the fun. The reason for the single clause funs is that of uniformity, since programs are to be compiled to CORE ERLANG as opposed to written in ERLANG, there is no reason to have redundant ways of writing the same code.

The variables in CORE ERLANG are alphanumeric sequences starting with either an uppercase letter or '_', unlike ERLANG there is no anonymous variable. All atoms in CORE ERLANG are quoted with single quotes and may contain any alphanumeric and special character except control characters, slash (/), and the single quote (').

4.2.2 Expressions

All expressions in CORE ERLANG return a possibly empty sequence of values, written enclosed in angular brackets of form $< value_1, \ldots, value_n >$. Expressions return sequences of values instead of only singular values in order to allow certain optimisations be made at CORE ERLANG level; when a value sequence consists of one value the brackets may be omitted. The use of an expression must match the number of values it produces and in constructing a value sequence out of expressions, these expressions must all produce one value. Should an expression be the argument of a list or tuple constructor, or function call it must also result in one value.

$$
\begin{array}{rcl}
exprs & ::= & expr \mid <expr_1, \ldots, expr_n> \\
expr & ::= & var \mid fname \mid fun \mid literal \\
 & \mid & \texttt{[}exprs_1\ \texttt{'|'}\ exprs_2\texttt{]} \\
 & \mid & \texttt{\{}exprs_1, \ldots, exprs_n\texttt{\}} \\
 & \mid & \texttt{let}\ vars = exprs_1\ \texttt{in}\ exprs_2 \\
 & \mid & \texttt{letrec}\ fname_1 = fun_1 \ldots fname_n = fun_n\ \texttt{in}\ exprs \\
 & \mid & \texttt{case}\ exprs\ \texttt{of}\ clause_1 \ldots clause_n\ \texttt{end} \\
 & \mid & \texttt{apply}\ exprs_0\ \texttt{(}exprs_1, \ldots, exprs_n\texttt{)} \\
 & \mid & \texttt{call}\ exprs_{n+1}\texttt{:}exprs_{n+2}\ \texttt{(}exprs_1, \ldots, exprs_n\texttt{)} \\
 & \mid & \texttt{primop}\ Atom\ \texttt{(}exprs_1, \ldots, exprs_n\texttt{)} \\
 & \mid & \texttt{try}\ exprs_1\ \texttt{catch (}var_1, var_2\texttt{) -> }exprs_2 \\
 & \mid & \texttt{receive}\ clause_1 \ldots clause_n\ \texttt{after}\ exprs_1\ \texttt{->}exprs_2\ \texttt{end} \\
 & & \\
vars & ::= & var \mid <var_1, \ldots, var_n>
\end{array}
$$

Figure 4.2: CORE ERLANG expressions.

The different types of expressions are summarised in Figure 4.2, they consists of variables (of which function names really are only a special case), funs, constants, data type expressions, binding expressions, conditional expressions, various types of function calls, exception handling, and message

retrieval. All these different types of expressions, not already described, will be described in more detail below.

4.2.2.1 Constants

There are two types of constants, the atomic constants that are not constructed out of other constants, and the compound constants that are either lists or tuples consisting entirely of constant subterms. The atomic constants are: integers, floats, atoms, and characters. These types are of the same format as in ERLANG (see Section 2.2.1.1) with the exception of atoms which have been described above. The characters are merely syntactic sugar for the corresponding integer that encodes the character in question.

$$\begin{array}{rcl} const & ::= & literal \mid String \mid [const_1\ \texttt{'|'}\ const_n] \\ & \mid & \{const_1, \ldots, const_n\} \\ literal & ::= & integer \mid float \mid atom \mid Char \end{array}$$

Figure 4.3: CORE ERLANG constant literals.

The compound constants are tuples, lists and strings have the same format as in ERLANG (see Section 2.2.1.2), where string constants are nothing more then a special syntax for lists consisting of characters.

4.2.2.2 Data Types Expressions

There are two types of data type expressions, used to construct elements of the compound data types tuples and lists. In all essential parts, these constructs are like those of ERLANG described in Section 2.2.1.2.

$$\begin{array}{rcl} expr & ::= & [exprs_1\ \texttt{'|'}\ exprs_2] \\ & \mid & \{exprs_1, \ldots, exprs_n\} \end{array}$$

Figure 4.4: CORE ERLANG data type expressions.

4.2.2.3 Binding Expressions

There are two types of binding constructs, first the `let` construct which replaces the explicit match of ERLANG (Section 2.2.2), then there is the `letrec` construct used to implement the ERLANG list comprehensions (Section 2.2.7).

A `let` expression binds the n variables occurring in *vars* to the n values returned by expression $exprs_1$, in the evaluation of $exprs_2$. The result of the expression is the values returned by the evaluation of $exprs_2$. Should the num-

$$
\begin{array}{rcl}
expr & ::= & \texttt{let}\ vars = exprs_1\ \texttt{in}\ exprs_2 \\
& | & \texttt{letrec}\ fname_1 = fun_1 \ldots fname_n = fun_n\ \texttt{in}\ exprs
\end{array}
$$

Figure 4.5: CORE ERLANG binding expressions.

ber of variables in the *vars* part differ from the number of values returned by $exprs_1$ there will be a runtime error. There are two important differences between the explicit matching of ERLANG and the `let` construct; the first is that in a `let` expression a variable may only occur once in the *vars* variable sequence. The second difference is that in CORE ERLANG binding scopes are nested as in lambda calculus rather than the per function single scope of ERLANG (Section 2.2.6).

The `letrec` construct binds function name variables to funs much in the same way as a function definition, the only difference to a function definition is that the function name variable is local to the `letrec` expression. The bindings of the `letrec` are available everywhere in the expression, more particularly they are available in the body of the funs defining the function variables making it possible to define recursive and mutually recursive local functions. The result of evaluating a `letrec` expression is the result of evaluating the *exprs* part, with the bindings of the function names $fname_1, \ldots, fname_n$.

4.2.2.4 Conditional Expressions

The only conditional construct in CORE ERLANG is the `case` construct, which has several differences to the ERLANG `case` (Section 2.2.5) although it is similar in intent. The `case` evaluates the *exprs* and matches the clauses patterns sequence against the resulting values in order to determine the clause to execute.

The patterns in the clause bind the occurring variables in the same manner as the `let` construct, and the same restriction that a variable may not be repeated is enforced by the compiler. The pattern sequences of the clauses must have the same number of patterns, which must be the same as the number of values in the result of the evaluation of the *exprs*.

The guards of the clauses may only produce a single value (either the atom `true` or the atom `false`) and may not have any side effects. The guards are a restricted form of expressions that only may consist of variables, constants, constructors, `let`, function call, and exception handling expressions. The function call can only be made to a set of predefined functions that are known to exist and be side effect free, e.g., type tests, comparisons and selectors.

$$
\begin{array}{rcl}
\mathit{expr} & ::= & \texttt{case}\ \mathit{exprs}\ \texttt{of}\ \mathit{clause}_1 \ldots \mathit{clause}_n\ \texttt{end} \\
\mathit{clause} & ::= & \mathit{pats}\ \texttt{when}\ \mathit{guard}\ \texttt{->}\ \mathit{exprs} \\
\mathit{pats} & ::= & \mathit{pat} \mid \texttt{<}\mathit{pat}_1, \ldots \mathit{pat}_n\texttt{>} \\
\mathit{pat} & ::= & \mathit{var} \mid \mathit{literal} \mid \texttt{[}\mathit{pat}_1\ \texttt{'|'}\ \mathit{pat}_2\texttt{]} \mid \texttt{\{}\mathit{pat}_1, \ldots, \mathit{pat}_n\texttt{\}} \\
 & \mid & \mathit{var} = \mathit{pat} \\
\mathit{guard} & ::= & \mathit{var} \mid \mathit{literal} \mid \texttt{[}\mathit{guard}_1\ \texttt{'|'}\ \mathit{guard}_2\texttt{]} \\
 & \mid & \texttt{\{}\mathit{guard}_1, \ldots, \mathit{guard}_n\texttt{\}} \\
 & \mid & \texttt{let}\ \mathit{vars} = \mathit{guard}_1\ \texttt{in}\ \mathit{guard}_2 \\
 & \mid & \texttt{call}\ \mathit{guard}_{n+1}\texttt{:}\mathit{guard}_{n+2}\ \texttt{(}\mathit{guard}_1, \ldots, \mathit{guard}_n\texttt{)} \\
 & \mid & \texttt{primop}\ \mathit{atom}\ \texttt{(}\mathit{guard}_1, \ldots, \mathit{guard}_n\texttt{)} \\
 & \mid & \texttt{try}\ \mathit{guard}_1\ \texttt{catch (}\mathit{var}_1, \mathit{var}_2\texttt{)}\ \texttt{->}\ \mathit{guard}_2
\end{array}
$$

Figure 4.6: CORE ERLANG conditional expressions.

4.2.2.5 Function Expressions

There are three different types of function calls in CORE ERLANG: calls to functions defined in the same module (`apply`); calls to functions that may be defined in another module (`call`); and calls to special implementation dependent functions (`primop`). Examples of all thre usages are found in Figure 4.8. The application of a local functions simply consists of evaluating its arguments and then applying the function which $exprs_0$ evaluates to. The calling of functions defined in other modules is similar although the module in which the function is defined must also be calculated. In most cases the function as well as the module are known statically at compile time.

The purpose of the primitive operations (or primops) is to distinguish between the functions supported directly by the runtime system and the normal functions. They are defined in a module supplied with the system. The name of the operation is an atom, e.g., in Figure 4.7 is a call to the primop `match_fail` which handles the situation where none of the clauses in the originating ERLANG `case` construct would match the values.

4.2.2.6 Exception Handling

Exceptions in CORE ERLANG have two components: a tag and a value. The tag indicates to what class an exception belongs; the class `'EXIT'` signifies an exception raised by the runtime system, and the class `'THROW'` signifies an exception explicitly raised by the programmer. The value of the exception may be any sequence of one value.

When evaluating the `try` expression, if $exprs_1$ does not raise an exception then the result of evaluating $exprs_1$ is the result of the whole `try` expression. Should the evaluation of $exprs_1$ cause an exception to be raised then the result

```
module 'example' ['reverse'/1, 'sort_reverse'/1,
                  'lazy_reverse'/1]
attributes ['author' =
            ['JanHenryNystrom@gmail.com']]

'reverse'/1 =
  fun (_cor0) -> apply 'reverse'/2 (_cor0, [])

'reverse'/2 =
  fun (_cor1, _cor0) ->
    case <_cor1, _cor0> of
      <[], Reversed> when 'true' -> Reversed
      <[H|T], Reversed> when 'true' ->
        apply 'reverse'/2 (T, [H|Reversed])
      <_cor3, _cor2> when 'true' ->
        primop 'match_fail' ({'function_clause',
                              _cor3,
                              _cor2})
    end

'sort_reverse'/1 =
  fun (_cor0) ->
    let <Sorted> = call 'lists':'sort' (_cor0)
    in  apply 'reverse'/2 (Sorted, [])

'lazy_reverse'/1 =
  fun (_cor0) ->
    case _cor0 of
      <List> when try
                  let <_cor1> =
                      call 'erlang':'length' (List)
                  in  call 'erlang':'<' (_cor1, 10)
                  catch (T, R) -> 'false'
                  ->
        apply 'reverse'/1 (List)
      <List> when 'true' -> List
    end
  end
```

Figure 4.7: An example CORE ERLANG module.

$$
\begin{aligned}
\mathit{expr} \;::=\; & \texttt{apply}\ \mathit{exprs}_0\ (\mathit{exprs}_1, \ldots, \mathit{exprs}_n) \\
\mid\; & \texttt{call}\ \mathit{exprs}_{n+1} : \mathit{exprs}_{n+2}\ (\mathit{exprs}_1, \ldots, \mathit{exprs}_n) \\
\mid\; & \texttt{primop}\ \mathit{atom}\ (\mathit{exprs}_1, \ldots, \mathit{exprs}_n)
\end{aligned}
$$

Figure 4.8: CORE ERLANG function call expressions.

$$
\mathit{expr} \;::=\; \texttt{try}\ \mathit{exprs}_1\ \texttt{catch}\ (\mathit{var}_1, \mathit{var}_2)\ \texttt{->}\ \mathit{exprs}_2
$$

Figure 4.9: CORE ERLANG exception handling expressions.

is what exprs_2 evaluates to with var_1 bound to the tag of the exception and var_2 to its value.

4.2.2.7 Message Retrieval

The `receive` construct can be viewed as a case statement looping over the contents of the process mailbox, where the `receive` tries to match its clauses over and over until there is a message present in the mailbox that can be matched by a clause. The first clause which matches an element in the mailbox, whose corresponding guard is true, is selected. The expression of the clause is executed, with bindings provided by the pattern match, and the matched message in the mailbox is removed from the mailbox.

$$
\mathit{expr} \;::=\; \texttt{receive}\ \mathit{clause}_1 \ldots \mathit{clause}_n\ \texttt{after}\ \mathit{exprs}_1\ \texttt{->}\mathit{exprs}_2\ \texttt{end}
$$

Figure 4.10: CORE ERLANG message retrieval expressions.

There is a timeout clause `after` exprs_1 `->`exprs_2 that might break out of the "loop" if the time elapsed in the statement is larger than the value of exprs_1 milliseconds. Unlike in ERLANG (Section 2.3.1) the timeout clause is not optional, however if exprs_1 evaluates to `infinity` the timeout clause will never be evaluated. The exprs_1 is evaluated first, before any matches are tried in the `receive` construct, and should evaluate either to a nonnegative integer or `infinity`. The result of the `receive` statement if the timeout clause is evaluated, is what exprs_2 evaluates to.

4.2.2.8 Syntactic Sugar (do)

A notation for a special application of the `let` construct has been introduced in CORE ERLANG, even if it is redundant, due to the great improvement in

readability it gives. The construct is `do`. The usage is that of evaluating an expression whose result is of no interest, for which merely the side effects that are sought. First $exprs_1$ is evaluated and then $exprs_2$ is evaluated, with the result of the statement being what $exprs_2$ evaluates to. The same result can be achieved by writing `let` *vars* = $exprs_1$ `in` $exprs_2$ where *vars* does not contain any variables occurring free in $exprs_2$.

$$expr \quad ::= \quad \texttt{do}\ exprs_1\ exprs_2$$

Figure 4.11: CORE ERLANG sequential expressions.

4.2.3 Normal Form

```
case <first(_cor1), second(3)> of
  <'true', _cor2> when 'true' ->
    'true'
  <'false', _cor2> when 'erlang':'>'(_cor2, 8) ->
    false'
  <_cor2, _cor3> when 'true' ->
    'erlang':'throw'('error')
end

let <_cor2, cor3> =
  <first(_cor1), let <_cor4> = 3 in second(_cor4)>
in case <_cor2, _cor3> of
   <'true', _cor2> when 'true' ->
     'true'
   <'false', _cor2> when let <_cor3> = 8
                         in 'erlang':'>'
                            (_cor2, _cor3) ->
     'false'
   <_cor2, _cor3> when 'true' ->
     let <_cor4> = 'error'
     in 'erlang':'throw'(_cor4)
end
```

Figure 4.12: Example code snippet with the equivalent normalised version.

To simplify the description of the semantics, we assume that the CORE ERLANG program has been transformed into an equivalent program where

each intermediate result has been given a name, similar to the normal form of [Sabry and Felleisen 1994]. That means that, with the exception of `let`, the arguments of each construct must be variables. An example code snippet, together with its transformed form is shown in Figure 4.12. The transformation into the normal form is defined in Figure 4.13.

4.3 Formal Semantics

In this section is presented an operational semantics of CORE ERLANG. The purpose of the semantics is to faithfully mirror the meaning of the language at the level of detail relevant for the extraction of supervision structures.

4.3.1 Global Context and Resources

An important and involved part of the formal semantics is how we model the handling of message passing, since we only regard one process in isolation. Since we model the processes and not the whole of their environment we have chosen not to explicitly model the stream of incoming messages that have not been delivered to the process' message queue. This stream of messages is modelled by rule (4.38) that may insert any message into the process' incoming queue. All the transformation rules of the semantics are defined later in this section. The message queue is modelled as a part of the global state.

The model of message passing gives rise to nondeterminism: since the semantics does not model the time of arrival for a message, it can not be determined whether the message was delivered before a timeout would occur in a `receive` expression. As a consequence of this nondeterminism, the concrete semantics will abstract over this distinction and include both possible action sequences.

Another important aspect is side effects. Side effects that may influence the execution of the process will be recorded in the state of the process, e.g., the changes made to the ETS tables. We assume that processes not involved in the same ERLANG application do not change the values recorded in the state. Note that processes in the same application may well share resources, such as ETS tables. The node global resources affected by side effects which we will model are:

The ERLANG term storage, or ETS for short, which is a node local database of key/value tables accessed via functions in the `ets` module. The tables are created, destroyed and updated destructively, acting as node global variables. The ETS table has privileges that can be used to restrict the access to the tables.

Registered names described in Section 2.3.2.

$$\mathscr{A}[\![x]\!] \Rightarrow x$$

$$\mathscr{A}[\![literal[l]]\!] \Rightarrow literal[l]$$

$$\mathscr{A}[\![\{e_1,\ldots,e_i\}]\!] \Rightarrow \begin{array}{l}\texttt{let}\ \langle x_1,\ldots,x_i\rangle = \langle \mathscr{A}[\![e_1]\!],\ldots,\mathscr{A}[\![e_i]\!]\rangle \\ \texttt{in}\ \{x_1,\ldots,x_i\}\end{array}$$

$$\mathscr{A}[\![[e_1 \mid e_2]]\!] \Rightarrow \texttt{let}\ \langle x_1,x_2\rangle = \langle \mathscr{A}[\![e_1]\!],\mathscr{A}[\![e_2]\!]\rangle\ \texttt{in}\ [x_1 \mid x_2]$$

$$\mathscr{A}[\![\langle e_1,\ldots,e_i\rangle]\!] \Rightarrow \begin{array}{l}\texttt{let}\ \langle x_1,\ldots,x_i\rangle = \langle \mathscr{A}[\![e_1]\!],\ldots,\mathscr{A}[\![e_i]\!]\rangle \\ \texttt{in}\ \langle x_1,\ldots,x_i\rangle\end{array}$$

$$\mathscr{A}[\![\texttt{fun}(x_1,\ldots,x_i) = e]\!] \Rightarrow \texttt{fun}(x_1,\ldots,x_i) = \mathscr{A}[\![e]\!]$$

$$\mathscr{A}[\![\texttt{let}\ \langle x_1,\ldots,x_i\rangle = \langle e_1,\ldots,e_i\rangle\ \texttt{in}\ e]\!] \Rightarrow$$
$$\texttt{let}\ \langle x_1,\ldots,x_i\rangle = \langle \mathscr{A}[\![e_1]\!],\ldots,\mathscr{A}[\![e_i]\!]\rangle\ \texttt{in}\ \mathscr{A}[\![e]\!]$$

$$\mathscr{A}[\![\texttt{letrec}\ x_1^f = e_1^f,\ldots,x_i^f = e_i^f\ \texttt{in}\ e]\!] \Rightarrow$$
$$\texttt{letrec}\ x_1^f = \mathscr{A}[\![e_1^f]\!] \ldots x_i^f = \mathscr{A}[\![e_i^f]\!]\ \texttt{in}\ \mathscr{A}[\![e]\!]$$

$$\mathscr{A}[\![\texttt{do}\ e_1\ e_2]\!] \Rightarrow \texttt{let}\ \langle x_1,x_2\rangle = \langle \mathscr{A}[\![e_1]\!],\mathscr{A}[\![e_2]\!]\rangle\ \texttt{in}\ \texttt{do}\ x_1\ x_2$$

$$\mathscr{A}[\![\texttt{case}\ \langle e_1,\ldots,e_i\rangle\ \texttt{of}\ clauses\ \texttt{end}]\!] \Rightarrow$$
$$\texttt{let}\ \langle x_1,\ldots,x_i\rangle = \langle \mathscr{A}[\![e_1]\!],\ldots,\mathscr{A}[\![e_i]\!]\rangle$$
$$\texttt{in case}\ \langle x_1,\ldots,x_i\rangle\ \texttt{of}\ \mathscr{A}[\![clauses]\!]\ \texttt{end}$$

$$\mathscr{A}[\![clause\ clauses]\!] \Rightarrow \mathscr{A}[\![clause]\!]\ \mathscr{A}[\![clauses]\!]$$

$$\mathscr{A}[\![p\ \texttt{when}\ g\ \texttt{->}\ e]\!] \Rightarrow p\ \texttt{when}\ \mathscr{A}[\![g]\!]\ \texttt{->}\ \mathscr{A}[\![e]\!]$$

$$\mathscr{A}[\![\texttt{apply}\ e^f(e_1,\ldots,e_i)]\!] \Rightarrow$$
$$\texttt{let}\ \langle x^f,x_1,\ldots,x_i\rangle = \langle \mathscr{A}[\![e^f]\!],\mathscr{A}[\![e_1]\!],\ldots,\mathscr{A}[\![e_i]\!]\rangle$$
$$\texttt{in apply}\ x^f(x_1,\ldots,x_i)$$
$$\mathscr{A}[\![\texttt{call}\ e^m\texttt{:}e^f(e_1,\ldots,e_i)]\!] \Rightarrow$$
$$\texttt{let}\ \langle x^m,x^f,x_1,\ldots,x_i\rangle = \langle \mathscr{A}[\![e^m]\!],\mathscr{A}[\![e^f]\!],\mathscr{A}[\![e_1]\!],\ldots,\mathscr{A}[\![e_i]\!]\rangle$$
$$\texttt{in call}\ x^m\texttt{:}x^f(x_1,\ldots,x_i)$$
$$\mathscr{A}[\![\texttt{primop}\ literal[a](e_1,\ldots,e_i)]\!] \Rightarrow$$
$$\texttt{let}\ \langle x_1,\ldots,x_i\rangle = \langle \mathscr{A}[\![e_1]\!],\ldots,\mathscr{A}[\![e_i]\!]\rangle$$
$$\texttt{in primop}\ literal[a](x_1,\ldots,x_i)$$
$$\mathscr{A}[\![\texttt{try}\ e_1\ \texttt{catch}\ (x_1,x_2)\ \texttt{->}\ e_2]\!] \Rightarrow$$
$$\texttt{try}\ \mathscr{A}[\![e_1]\!]\ \texttt{catch}\ (x_1,x_2)\ \texttt{->}\ \mathscr{A}[\![e_2]\!]$$

$$\mathscr{A}[\![\texttt{receive}\ clauses\ \texttt{after}\ e_t\ \texttt{->}\ e_a]\!] \Rightarrow$$
$$\texttt{let}\ x_t = \mathscr{A}[\![e_t]\!]\ \texttt{in receive}\ \mathscr{A}[\![clauses]\!]\ \texttt{after}\ x_t\ \texttt{->}\ \mathscr{A}[\![e_a]\!]$$

Figure 4.13: Transformation of CORE ERLANG into normal form.

Mnesia is a distributed real time database implemented in ERLANG using ETS [Mattsson et al. 1999, Ericsson Utvecklings AB 2000].

The File system must be considered for the type of applications we analyse since quite often application configuration data are stored in files. If we chose to include the file system in the state we make the assumption that the files are not modified by any processes not part of the application under analysis.

4.3.2 Abstract Machine

The operational semantics is based on an abstract machine model. The abstract machine of the operational semantics has been influenced by Landin's SECD-machine [Landin 1964]. The SECD-machine was designed to provide a semantics, as well as serve as a basis of implementation, of the programming language Lisp [Allen 1978]. A configuration of the SECD-machines consists of: a value stack, an environment, a code stack and a stack of system dumps. We believe that the SECD-machine with its simple configuration and small step transformations from one configuration to the next makes it easy to understand the semantics and relate it to the informal semantics of the language being described.

Since the SECD-machine lacks any notion of input/output or persistent state (being purely functional) we have added a global state to our abstract machine in order to represent global resources. The SECD-machine operates on an "assembly" language into which the source languages' (Lisp) expressions have been compiled, but since our semantics will act as a basis for implementation of an analysis rather than of implementing the language itself, we have chosen to let our machine operate directly on CORE ERLANG-expressions. The final change is that since our machine operates on CORE ERLANG-expressions we do not need the system dump, as we will encode this information in the call stack.

Our abstract machine will be called the concrete abstract machine (CAM) to distinguish it from the abstract abstract machine (AAM) of the abstract semantics described in Chapter 5.

The CAM's configuration is a four-tuple $\langle s\ \rho\ c\ \sigma \rangle$, where:

$\boxed{s}$ — the *value stack*, which will contain the result of evaluations of elements on the code stack.

$\boxed{\rho}$ — a *context*, which keeps track of which environment the CAM is currently evaluating, as well as all the environments. Each environment is a mapping from variables to values.

$\boxed{c}$ — A sequence of CORE ERLANG expressions and intermediate expressions to evaluate; we represent this sequence as a stack. The intermediate expressions represent intermediate sub-evaluations generated by the machine during the evaluation of expressions.

$\boxed{\sigma}$ — the global state of the system. A system state contains the name of the module in which the machine is currently evaluating an expression; a queue of incoming messages; the definitions of modules; and a sequence of side effects that have occurred.

4.3.3 Dynamic Behaviour

The dynamic behaviour of the CAM is described through a set of transition rules of form:

$$(4.0) \qquad \frac{Guard}{\langle s\ \rho\ c\ \sigma \rangle \overset{Action}{\rightarrow}_{Semantics} \langle s'\ \rho'\ c'\ \sigma' \rangle}$$

where *Shorthand* = *expression*

A rule is enabled in a configuration if the configuration matches $\langle s\ \rho\ c\ \sigma \rangle$ and the *Guard* is satisfied. If the rule is applied to a configuration in which it is enabled the resulting configuration is $\langle s'\ \rho'\ c'\ \sigma' \rangle$. In order to make the rules easier to read it is possible to write *where* statements. To clarify what semantics a rule belongs to we have the *Semantics* subscript to the arrow, in the CAM's case an 'c'.

A run of the system is a sequence $\Gamma_0 \overset{\mu^1}{\rightarrow}_c \quad \overset{\mu^i}{\rightarrow}_c \Gamma_i$, where Γ_0 is an initial configuration and $\overset{\mu^i}{\rightarrow}_c$ is the i^{th} transition rule used. The observable result of a run is the sequence of *Action* labels $\mu^1, \ldots, \mu^i$ generated by the run,

The meaning of the evaluation of an expression is the set of possible sequence of *Action* labels generated by a run of the system. The special action τ label signifies an internal unobservable action which is not included in the sequence. Nondeterminism is expressed by more than one rule being enabled in a given configuration.

In the description of the semantics below, we first present the general properties of the semantics and thereafter the domains of the configuration. Then the rules, and auxiliary functions used in them, are presented.

4.4 Domains

The domains used in the concrete and abstract semantics are summarised in Figure 4.15. Each domain definition is of the form:

form $\in$ *Domain name* $\equiv$ *Domain definition*

The *Domain definition* defines the *Domain name* and the *form* shows how elements of the domain will be written in the transition rules. For example variables will be written as x, possibly with a subscript. The addition of a smallest or largest element to a domain D is denoted $D_\perp$ and $D^\top$ respectively. The smallest element is written as $\perp_D$ and the largest as $\top_D$; the domain D omitted when it is obvious from the context. The $\top$ signifies an unknown value, i.e., it can be any value in the domain. When a domain is constructed from a set of values augmented with a largest element (such as $\text{Value}^\top$), this largest value will come into play as a result of approximations in the abstract semantics and play no rôle in the concrete semantics.

The syntactic objects of expressions and variables in CORE ERLANG are represented by the syntactic domains in the semantics, the syntactic domains used are shown in Figure 4.14. The literal constants (of which atoms are

$$
\begin{array}{rcll}
x & \in & \text{Var} & \equiv \quad var \\
a & \in & \text{Atom} & \equiv \quad \{atom[a] \mid a \in atom\} \\
l & \in & \text{Literal} & \equiv \quad \{literal[l] \mid l \in literal\} \\
e & \in & \text{Exprs} & \equiv \quad exprs \\
f & \in & \text{Fun} & \equiv \quad f \\
p \texttt{ when } g \texttt{ -> } e, cl & \in & \text{Clause} & \equiv \quad clause \\
p & \in & \text{Pattern} & \equiv \quad pats \\
g & \in & \text{Guard} & \equiv \quad guard
\end{array}
$$

Figure 4.14: The syntactic domains of CORE ERLANG used in the concrete semantics

an instance) are denoted by the domains that the syntactic objects represent, i.e., the syntactic domain of *Integers* is denoted by the semantic domain of $\mathscr{Z}$. Elements of the syntactic domain are denoted as $literal[l]$ (or $atom[a]$ for atoms), where l is the written format of the literal, e.g., $literal[1]$ for the integer 1.

The other domains are:

- The Value domain consists of literal values, tuples, lists and function closures.
- The Closure domain consists of triples $\langle f, n, m \rangle$, where

  f is a syntactic object in the Fun domain,

  n is an environment name and

  m is a module name.

  The environment name is the name of the environment that provides bindings of the free variables occurring in f and the module name is the module in which the closure was defined in order that local function calls can be resolved.

- The intermediate expressions (AuxExpr) domain consists of an operator and a sequence of arguments. The arguments can either be values, expressions, environment names, clause lists, code stacks, or input queues. The expressions are:

  `def` adds bindings from its arguments by constructing a new environment, which is added to the context.

  `restore` restores a previous environment as the current.

  `if` switches on a boolean value on the value stack.

  `return` returns from a function call.

  `exception` signifies that an exception has occurred.

  `catch` catches exceptions.

  `rec` traverses the input queue of messages.

  `message` generates a `receive` action label.

  `approx` indicates that a function call has been approximated.

- The Module domain consists of maps from atoms and integers to closures, where the atom is a function name and the integer its arity. In the closure the environment name is that of the special empty environment called 0.
- The Modules domain consists of maps from module names to modules, where modules names are atoms. This domain intuitively represents the modules of code loaded into the ERLANG node.
- The Queue domain consists of lists of values.
- The Effects domain is a list of key/value pairs $\langle k, v \rangle$ where the pair to the front of the list is the latest. Each key/value pair represents a side effect which affects the state of a global resource.
- The domain of environments (Env) consists of maps from variables to values.
- The domain of contexts (Context) consists of triples $\langle n, env, k \rangle$, where n and k are environment names and *env* is a map from environment names to environments. The environment name n is the name of the current environment, i.e., the environment in which the abstract machine evaluates at the moment. The environment k is the name of the last allocated environment name, providing a way of allocating new unique environment names since the environments names are natural numbers.

We represent stacks using the list syntax of ERLANG.

A machine configuration is either `NIL`, which signifies that the machine has stopped, or a four-tuple consisting of the following parts:

Stack — a stack of values that are results of evaluating expressions.

Context — consists of three parts: the first part is the name of the current environment; the second part is a mapping of environment names to environments; and the third part is the last environment name allocated. By convention, the second part of the context should map the environment name 0 to the empty environment.

$$
\begin{array}{rcll}
\Gamma, \langle s\ \rho\ c\ \sigma \rangle & \in & \text{Configuration} & \equiv (\text{Stack} \quad \text{Context} \quad \text{Code} \quad \text{State}) \cup \{\texttt{NIL}\} \\
[v \mid s] & \in & \text{Stack} & \equiv \{[\,]\} \cup (\text{Value}^{\top} \quad \text{Stack}) \\
\rho & \in & \text{Context} & \equiv \text{EnvName} \quad \text{EnvMap} \quad \text{EnvName} \\
[[\![e]\!], op(\ldots) \mid c] & \in & \text{Code} & \equiv \{[\,]\} \cup ((\text{Exprs} \cup \text{AuxExpr}) \quad \text{Code}) \\
\sigma & \in & \text{State} & \equiv \text{ModuleName} \quad \text{Queue}^{\top} \quad \text{Modules} \quad \text{Effects} \\
 & & \text{EnvMap} & \equiv \text{EnvName} \to \text{Env} \\
 & & \text{Env} & \equiv \text{Var} \to \text{Value}^{\top} \\
 & & \text{EnvName} & \equiv \mathscr{N} \\
\langle s, n, m \rangle & \in & \text{Save} & \equiv \text{Stack} \quad \text{EnvName} \quad \text{ModuleName} \\
v & \in & \text{Value} & \equiv \text{Literal} \cup \text{Tuple} \cup \text{List} \cup \text{Closure} \\
\{v_1, \ldots, v_i\} & \in & \text{Tuple} & \equiv \{\ \{\}\ \} \cup (\text{Value}^{\top} \quad \text{Tuple}) \\
[v_1 | v_2] & \in & \text{List} & \equiv \{[\,]\} \cup (\text{Value}^{\top} \quad \text{Value}^{\top}) \\
\langle v_1, \ldots, v_i \rangle & \in & \text{Values} & \equiv \{\langle\,\rangle\} \cup (\text{Value}^{\top} \quad \text{Values}) \\
\langle\langle f, n, M \rangle\rangle & \in & \text{Closure} & \equiv \text{Fun} \quad \text{EnvName} \quad \text{ModuleName} \\
op(o_1, \ldots, o_i) & \in & \text{AuxExpr} & \equiv \text{AuxOp} \quad \text{Args} \\
 & & \text{AuxOp} & \equiv \{\mathsf{def}, \mathsf{restore}, \mathsf{if}, \mathsf{return}, \mathsf{exception}, \mathsf{catch}, \mathsf{rec}, \mathsf{message}, \mathsf{approx}\} \\
(o_1, \ldots, o_i) & \in & \text{Args} & \equiv \{()\} \cup (\text{Arg} \quad \text{Args}) \\
 & & \text{Arg} & \equiv \text{Value}^{\top} \cup \text{Exprs} \cup \text{EnvName} \cup \text{ClauseList} \cup \text{Code} \cup \text{Queue}^{\top} \\
[cl \mid cls] & \in & \text{ClauseList} & \equiv \text{Clause} \quad \text{ClauseList} \\
[v \mid q] & \in & \text{Queue} & \equiv \{[\,]\} \cup (\text{Value} \quad \text{Queue}) \\
ms & \in & \text{Modules} & \equiv \text{ModuleName} \to \text{Module} \\
m & \in & \text{ModuleName} & \equiv \text{Atom} \\
 & & \text{Module} & \equiv (\text{Atom} \quad \mathscr{N}) \to \text{Closure} \\
[\langle v_k, v_v \rangle \mid \pi] & \in & \text{Effects} & \equiv \{[\,]\} \cup ((\text{Value}^{\top} \quad \text{Value}^{\top}) \quad \text{Effects})
\end{array}
$$

Figure 4.15: The domains of the concrete and abstract semantics

Code — a sequence of CORE ERLANG and intermediate expressions to be evaluated. The sequence is represented as a stack since we pop the expression to be evaluated, and then during the evaluation we push more expressions to be evaluated onto the stack. The first element of the expression sequence may be the approximation ($\mathsf{approx}()$) in the abstract semantics, when approximating a function call.

State — the global state of the system consists of four parts: the name of the CORE ERLANG module in which the machine is currently evaluating; the input queue of messages; the mapping of module names to the modules definitions; a sequence of side effects that have occurred.

The action labels used in the semantics are:

`exit`(r) which signifies that the process has terminated with reason r which is either `fail` or `normal`.

m: $f(v_1,\ldots,v_i)$ which signifies that function f in module m has been called with arguments $v_1,\ldots,v_i$.

`primop` $f(v_1,\ldots,v_i)$ which signifies that the primitive operation f has been called with arguments $v_1,\ldots,v_i$.

`receive` v which signifies that message v has been removed from the input queue by a `receive` statement.

`timeout` which signifies that a timeout has occurred in a `receive` statement.

$\top$ which signifies that an approximation has been made. This label will only occur in the abstract semantics.

4.5 Auxiliary Functions

These auxiliary functions will be used in both the concrete and abstract semantics and consequently some of them have special cases for approximations that will never be used in the concrete semantics. E.g., the match function must be defined for the case where the value (or a part of a compound value) is an approximation.

The following functions all use or manipulate the context part of the machine configuration. The `set` function sets the current environment. The `current` function returns the name of the current environment. The `update` function adds a new environment to the context and sets the new environment to be the current environment. When ρ is a context, we use $\rho(x)$ as a shorthand for $(env(\mathsf{current}(\rho)))(x)$.

$$
\begin{aligned}
&\mathsf{set}\colon (\mathrm{EnvName} \quad \mathrm{Context}) \to \mathrm{Env} \\
&\mathsf{set}\,(n, \langle n', env, k \rangle) \quad = \quad \langle n, env, k \rangle
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{current}\colon \mathrm{Context} \to \mathrm{EnvName} \\
&\mathsf{current}\,(\langle n, env, k \rangle) \quad = \quad n
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{update}\colon \mathrm{Env} \quad \mathrm{Context} \to \mathrm{Context} \\
&\mathsf{update}\,(map, \langle n, env, k \rangle) \quad = \\
&\quad \langle k+1, env \cup \{k+1 \mapsto (env(n) \oplus map)\}, k+1 \rangle
\end{aligned}
$$

Both the `bind` and `bindf` functions add a new environment to the context and sets the new environment to be the current environment. The `bindf` functions create closures that refer to the environment being defined in order to enable mutual recursion between functions.

$$
\begin{aligned}
&\mathsf{bind}\colon (\mathrm{Vars} \quad \mathrm{Values} \quad \mathrm{Context}) \to \mathrm{Context} \\
&\mathsf{bind}\,(\langle x_{f1}, \ldots, x_{fn} \rangle, \langle v_1, \ldots, v_n \rangle, \rho) = \\
&\quad \mathsf{update}\,(\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}, \rho)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{bindf}\colon (\mathrm{Vars} \quad \mathrm{Values} \quad \mathrm{Context} \quad \mathrm{ModuleName}) \to \mathrm{Context} \\
&\mathsf{bindf}\,(\langle x_{f1}, \ldots, x_{fn} \rangle, \langle f_1, \ldots, f_n \rangle, \langle n, env, k \rangle, m) = \\
&\quad \mathsf{update}(\{x_{f1} \mapsto \langle\langle f_1, k+1, m \rangle\rangle, \ldots, x_{fn} \mapsto \langle\langle f_n, k+1, m \rangle\rangle\}, \\
&\qquad \langle n, env, k \rangle)
\end{aligned}
$$

The functions module, queue, and effects are selectors on the global state of the configuration. The func function performs a lookup for a particular function in the module definitions[1]. We will for simplicity assume that all called functions are defined. The set_module function changes the module in the configuration.

[1] If we were to model ERLANG's exact behaviour we would have to introduce a special case when no function matches, and return a predefined library function that handles this. The action to be taken when an undefined function is called can defined by the user.

$$\mathsf{module}: \mathrm{State} \to \mathrm{ModuleName}$$
$$\mathsf{module}\,(\langle m, q, ms, \pi \rangle) = m$$

$$\mathsf{queue}: \mathrm{State} \to \mathrm{Queue}$$
$$\mathsf{queue}\,(\langle m, q, ms, \pi \rangle) = q$$

$$\mathsf{effects}: \mathrm{State} \to \mathrm{Effects}$$
$$\mathsf{effects}\,(\langle m, q, ms, \pi \rangle) = \pi$$

$$\mathsf{func}: (\mathrm{ModuleName} \quad \mathrm{Atom} \quad \mathscr{N} \quad \mathrm{State}) \to \mathrm{Closure}$$
$$\mathsf{func}\,(m', f, i, \langle m, q, ms, \pi \rangle) = (ms\,(m'))\,(f, i)$$

$$\mathsf{set\_module}: (\mathrm{ModuleName} \quad \mathrm{State}) \to \mathrm{State}$$
$$\mathsf{set\_module}\,(m', \langle m, q, ms, \pi \rangle) = \langle m', q, ms, \pi \rangle$$

The `deliver` function places a value at the end of the input queue, unless the queue is an approximation in which case it returns the state unchanged. The `remove` function removes an element from the input queue in the configuration. The `delete` function removes an element from a queue, unless the queue is an approximation in which case the approximation is returned.

$$\mathsf{deliver}: \mathrm{Configuration} \to \mathrm{Configuration}$$
$$\mathsf{deliver}\,(\langle m, q, ms, \pi \rangle) =$$
$$\begin{cases} \langle m, [v_1, \ldots, v_i, v], ms, \pi \rangle & \textbf{if } q = [v_1, \ldots, v_i] \\ \langle m, q, ms, \pi \rangle & \textbf{otherwise} \end{cases}$$

$$\mathsf{remove}: (\mathrm{Value} \quad \mathrm{Configuration}) \to \mathrm{Configuration}$$
$$\mathsf{remove}\,(v, \langle v, q, ms, \pi \rangle) = \langle v, \mathsf{delete}\,(v, q), ms, \pi \rangle$$

$$\mathsf{delete}: (\mathrm{Value} \quad \mathrm{Queue}^{\top}) \to \mathrm{Queue}^{\top}$$
$$\mathsf{delete}\,(v, q) = \begin{cases} \top & \textbf{if } q = \top \\ q' & \textbf{if } q = [v' \mid q'] \wedge v' = v \\ [v' \mid \mathtt{remove}\,(v, q')] & \textbf{if } q = [v' \mid q'] \wedge v' \neq v \end{cases}$$

The `publish` and `read` functions deal with the side effects. All side effects that change the global state are modelled using `publish`, and all calls examining the global state are modelled using `read`. The `publish` function adds a key/value pair to the list of side effects, whereas the `read` function looks up the associated value of a particular key. To be able to consider the order of side effects (e.g., when one side effects cancel an earlier effect) the `read` function takes a second key that may not occur before the first key in order for the lookup to succeed. After a key that contains an approximation has been

published the publish function becomes an identity function and all calls to read return $\top$.

$$
\begin{array}{l}
\mathsf{publish}\colon (\mathrm{Value}^{\top} \quad \mathrm{Value}^{\top} \quad \mathrm{Configuration}) \to \mathrm{Configuration} \\
\mathsf{publish}\,(k, v, \langle m, q, ms, \pi \rangle) = \\
\quad \begin{cases}
\langle m, q, ms, \pi \rangle & \textbf{if}\, \pi = [\langle k_1, v_1 \rangle \mid \pi'] \wedge \mathsf{contain\_top}(k_1) \\
\langle m, q, ms, [\langle k, v \rangle \mid \pi] \rangle & \textbf{otherwise}
\end{cases} \\
\mathsf{read}\colon (\mathrm{Value} \quad \mathrm{Value} \quad \mathrm{Effects}) \to \mathrm{Value}^{\top} \\
\mathsf{read}\,(k_1, k_2, \pi) = \\
\quad \begin{cases}
\top & \textbf{if}\, \mathsf{contain\_top}(k_1) \vee \mathsf{contain\_top}(k_2) \\
\mathtt{fail} & \textbf{if}\, \pi = [\,] \\
\top & \textbf{if}\, \pi = [\langle k_3, v \rangle \mid \pi] \wedge \mathsf{contain\_top}(k_3) \\
\mathtt{fail} & \textbf{if}\, \pi = [\langle k_2, v \rangle \mid \pi'] \\
\{\mathtt{ok}, v\} & \textbf{if}\, \pi = [\langle k_1, v \rangle \mid \pi'] \\
\mathsf{read}\,(k_1, k_2, \pi') & \textbf{if}\, \pi = [\langle k_3, v \rangle \mid \pi'] \wedge \\
 & \qquad k_1 \neq k_3 \wedge k_2 \neq k_3 \ \wedge \neg \mathsf{contain\_top}(k_3)
\end{cases}
\end{array}
$$

The `match` function takes a pattern and a value and tries to match them. If successful the result is an environment containing the bindings generated by the match. If unsuccessful the result is an enviroment that contains the $\bot$ element. Since $\top$ will match any value, a match against a value containing $\top$ a successful match will not be conclusive but an unsuccessful will be conclusive. This means that when we get an inconclusive match we only know that the match may succeed.

There are two important aspects to note: First is that the match in CORE ERLANG is linear as opposed to ERLANG. Secondly, that in the concrete semantics the approximation $\top$ will never occur, it is included in the definition because it will be needed for the abstract semantics.

$$
\begin{array}{l}
\mathsf{match}\colon (\text{Pattern} \quad \text{Value}^{\top}) \to \text{Env}_{\perp} \\
\mathsf{match}\,(p,v) = \\
\left\{ \begin{array}{ll}
\{x \mapsto v\} & \textbf{if } p = [\![x]\!] \\
\{x \mapsto v\} \cup \mathsf{match}\,(p_1,v) & \textbf{if } p = [\![x = p_1]\!] \\
\{\} & \textbf{if } p = [\![\mathit{literal}[v_1]]\!] \wedge \\
 & \quad (v_1 = v \vee v = \top) \\
\mathsf{match}\,(p_1,v_1) \cup \mathsf{match}\,(p_2,v_2) & \textbf{if } p = [\![[p_1 \mid p_2]]\!] \wedge \\
 & \quad v = [v_1 \mid v_2] \\
\mathsf{match}\,(p_1,\top) \cup \mathsf{match}\,(p_2,\top) & \textbf{if } p = [\![[p_1 \mid p_2]]\!] \wedge v = \top \\
\mathsf{match}\,(p_1,v_1) \cup \quad \cup \mathsf{match}\,(p_i,v_i) & \textbf{if } p = [\![\{p_1,\ldots,p_i\}]\!] \wedge \\
 & \quad v = \{v_1,\ldots,v_i\} \\
\mathsf{match}\,(p_1,\top) \cup \quad \cup \mathsf{match}\,(p_i,\top) & \textbf{if } p = [\![\{p_1,\ldots,p_i\}]\!] \wedge \\
 & \quad v = \top \\
\mathsf{match}\,(p_1,v_1) \cup \quad \cup \mathsf{match}\,(p_i,v_i) & \textbf{if } p = [\![\langle p_1,\ldots,p_i\rangle]\!] \wedge \\
 & \quad v = \langle v_1,\ldots,v_i\rangle \\
\mathsf{match}\,(p_1,\top) \cup \quad \cup \mathsf{match}\,(p_i,\top) & \textbf{if } p = [\![\langle p_1,\ldots,p_i\rangle]\!] \wedge \\
 & \quad v = \top \\
\{\perp\} & \textbf{otherwise}
\end{array} \right.
\end{array}
$$

The contain_top function examines its argument to see if it is $\top$ or contains $\top$ if it is a compound value.

$$
\begin{array}{l}
\mathsf{contain\_top}\colon \text{Value}^{\top} \to \{\texttt{true},\texttt{false}\} \\
\mathsf{contain\_top}\,(v) = \\
\left\{ \begin{array}{ll}
\texttt{true} & \textbf{if } v = \top \\
\texttt{false} & \textbf{if } v = \mathit{literal}[l] \\
\mathsf{contain\_top}\,(v_1) \vee \mathsf{contain\_top}\,(v_2) & \textbf{if } v = [v_1 \mid v_2] \\
\mathsf{contain\_top}\,(v_1) \vee \quad \vee \mathsf{contain\_top}\,(v_i) & \textbf{if } v = \{v_1,\ldots,v_i\} \\
\mathsf{contain\_top}\,(v_1) \vee \quad \vee \mathsf{contain\_top}\,(v_i) & \textbf{if } v = \langle v_1,\ldots,v_i\rangle \\
\texttt{false} & \textbf{if } v = \langle\langle f,n,m\rangle\rangle
\end{array} \right.
\end{array}
$$

4.6 Transition Rules

4.6.1 Normal Termination

An ERLANG process can either terminate normally or abnormally, as described in Section 2.3.2. This rule states that the process terminates normally,

if the code stack is empty. The rule results in the `NIL` configuration, issuing the `exit(normal)` action.

$$\langle\, s\ \rho\ [\,]\ \sigma\,\rangle \xrightarrow{\texttt{exit(normal)}}_{c} \texttt{NIL} \tag{4.1}$$

4.6.2 Variables and Literals

When evaluating a variable, the value to which the variable is bound is placed on the value stack. A literal that is evaluated is simply placed directly on the value stack.

$$\langle\, s\ \rho\ [\,[\![x]\!]\mid c\,]\ \sigma\,\rangle \xrightarrow{\tau}_{c} \langle\, [\,\rho(x)\mid s\,]\ \rho\ c\ \sigma\,\rangle \tag{4.2}$$

$$\langle\, s\ \rho\ [\,[\![\mathit{literal}[l]]\!]\mid c\,]\ \sigma\,\rangle \xrightarrow{\tau}_{c} \langle\, [\,l\mid s\,]\ \rho\ c\ \sigma\,\rangle \tag{4.3}$$

4.6.3 Compound Expressions

Compound expressions are evaluated by placing the compound data structure on the value stack, with the variables replaced by their values. Note that since we have the "A-normal" form all subexpressions are variables.

$$\langle\, s\ \rho\ [\,[\![\,\{x_1,\ldots,x_i\}\,]\!]\mid c\,]\ \sigma\,\rangle \xrightarrow{\tau}_{c} \langle\, [\,\{\rho(x_1),\ldots,\rho(x_i)\}\mid s\,]\ \rho\ c\ \sigma\,\rangle \tag{4.4}$$

$$\langle\, s\ \rho\ [\,[\![\,[x_1\mid x_2]\,]\!]\mid c\,]\ \sigma\,\rangle \xrightarrow{\tau}_{c} \langle\, [\,[\rho(x_1)\mid\rho(x_2)]\mid s\,]\ \rho\ c\ \sigma\,\rangle \tag{4.5}$$

$$\langle\, s\ \rho\ [\,[\![\,\langle x_1,\ldots,x_i\rangle\,]\!]\mid c\,]\ \sigma\,\rangle \xrightarrow{\tau}_{c} \langle\, [\,\langle\rho(x_1),\ldots,\rho(x_i)\rangle\mid s\,]\ \rho\ c\ \sigma\,\rangle \tag{4.6}$$

4.6.4 Funs

The evaluation of a *fun* results in a closure, consisting of the function definition, the current environment and module, being placed on the value stack.

$$\langle\, s\ \rho\ [\,[\![\texttt{fun}(x_1,\ldots,x_i) = e]\!]\mid c\,]\ \sigma\,\rangle \xrightarrow{\tau}_C \langle\, [\,v\mid s\,]\ \rho\ c\ \sigma\,\rangle \tag{4.7}$$

$$\textbf{where}\ n = \mathsf{current}(\rho)$$
$$m = \mathsf{module}(\sigma)$$
$$v = \langle\langle [\![\texttt{fun}(x_1,\ldots,x_n) = e]\!], n, m\rangle\rangle$$

4.6.5 Binding Expressions

A `let` expression is evaluated by transforming it into a sequence of expressions and intermediate expressions placed on the code stack. This sequence contains first the expressions to which the variables are to be bound, followed by the intermediate expression `def` that binds the variables to the values of these expressions. Thereafter follows the body e of the `let` expression, followed by the intermediate expression `restore` which restores the current environment to its value before the evaluation of the `let` expression.

$$\begin{array}{l}\langle\, s\ \rho\ [\,[\![\texttt{let}\ \langle x_1,\ldots,x_i\rangle = \langle e_1,\ldots,e_i\rangle\ \texttt{in}\ e]\!]\mid c\,]\ \sigma\,\rangle \xrightarrow{\tau}_C \\ \langle\, s\ \rho\ [\,e_1,\ldots,e_i,\mathsf{def}(x_1,\ldots,x_i),e,\mathsf{restore}(n)\mid c\,]\ \sigma\,\rangle \\ \textbf{where}\ n = \mathsf{current}(\rho)\end{array} \tag{4.8}$$

The `def` intermediate, which takes the sequence of variables to be bound as argument, binds its arguments to the i first values on the value stack.

$$\langle\, [\,v_i,\ldots,v_1\mid s\,]\ \rho\ [\,\mathsf{def}(x_1,\ldots,x_i)\mid c\,]\ \sigma\,\rangle \xrightarrow{\tau}_C \langle\, s\ \rho'\ c\ \sigma\,\rangle \tag{4.9}$$

$$\textbf{where}\ \rho' = \mathsf{bind}(\langle x_1,\ldots,x_i\rangle, \langle v_1,\ldots,v_i\rangle, \rho)$$

The `restore` intermediate takes the name of an environment as argument and sets the current environment to the argument environment name. It is used to restore a previous environment, thus "forgetting" later bindings.

$$\langle\, s\ \rho\ [\,\mathsf{restore}(n)\mid c\,]\ \sigma\,\rangle \xrightarrow{\tau}_C \langle\, s\ \rho'\ c\ \sigma\,\rangle \tag{4.10}$$

$$\textbf{where}\ \rho' = \mathsf{set}(n,\rho)$$

A `letrec` expressions is similar to a `let` expression, but takes funs as arguments. It is evaluated by creating a new environment where the variables are

bound to the closures of the *fun*s, and replacing the `letrec` expression by its body e followed by the `restore`.

$$
\begin{aligned}
(4.11) \qquad & \langle\, s\ \rho\ [\,[\![\, \texttt{letrec}\ x_1^f = e_1^f\ \ldots\ x_i^f = e_i^f\ \texttt{in}\ e \,]\!] \mid c\,]\ \sigma\,\rangle \xrightarrow{\tau}_c \\
& \langle\, s\ \rho'\ [\,e, \mathsf{restore}(n) \mid c\,]\ \sigma\,\rangle \\
& \textbf{where}\ n = \mathsf{current}(\rho) \\
& \qquad\quad m = \mathsf{module}(\sigma) \\
& \qquad\quad \rho' = \mathsf{bindf}\left(\langle x_1^f, \ldots, x_i^f\rangle, \langle e_1^f, \ldots, e_i^f\rangle, \rho, m\right)
\end{aligned}
$$

4.6.6 Sequencing Expressions

The sequencing expressions simply places the value, to which the second argument x_2 is bound, on the value stack. The value to which the first variable x_1 is bound is simply ignored.

$$
(4.12) \qquad \langle\, s\ \rho\ [\,[\![\, \texttt{do}\ x_1\ x_2 \,]\!] \mid c\,]\ \sigma\,\rangle \xrightarrow{\tau}_c \langle\, [\,\rho(x_2) \mid s\,]\ \rho\ c\ \sigma\,\rangle
$$

4.6.7 Conditional Expressions

The conditional `case` expression is somewhat complex to handle since the choice of clause depends both on that the pattern match and that the guard evaluates to `'true'`. Guards are handled using an intermediate `if` expression.

The first `case` rule (4.13), deals with the case when the first clause of the `case` expression matches the value sequence to which the variable sequence $\langle x_1, \ldots, x_i \rangle$ is bound (i.e, $\perp$ is not a member in the environment returned by the match). The match results in a new context. The `case` expression is replaced on the code stack by the guard g, followed by an intermediate `if` expression. The true branch of the `if` intermediate contains the body e of the clause followed by a `restore` intermediate to restore the environment to that prior to evaluation of the `case` expression. The false branch consists of the `restore` intermediate followed by the `case` expression without the first clause.

(4.13)
$$\frac{\bot \notin \mathsf{match}(p, \langle \rho(x_1), \ldots, \rho(x_i) \rangle)}{\begin{array}{l}\langle s\ \rho\ [[\![\texttt{case}\ \langle x_1, \ldots, x_i \rangle\ \texttt{of}\ \mathit{clause}\ \mathit{clauses}\ \texttt{end}]\!] \mid c]\ \sigma \rangle \xrightarrow{\tau}_c \\ \langle s\ \rho'\ [g, \mathsf{if}(c_1, c_2) \mid c]\ \sigma \rangle \\ \textbf{where}\ [\![p\ \texttt{when}\ g\ \texttt{->}\ e]\!] = \mathit{clause} \\ \qquad \rho' = \mathsf{update}(\mathsf{match}(p, \langle \rho(x_1), \ldots, \rho(x_i) \rangle), \rho) \\ \qquad n = \mathsf{current}(\rho) \\ \qquad c_1 = [e, \mathsf{restore}(n)] \\ \qquad c_2 = [\mathsf{restore}(n), [\![\texttt{case}\ \langle x_1, \ldots, x_i \rangle\ \texttt{of}\ \mathit{clauses}\ \texttt{end}]\!]]\end{array}}$$

The second `case` rule (4.14), deals with the case where the first clause does not match the value sequence to which the variable sequence $\langle x_1, \ldots, x_i \rangle$ is bound. The result is the evaluation of `case` expression without the first clause. The case where no more clauses remain need not be handled, since in a well formed `case` expression the patterns of the clauses must be exhaustive. This can easily be achieved by adding a last clause of form: $\langle x_{i+1} \rangle$ `when 'true' ->` e_{i+1}, where e_{i+1} could for example raise an exception.

(4.14)
$$\frac{\bot \in \mathsf{match}(p, \langle \rho(x_1), \ldots, \rho(x_i) \rangle)}{\begin{array}{l}\langle s\ \rho\ [[\![\texttt{case}\ \langle x_1, \ldots, x_i \rangle\ \texttt{of}\ \mathit{clause}\ \mathit{clauses}\ \texttt{end}]\!] \mid c]\ \sigma \rangle \xrightarrow{\tau}_c \\ \langle s\ \rho\ [[\![\texttt{case}\ \langle x_1, \ldots, x_i \rangle\ \texttt{of}\ \mathit{clauses}\ \texttt{end}]\!] \mid c]\ \sigma \rangle \\ \textbf{where}\ [\![p\ \texttt{when}\ g\ \texttt{->}\ e]\!] = \mathit{clause}\end{array}}$$

The `if` intermediate is evaluated by replacing the `if` intermediate either with its 1st or 2nd argument on the code stack. The top element is popped from the value stack and if it is `true` then the 1st argument replaces the `if`, and if it is `false` then the 2nd argument replaces the `if`.

(4.15)
$$\frac{}{\begin{array}{l}\langle [\texttt{'true'} \mid s]\ \rho\ [\mathsf{if}([c_1, \ldots, c_i], c') \mid c]\ \sigma \rangle \xrightarrow{\tau}_c \\ \langle s\ \rho\ [c_1, \ldots, c_i \mid c]\ \sigma \rangle\end{array}}$$

(4.16)
$$\frac{}{\begin{array}{l}\langle [\texttt{'false'} \mid s]\ \rho\ [\mathsf{if}(c', [c_1, \ldots, c_i]) \mid c]\ \sigma \rangle \xrightarrow{\tau}_c \\ \langle s\ \rho\ [c_1, \ldots, c_i \mid c]\ \sigma \rangle\end{array}}$$

4.6.8 Function Expressions

There are three different types of functions expressions: `apply` expressions are local function calls within the same module, `call` expressions which are

calls to functions that may be defined in another module, `primop` calls which are typically generated by the compiler to handle implementation dependent aspects. We will also use the `primop` calls when modelling exceptions and side effects.

Evaluation of a function expression generates a visible action. The `apply` only constitutes a special case of the more general `call`. An intermediate expression return is used to restore the value stack, environment and current module to that prior to the evaluation of the function expression.

4.6.8.1 Apply

The first `apply` rule (4.17) deals with case where the function name variable x^f is bound to an atom. The defining closure $\langle\langle [\![\texttt{fun}(x_1,\ldots,x_i) = e]\!], 0, m \rangle\rangle$ of the function is looked up in the current module. The 0 environment name in the closure indicates that the empty environment should be used. The `apply` is transformed into the body e of the closure followed by the return intermediate that takes three parameters, the previous stack s, the current environment name n and the module name m. In the new configuration in which the body will be evaluated, the value stack is empty and an environment (where the formal parameters $x'_1,\ldots,x'_i$ are bound to the values of the actual parameters $x_1,\ldots,x_i$) is added to the context.

The second `apply` rule (4.18) deals with case where the function name variable x^f is bound to a closure $\langle\langle [\![\texttt{fun}(x_1,\ldots,x_i) = e]\!], n', m \rangle\rangle$. The rule is the same as the first `apply` rule with the important exception that the binding of the formal parameters to the actual parameters should be added to the environment n' given by the closure, as opposed to the special empty environment of rule (4.17).

$$
\text{(4.17)}\quad
\frac{\rho(x^f) \in \text{Atom}}{
\begin{array}{l}
\langle s\ \rho\ [[\![\texttt{apply}\ x^f\ (x_1,\ldots,x_i)]\!] \mid c]\ \sigma \rangle \xrightarrow{\mu}_c \\
\langle [\,]\ \rho'\ [e, \mathsf{return}(\mathsf{s},\mathsf{n},\mathsf{m}) \mid c]\ \sigma \rangle \\
\textbf{where}\ n = \mathsf{current}(\rho) \\
\qquad m = \mathsf{module}(\sigma) \\
\qquad \langle\langle [\![\texttt{fun}(x'_1,\ldots,x'_i) = e]\!], 0, m \rangle\rangle = \mathsf{func}\left(m, \rho(x^f), i, \sigma\right) \\
\qquad \rho' = \mathsf{bind}(\langle x'_1,\ldots,x'_i \rangle, \langle \rho(x_1),\ldots,\rho(x_i) \rangle, \mathsf{set}(0,\rho)) \\
\qquad \mu = m : \rho(x^f)\,(\rho(x_1),\ldots,\rho(x_i))
\end{array}
}
$$

$$(4.18)\quad \frac{\rho(x^f) = \langle\langle [\![\texttt{fun}(x'_1,\ldots,x'_i) = e]\!], n', m \rangle\rangle}{\begin{array}{l}\langle s\ \rho\ [[\![\texttt{apply}\ x^f\ (x_1,\ldots,x_i)]\!] \mid c]\ \sigma\rangle \xrightarrow{\mu}_c \\ \langle [\,]\ \rho'\ [e, \mathsf{return}(\mathsf{s},\mathsf{n},\mathsf{m}) \mid c]\ \sigma\rangle \\ \textbf{where}\ n = \mathsf{current}(\rho) \\ \qquad m = \mathsf{module}(\sigma) \\ \qquad \rho' = \mathsf{bind}(\langle x'_1,\ldots,x'_i\rangle, \langle \rho(x_1),\ldots,\rho(x_i)\rangle, \mathsf{set}(n',\rho)) \\ \qquad \mu = m : \rho(x^f)\,(\rho(x_1),\ldots,\rho(x_i))\end{array}}$$

The third `apply` rule (4.19), deals with case where the function name variable x^f is not bound to an atom or a closure. In this case an exception is raised by transforming the `apply` into a `primop` call which raises an exception, in a new context where the argument variable is bound to `badarg`. We have to insert a return intermediate to restore the environment.

The condition in the guard of the rule that x^f should not be bound to $\top_{\text{Value}}$, is only relevant in the abstract semantics since it can never be bound to $\top_{\text{Value}}$ in the concrete semantics. The reason we include the condition in the concrete semantics is that then we can used the same rule in the abstract semantics.

$$(4.19)\quad \frac{\rho(x^f) \notin \text{Atom} \cup \text{Closure} \cup \{\top_{\text{Value}}\}}{\begin{array}{l}\langle s\ \rho\ [[\![\texttt{apply}\ x^f\ (x_1,\ldots,x_i)]\!] \mid c]\ \sigma\rangle \xrightarrow{\mu}_c \\ \langle [\,]\ \rho'\ [[\![\texttt{primop}\ atom[\texttt{'raise'}](x')]\!], \mathsf{return}(s,n,m) \mid c]\ \sigma\rangle \\ \textbf{where}\ n = \mathsf{current}(\rho) \\ \qquad m = \mathsf{module}(\sigma) \\ \qquad \rho' = \mathsf{bind}(\langle x'\rangle, \langle \texttt{'badarg'}\rangle, \rho) \\ \qquad \mu = m : \rho(x^f)\,(\rho(x_1),\ldots,\rho(x_i))\end{array}}$$

The return has three parameters the previous stack s, the current environment name n and the module name m. Before the evaluation of the return intermediate, the value stack should only contain one value. The value on the stack is the return value of the function from which we are currently returning. This value should be placed on top of the restored stack. The current environment name and module name are restored to those retrieved from the save.

$$(4.20)\quad \langle [v]\ \rho\ [\mathsf{return}(s,n,m) \mid c]\ \sigma\rangle \xrightarrow{\tau}_c \langle [v \mid s]\ \rho'\ c\ \sigma'\rangle$$

$$\textbf{where}\ \rho' = \mathsf{set}(n,\rho)$$
$$\sigma' = \mathsf{set\_module}(m,\sigma)$$

4.6.8.2 Call

For the `call` expressions there are a number of the built in functions that have to be handled in a special way, first are those that can only be modelled as special `call` expression rules. These calls we will refer to as the domain SpecialCall which contain triplets $\langle m, f, i \rangle$ of module name m, function name f and function arity i. Among these functions we present example rules for the `erlang:apply/3` and `erlang:throw/1` functions, but there are several more, they are all listed in Table 4.1.

Another category of built in functions are those that contain side effects, we model them by writing special abstract version CORE ERLANG modules using the `primop` calls `'read'` and `'publish'`. An example of a built in function modelled in CORE ERLANG is the `erlang:unregister/1` function which is depicted in Figure 4.16, the registering of processes is described in Section 2.3.2. The modules that contain functions that need to be modelled in this ways are split into two tables: first is Table 4.2 where is listed all the modules for which we have an abstract version, the second Table 4.3 containing all the applications that have modules we should write an abstract version for but not had the time to write.

```
'unregister'/1 =
  fun(Name) ->
    case primop 'read' ({'register', Name},
                        {'unregister', Name})
    of
      <{'ok', _cor0}> when
                      call 'erlang':'pid'(_cor0) ->
        primop 'publish' ({'unregister', Name},
                          'false')
      <fail> when 'true' ->
        primop 'raise' ('badarg')
    end
```

Figure 4.16: Model of `erlang:unregister/3` in CORE ERLANG.

The first and second `call` expression rules, deal with a call to the built in function `erlang:apply`. The first rule is the case where the arguments to the call have the correct type and the second is the case where the types are incorrect and a `badarg` exception is raised.

The first rule (4.21) transforms the `call` expression into a `call` expression of the arguments where the fresh variables $x'_1, \ldots, x'_i$ are bound to the elements of third argument's value. A return intermediate is inserted to restore the environment after the evaluation of the call.

The second `call` rule (4.22) transforms the `call` expression into a `primop` expression raising an exception in a new context where the

Tests:

`is_atom/1`	`is_binary/1`	`is_float/1`
`is_fun/1`	`is_integer/1`	`is_list/1`
`is_pid/1`	`is_port/1`	`is_ref/1`
`is_tuple/1`		

Conversions:

`atom_to_list/1`	`binary_to_float/1`	`binary_to_list/1`
`binary_to_list/3`	`binary_to_term`	`float/1`
`float_to_list/1`	`fun_to_list/1`	`integer_to_list/1`
`list_to_atom/1`	`list_to_binary/1`	`list_to_float/1`
`list_to_integer/1`	`list_to_pid/1`	`list_to_tuple/1`
`pid_to_list/1`	`port_to_list/1`	`ref_to_list/1`
`term_to_binary/1`	`term_to_binary/2`	`tuple_to_list/1`

Other:

`apply/2`	`apply/3`	`exit/1`
`exit/2`	`fault/1`	`fault/2`
`halt/0`	`halt/1`	`throw/1`

Table 4.1: *Functions in Module Erlang for which we have to have special transition rules.*

application	code	disk_log	erlang
error_logger	ets	file	gen_event
gen_fsm	gen_server	global	init
mnesia	net_kernel	proc_lib	supervisor
supervisor_bridge	timer	sysApp	sysSupervisor

Table 4.2: *Modules for which there is an abstract version.*

SASL	mnemosyne	mnesia	obdc
os_mon	megaco	snmp	CosEvent
CosTransActions	CosNotification	CosTime	IC
Orber	ASN1	GS	COMET
inets	ssl	crypto	appmon
dbg	int	toolbar	

Table 4.3: *Unsupported applications.*

argument variable is bound to `badarg`. Also in this case we have to insert a return intermediate to restore the environment.

$$
\frac{\begin{array}{c}\rho(x^m) = \texttt{'erlang'} \wedge \rho(x^f) = \texttt{'apply'} \wedge \\ \rho(x_1) \in \text{Atom} \wedge \rho(x_1) \in \text{Atom} \wedge \rho(x_3) \in \text{List}\end{array}}{\begin{array}{l}\langle s\ \rho\ [[\![\ \texttt{call}\ x^m{:}x^f\ (x_1,x_2,x_3)\ ]\!] \mid c]\ \sigma\rangle \xrightarrow{\mu}_c \\ \langle [\,]\ \rho'\ [[\![\ \texttt{call}\ x_1{:}x_2(x'_1,\ldots,x'_i)\ ]\!], \text{return}(s,n,m) \mid c]\ \sigma\rangle \\ \textbf{where}\ n = \text{current}(\rho) \\ \qquad m = \text{module}(\sigma) \\ \qquad \langle v_1,\ldots,v_i\rangle = \rho(x_3) \\ \qquad \rho' = \text{bind}(\langle x'_1,\ldots,x'_i\rangle, \langle v_1,\ldots,v_i\rangle, \rho) \\ \qquad \mu = \rho(x^m){:}\rho(x^f)(\rho(x_1),\rho(x_2),\rho(x_3))\end{array}} \tag{4.21}
$$

$$
\frac{\begin{array}{c}\rho(x^m) = \texttt{'erlang'} \wedge \rho(x^f) = \texttt{'apply'} \wedge \\ (\rho(x_1) \notin \text{Atom} \vee \rho(x_1) \notin \text{Atom} \vee \rho(x_3) \notin \text{List})\end{array}}{\begin{array}{l}\langle s\ \rho\ [[\![\ \texttt{call}\ x^m{:}x^f\ (x_1,x_2,x_3)\ ]\!] \mid c]\ \sigma\rangle \xrightarrow{\mu}_c \\ \langle [\,]\ \rho'\ [[\![\ \texttt{primop}\ atom[\texttt{'raise'}]\ (x')\ ]\!], \text{return}(s,n,m) \mid c]\ \sigma\rangle \\ \textbf{where}\ n = \text{current}(\rho) \\ \qquad m = \text{module}(\sigma) \\ \qquad \rho' = \text{bind}(\langle x'\rangle, \langle \texttt{'badarg'}\rangle, \rho) \\ \qquad \mu = \rho(x^m){:}\rho(x^f)(\rho(x_1),\rho(x_2),\rho(x_3))\end{array}} \tag{4.22}
$$

The third `call` expression rule (4.23) handles a call to the built in function `erlang:throw` which is transforms into the `primop` expression `throw` which raises an exception.

(4.23)
$$\frac{\rho(x^m) = \texttt{'erlang'} \wedge \rho(x^f) = \texttt{'throw'}}{\begin{array}{l}\langle s\ \rho\ [[\![\texttt{call}\ x^m{:}x^f\ (x)]\!] \mid c]\ \sigma\rangle \xrightarrow{\rho(x^m):\rho(x^f)(\rho(x))}_c \\ \langle [\,]\ \rho\ [[\![\texttt{primop throw}\ (x)]\!], \mathsf{return}(s,n,m) \mid c]\ \sigma\rangle \\ \textbf{where}\ n = \mathsf{current}(\rho) \\ \qquad\quad m = \mathsf{module}(\sigma)\end{array}}$$

The fourth `call` expression rule (4.24), is the general rule for those function calls that does not have to be specially treated. The rule is similar to rule (4.17), with the exception that we have to set the current module to the value of the module variable x^m, since the called function may reside in another module.

(4.24)
$$\frac{m \in \mathrm{Atom} \wedge f \in \mathrm{Atom} \wedge \langle m,f,i\rangle \notin \mathrm{SpecialCalls}}{\begin{array}{l}\langle s\ \rho\ [[\![\texttt{call}\ x^m{:}x^f\ (x_1,\ldots,x_i)]\!] \mid c]\ \sigma\rangle \xrightarrow{\mu}_c \\ \langle [\,]\ \rho'\ [e, \mathsf{return}(s,n,m') \mid c]\ \sigma'\rangle \\ \textbf{where}\ m,f = \rho(x^m), \rho(x^f) \\ \qquad\quad n = \mathsf{current}(\rho) \\ \qquad\quad m' = \mathsf{module}(\sigma) \\ \qquad\quad \sigma' = \mathsf{set\_module}(m,\sigma) \\ \qquad\quad \langle\langle [[\![\texttt{fun}(x'_1,\ldots,x'_i) = e]\!], 0, m\rangle\rangle = \mathsf{func}(m,f,i,\sigma) \\ \qquad\quad \rho' = \mathsf{bind}(\langle x'_1,\ldots,x'_i\rangle, \langle \rho(x_1),\ldots,\rho(x_i)\rangle, \mathsf{set}(0,\rho)) \\ \qquad\quad \mu = m{:}f(\rho(x_1),\ldots,\rho(x_i))\end{array}}$$

The fifth `call` expression rule (4.25), deals with the case where the module name variable ^{m} or the function name variable x^f is not bound to an atom. The rule is similar to rule (4.19).

(4.25)
$$\frac{\{\rho(x^m), \rho(x^f)\} \not\subseteq \mathrm{Atom} \cup \{\top_{\mathrm{Value}}\}}{\begin{array}{l}\langle s\ \rho\ [[\![\texttt{call}\ m^f{:}x^f\ (x_1,\ldots,x_i)]\!] \mid c]\ \sigma\rangle \xrightarrow{\mu}_c \\ \langle [\,]\ \rho'\ [[\![\texttt{primop}\ \mathit{atom}[\texttt{'raise'}](x')]\!], \mathsf{return}(s,n,m) \mid c]\ \sigma\rangle \\ \textbf{where}\ n = \mathsf{current}(\rho) \\ \qquad\quad m = \mathsf{module}(\sigma) \\ \qquad\quad \rho' = \mathsf{bind}(\langle x'\rangle, \langle \texttt{'badarg'}\rangle, \rho) \\ \qquad\quad \mu = \rho(x^m) : \rho(x^f)(\rho(x_1),\ldots,\rho(x_i))\end{array}}$$

4.6.8.3 PrimOp

The `primop` expressions are used for implementation dependant aspects, and the intention is that they should only be generated by the compiler. Here we have chosen to use `primop` calls to model the raising of exceptions, both by the runtime system and as a result of calls to `throw/1`. The handling of global resources such as registered names are also modelled using the two `primop`s `read` and `publish`.

The first and second `primop` expressions rules (4.26. 4.27), deal with the `raise` and `throw` primitive operations. The evaluation of `primop` expressions `raise` and `throw` are transformed into the exception intermediate, with the arguments *type* and *value*. The *type* is in the case of `raise 'EXIT'` and in the case of `throw 'THROW'`. The value stack is replaced by an empty value stack. The handling of the `exception` intermediate is described in Section 4.6.9 and Section 4.6.10.

$$
\begin{array}{ll}
(4.26) & \langle s\ \rho\ [[\![\texttt{primop}\ atom[\texttt{'raise'}](x)]\!]\mid c]\ \sigma\rangle \xrightarrow{\mu}_c \\
& \langle [\,]\ \rho\ [\mathsf{exception}(\texttt{'EXIT'},\rho(x))\mid c]\ \sigma\rangle \\
& \textbf{where}\ \mu = \texttt{primop raise}(\rho(x))
\end{array}
$$

$$
\begin{array}{ll}
(4.27) & \langle s\ \rho\ [[\![\texttt{primop}\ atom[\texttt{'throw'}](x)]\!]\mid c]\ \sigma\rangle \xrightarrow{\mu}_c \\
& \langle [\,]\ \rho\ [\mathsf{exception}(\texttt{'THROW'},\rho(x))\mid c]\ \sigma\rangle \\
& \textbf{where}\ \mu = \texttt{primop throw}(\rho(x))
\end{array}
$$

The third `primop` expressions rule (4.28), deals with the examining side effects, i.e., side effects that examines some global resources in the ERLANG node. The evaluation of the `primop` expression results in a value v being pushed on the value stack, where the value is determined by the read function applied to the accumulated side effects in the global state.

$$
\begin{array}{ll}
(4.28) & \langle s\ \rho\ [[\![\texttt{primop}\ atom[\texttt{'read'}](x_1,x_2)]\!]\mid c]\ \sigma\rangle \xrightarrow{\mu}_c \\
& \langle [v\mid s]\ \rho\ c\ \sigma\rangle \\
& \textbf{where}\ v = \mathsf{read}(\rho(x_1),\rho(x_2),\mathsf{effects}(\sigma)) \\
& \qquad \mu = \texttt{primop 'read'}\ (\rho(x_1),\rho(x_2))
\end{array}
$$

The fourth `primop` expression rule (4.29), deals with side effects that change the global resources in the ERLANG node. The evaluation of the `primop` expression results in the addition of the key/value pair x_1/x_2 to the side effects and pushes `true` on the value stack.

$$
\begin{array}{ll}
(4.29) & \langle\, s\ \rho\ [\,[\![\texttt{primop}\ atom[\texttt{'publish'}](x_1,x_2)]\!]\mid c\,]\ \sigma\,\rangle \xrightarrow{\mu}_c \\
& \langle\,[\texttt{'true'}\mid s]\ \rho\ c\ \sigma'\,\rangle \\
& \textbf{where}\ \sigma' = \mathsf{publish}\,(\rho\,(x_1),\rho\,(x_2),\sigma\,) \\
& \qquad\quad \mu = \texttt{primop publish}\,(\rho\,(x_1),\rho\,(x_2)\,)
\end{array}
$$

The fifth and sixth `primop` expression rules, handle the general cases which can be used to define further implementation dependant constructs, by defining the auxiliary function primop: (Atom Values State) $\rightarrow$ (Value$_\perp$ State). The first rule (4.29) handles the case where the primop function returns successfully and the value returned is pushed on the value stack and the a new state is used. The second rule (4.30) deals with the case when the primop call fails and the `primop` call is transformed into a `primop raise` call.

$$
\begin{array}{ll}
(4.30) & \dfrac{a \notin \{\texttt{'raise'},\texttt{'throw'},\texttt{'read'},\texttt{'publish'}\} \wedge v \neq \perp}{\begin{array}{l}
\langle\, s\ \rho\ [\,[\![\texttt{primop}\ atom[a](x_1,\ldots,x_i)]\!]\mid c\,]\ \sigma\,\rangle \xrightarrow{\mu}_c \\
\langle\,[v\mid s]\ \rho\ c\ \sigma'\,\rangle \\
\textbf{where}\ \{v,\sigma'\} = \mathsf{primop}(a,\langle\,\rho\,(x_1),\ldots,\rho\,(x_i)\,\rangle,\sigma) \\
\qquad\quad \mu = \texttt{primop}\ a\,(\rho\,(x_1),\ldots,\rho\,(x_i)\,)
\end{array}}
\end{array}
$$

$$
\begin{array}{ll}
(4.31) & \dfrac{a \notin \{\texttt{'raise'},\texttt{'throw'},\texttt{'read'},\texttt{'publish'}\} \wedge v = \perp}{\begin{array}{l}
\langle\, s\ \rho\ [\,[\![\texttt{primop}\ atom[a](x_1,\ldots,x_i)]\!]\mid c\,]\ \sigma\,\rangle \xrightarrow{\mu}_c \\
\langle\, s\ \rho'\ [\,[\![\texttt{primop}\ atom[\texttt{'raise'}](x')]\!],\mathsf{return}()\mid c\,]\ \sigma\,\rangle \\
\textbf{where}\ \{v,\sigma'\} = \mathsf{primop}(a,\langle\,\rho\,(x_1),\ldots,\rho\,(x_i)\,\rangle,\sigma) \\
\qquad\quad n = \mathsf{current}\,(\rho\,) \\
\qquad\quad \rho' = \mathsf{bind}\,(\langle x'\rangle,\langle\,\texttt{'badarg'}\,\rangle,\rho\,) \\
\qquad\quad \mu = \texttt{primop}\ a\,(\rho\,(x_1),\ldots,\rho\,(x_i)\,)
\end{array}}
\end{array}
$$

4.6.9 Exception

The exception intermediate will simply remove expressions and intermediate from the code stack until it reaches an enclosing `catch` or the code stack is empty.

The first exception intermediate expression rule (4.32), is when the topmost element on the code stack is an exception intermediate and the following el-

ement is not a `catch` or a `return` intermediate, in which case the element following the exception intermediate is removed from the code stack.

$$(4.32)\quad \frac{c_1 \notin \{\mathsf{catch}(x_1,x_2,e,s,n,m),\mathsf{return}(s,n,m)\}}{\langle\,[\,]\;\rho\;[\mathsf{exception}(v_1,v_2),c_1 \mid c]\;\sigma\,\rangle \xrightarrow{\tau}_c \langle\,[\,]\;\rho\;[\mathsf{exception}(v_1,v_2) \mid c]\;\sigma\,\rangle}$$

The second exception intermediate expression rule (4.33), handles the case where the element after the exception is a return intermediate, then the return resets the context and state, but the stack remains empty and the exception intermediate remains on the code stack.

The third exception intermediate expression rule (4.34), handles the case where the code stack is empty, in which case the process terminates abnormally and the action `exit(fail)` is issued with the resulting `NIL` configuration.

$$(4.33)\quad \langle\,[\,]\;\rho\;[\mathsf{exception}(v_1,v_2),\mathsf{return}(s,n,m) \mid c]\;\sigma\,\rangle \xrightarrow{\tau}_c \langle\,[\,]\;\rho'\;[\mathsf{exception}(v_1,v_2) \mid c]\;\sigma'\,\rangle$$
$$\textbf{where}\ \rho' = \mathsf{set}(n,\rho)$$
$$\sigma' = \mathsf{set\_module}(m,\sigma)$$

$$(4.34)\quad \langle\,[\,]\;\rho\;[\mathsf{exception}(v_1,v_2)]\;\sigma\,\rangle \xrightarrow{\texttt{exit(fail)}}_c \texttt{NIL}$$

4.6.10 Exception Handling

The first `try-catch` expression rule transform the expression into the expression to be protected e_1, followed by the catch and restore intermediate expressions. The purpose of the restore intermediate is to restore the environment previous to the evaluation of the catch intermediate, since the catch intermediate may create bindings.

The catch intermediate needs the value stack s, current environment name n and module name m as parameters in the same manner as the return intermediate.

$$(4.35)\quad \langle\,s\;\rho\;[[\![\texttt{try}\ e_1\ \texttt{catch(}x_1,x_2\texttt{)}\ \texttt{->}\ e_2]\!] \mid c]\;\sigma\,\rangle \xrightarrow{\tau}_c \langle\,s\;\rho\;[e_1,\mathsf{catch}(x_1,x_2,e_2,s,n,m),\mathsf{restore}(n) \mid c]\;\sigma\,\rangle$$
$$\textbf{where}\ n = \mathsf{current}(\rho)$$
$$m = \mathsf{module}(\sigma)$$

The first catch intermediate expression rule (4.36), handles the case where catch intermediate expressions is topmost in the code stack, the intermediate is simply removed.

$$\langle s\ \rho\ [\mathsf{catch}(x_1,x_2,e,s_1,n,m) \mid c]\ \sigma\rangle \xrightarrow{\tau}_c \langle s\ \rho\ c\ \sigma\rangle \tag{4.36}$$

The second catch intermediate expression rule(4.37), handles the case where exception intermediate is topmost followed by a catch intermediate, then the two intermediates are transformed the expression e from the catch intermediates.

The configuration is restored to the one previous to the evaluation of the `try-catch` expression and with the variables x_1 and x_2 bound to the type and value of the exception.

$$\begin{aligned}&\langle [\,]\ \rho\ [\mathsf{exception}(v_1,v_2),\mathsf{catch}(x_1,x_2,e,s,n,m) \mid c]\ \sigma\rangle \xrightarrow{\tau}_c \\ &\langle s\ \rho'\ [e \mid c]\ \sigma'\rangle \\ &\textbf{where } \rho' = \mathsf{bind}(\langle x_1,x_2\rangle,\langle v_1,v_2\rangle,\mathsf{set}(n,\rho)) \\ &\qquad \sigma' = \mathsf{set\_module}(m,\sigma)\end{aligned} \tag{4.37}$$

4.6.11 Message Retrieval

The rules for `deliver` and `receive` expressions and for rec intermediate expressions, deal with the message retrieval from the input queue and the delivery of messages to the message queue from the input stream. In order to correctly model the complex selection of a clause to match in a `receive` expression we use the intermediate expressions rec and message.

The first delivery rule simply delivers a message from the stream to the queue and is always applicable (provided there are some message still left in the input stream). This rule is the only source of nondeterminism in the concrete semantics.

$$\langle s\ \rho\ c\ \sigma\rangle \xrightarrow{\tau}_c \langle s\ \rho\ c\ \mathsf{deliver}(\sigma)\rangle \tag{4.38}$$

The first `receive` expression rule, transforms the `receive` into a rec intermediate expression that has the input queue as an argument (actually two copies, where one acts as a scratch copy) to make the `receive` expression atomic. The clauses of the `receive` expression are converted into a list of clauses.

$$
\begin{array}{ll}
(4.39) & \langle\, s\ \rho\ [\,[\![\,\texttt{receive}\ \mathit{clauses}\ \texttt{after}\ x_t\ \texttt{->}\ e_t\ \texttt{end}\,]\!]\mid c\,]\ \sigma\,\rangle \xrightarrow{\tau}_c \\
& \langle\, s\ \rho\ [\,\mathsf{rec}(q,q,cls,x_t,e_t)\mid c\,]\ \sigma\,\rangle \\
& \textbf{where}\ q = \mathsf{queue}(\,\sigma\,) \\
& \qquad \mathit{clause}_1 \ldots \mathit{clause}_i = \mathit{clauses} \\
& \qquad cls = [\mathit{clause}_1, \ldots, \mathit{clause}_i]
\end{array}
$$

The four rec intermediate expression rules has the combined effect that they will receive the first message to match a clause, all of whose predecessor fail to match any clause. If no clause match the `after` clause body is evaluated provided the timeout is not infinity.

The first rec intermediate expression rule (4.40), deals with the rec intermediate when the first element in the queue v matches the first clause's pattern. The match results in a new context, and the rec intermediate is transformed into the guard g followed by an intermediate if expression. The true branch of the if intermediate contains the message intermediate, the body e of the rec intermediate followed by a restore intermediate. The false branch of the if intermediate contains a restore intermediate and the rec intermediate without the first queue element q'.

$$
\begin{array}{ll}
(4.40) & \dfrac{\bot \notin \mathsf{match}(\,p,v\,)}{
\begin{array}{l}
\langle\, s\ \rho\ [\,\mathsf{rec}([\,v\mid q'\,],q,cls,x_t,e_t)\mid c\,]\ \sigma\,\rangle \xrightarrow{\tau}_c \\
\langle\, s\ \rho'\ [\,g,\mathsf{if}(c_1,c_2)\mid c\,]\ \sigma\,\rangle \\
\textbf{where}\ [\,[\![\,p\ \texttt{when}\ g\ \texttt{->}\ e\,]\!]\mid cls'\,] = cls \\
\qquad \rho' = \mathsf{update}(\,\mathsf{match}(\,p,v\,),\rho\,) \\
\qquad n = \mathsf{current}(\,\rho\,) \\
\qquad c_1 = [\,\mathsf{message}(v),e,\mathsf{restore}(n)\,] \\
\qquad c_2 = [\,\mathsf{restore}(n),\mathsf{rec}(q',q,cls,x_t,e_t)\,]
\end{array}}
\end{array}
$$

The second rec intermediate expression rule(4.41), deals with the rec intermediate when the first element v in the queue does not match the first clause's pattern. The rec intermediate is transformed into a rec intermediate without the value v in the queue.

$$
\begin{array}{ll}
(4.41) & \dfrac{\bot \in \mathtt{match}(\,p,v\,)}{
\begin{array}{l}
\langle\, s\ \rho\ [\,\mathsf{rec}([\,v\mid q'\,],q,cls,x_t,e_t)\mid c\,]\ \sigma\,\rangle \xrightarrow{\tau}_c \\
\langle\, s\ \rho\ [\,\mathsf{rec}(q',q,cls,x_t,e_t)\mid c\,]\ \sigma\,\rangle
\end{array}}
\end{array}
$$

The third rec intermediate expression rule (4.42), deals with the rec intermediate when queue is empty. The rec intermediate is transformed into a rec intermediate without the first clause, and the queue restored to q.

$$\langle s\ \rho\ [\mathsf{rec}([\,],q,[cl \mid cls],x_t,e_t) \mid c]\ \sigma\ \rangle \xrightarrow{\tau}_c \langle s\ \rho\ [\mathsf{rec}(q,q,cls,x_t,e_t) \mid c]\ \sigma\ \rangle \tag{4.42}$$

The fourth rec intermediate expression rule (4.43), deals with the rec intermediate when queue is empty, there are no more clauses and the timeout x_t is not `infinity`. The rec intermediate transforms into the timeout body e_t. This evaluation results in the action `timeout`.

$$\frac{\rho(x_t) \neq \texttt{'infinity'}}{\langle s\ \rho\ [\mathsf{rec}(q,q,[\,],x_t,e_t) \mid c]x\ \sigma\ \rangle \xrightarrow{\texttt{timeout}}_c \langle s\ \rho\ [e_t \mid c]\ \sigma\ \rangle} \tag{4.43}$$

The first message intermediate expression rule removes the intermediate and issues the action `receive v`.

$$\langle s\ \rho\ [\mathsf{message}(v) \mid c]\ \sigma\ \rangle \xrightarrow{\texttt{receive}\ v}_c \langle s\ \rho\ c\ \mathsf{remove}(v,\sigma)\ \rangle \tag{4.44}$$

This concludes the concrete semantics, where we have defined the domains and transformation rules for the concrete abstract machine.

5. Abstract Semantics

In this chapter we present a semantics for CORE ERLANG that is an abstraction of the semantics presented in the previous chapter. The semantics is described in terms of how it differs from the concrete semantics of the previous chapter.

5.1 Abstract Semantics

The purpose of the abstract semantics is to act as a bridge between the concrete semantics and the approximations we have to do in order to make the extraction of supervision structures feasible. The abstract semantics must be a safe abstraction of the concrete semantics.

The concrete semantics is formulated in a manner close to an implementation in order that it should be easy to see that it captures ERLANG appropriately. The approximations are presented as a more abstract semantics with altered rules, which can formally be proved to abstract the concrete semantics. In this way we can be confident that the approximations we make are safe, in the sense that we will consider all possible behaviours of the application for which we extract the supervision structure.

In order to make the evaluation of a CORE ERLANG expression in the semantics tractable and to ensure termination, we have imposed two important restrictions: (1) the input stream is unknown and hence the message queue; (2) nested calls may be evaluated only up to a max call depth. A further source of approximations are unknown libraries or code that does not yet exist. This will limit the precision of our evaluation.

As a consequence of these restrictions we must introduce an approximation of all possible values, which we shall denote $\top$. However in ERLANG it does not suffice to approximate the values. We must also be to be to handle the situation when we do not now whether a value or an exception is the result. This we have chosen to represent by explicitly adding the unknown exception, denoted as `exception`$(\top, \top)$.

A safe approximation of `receive` expressions must consider all the clauses of the expression, including the timeout rule. Cutting off the evaluation at a certain call depth means that the result of the last call and its potential side effects must be approximated.

The approximation of side effects is handled by "publishing" a side effect with the key $\top$ in the key/value pair modelling a side effect. Note that from

this point on all side effects must be approximated, e.g., the result of a lookup in an ETS table is not known if there have been changes to the table that have been approximated. The definition of the read function ensures that if a side effect with a key containing $\top$ is present, the result of the call is the approximation $\top$.

This gives rise to three distinct sources of nondeterminism[1] in the abstract semantics: all clauses of a `receive` expression must be considered; approximations within values switched over by a `case` expression necessitates the examination of all clauses that could have matched; and finally indirectly that all calls that produce side effects, after a function call has been approximated, generate new approximations.

In the presentation below all rules of the concrete semantics are included unless they have been replaced, in which case this is stated in the description of the replacing rule.

The added rules are:

Approximation of Function Calls (5.1) – (5.3)

Conditional Expressions (5.5) – (5.6)

Function Expressions (5.7) – (5.8)

The changed rules are:

Conditional Expressions (4.14) by (5.4)

Message Retrieval (4.38), (4.40) – (4.43) by (5.9) – (5.12) respectively.

5.2 Auxiliary Functions

The subst function takes a pattern and an environment and replaces all occurrences of variables in the pattern with their values as given by the environment.

$$\mathsf{subst}\colon (\text{Pattern} \quad \text{Env}) \to \text{Value}^{\top}$$

$$\mathsf{subst}(p,\rho) = \begin{cases} \rho(x) & \textbf{if } p = [\![x]\!] \\ \rho(x) & \textbf{if } p = [\![x = p_1]\!] \\ v & \textbf{if } p = [\![literal[v]]\!] \\ [\mathsf{subst}(p_1,\rho) \mid \mathsf{subst}(p_2,\rho)] & \textbf{if } p = [\![[p_1 \mid p_2]]\!] \\ \{\mathsf{subst}(p_1,\rho),\ldots,\mathsf{subst}(p_i,\rho)\} & \textbf{if } p = [\![\{p_1,\ldots,p_i\}]\!] \\ \langle \mathsf{subst}(p_1,\rho),\ldots,\mathsf{subst}(p_i,\rho)\rangle & \textbf{if } p = [\![\langle p_1,\ldots,p_i\rangle]\!] \end{cases}$$

[1] Nondeterminism is here used in the sense that more than one rule can be enabled in a machine configuration.

5.3 Approximation of Function Calls

These are the rules added to handle the case when a function call has been approximated and consequently the topmost element of the code stack is the `approx` intermediate. Either because the maximum number of nested calls has been exceeded, or the code for the function is not available.

All three approximation rules are applicable when the first element on the code stack is an approximation, they are all applicable at the same time.

The first approximation rule (5.1) states that the machine may remain in the same configuration and do nothing. This allows the abstract set of rules to simulate the concrete set of rules when they are evaluating expressions that the abstract set of rules have approximated.

The second approximation rule (5.2), results in the approx intermediate expression being removed from the code stack and the approximated value $\top$ being placed on top of the value stack. This simulates the application of a function call resulting in a value.

The third approximation rule (5.3), results in the approx intermediate expression being replaced on the code stack by an exception intermediate expression with the *type* and *value* arguments both being $\top$. This intermediate denotes the exception of unknown type and value.

$$\langle s\ \rho\ [\mathsf{approx}() \mid c]\ \sigma \rangle \xrightarrow{\top}_a \langle s\ \rho\ [\mathsf{approx}() \mid c]\ \sigma \rangle \tag{5.1}$$

$$\langle s\ \rho\ [\mathsf{approx}() \mid c]\ \sigma \rangle \xrightarrow{\tau}_a \langle [\top \mid s]\ \rho\ c\ \sigma \rangle \tag{5.2}$$

$$\langle s\ \rho\ [\mathsf{approx}() \mid c]\ \sigma \rangle \xrightarrow{\tau}_a \langle [\,]\ \rho\ [\mathsf{exception}(\top,\top) \mid c]\ \sigma \rangle \tag{5.3}$$

5.4 Conditional Expressions

In order to handle the presence of approximations in the value that is switched on by a `case` expression, the second concrete `case` expression rule (4.14) must be modified to examine all the possible outcomes. The modified rule is applicable both when the first clause does not match the value sequence to which the variable sequence $\langle x_1,\ldots,x_i \rangle$ is bound, and when the value sequence contains an approximation. The rule (5.4) replaces rule (4.14).

$$\frac{\bot \in \mathsf{match}(p, \langle v_1,\ldots,v_i \rangle) \vee \mathsf{contain\_top}(\langle v_1,\ldots,v_i \rangle)}{\begin{array}{l}\langle s\ \rho\ [[\![\texttt{case}\ \langle x_1\ \ldots,x_i \rangle\ \texttt{of}\ \mathit{clause}\ \mathit{clauses}\ \texttt{end}]\!] \mid c]\ \sigma \rangle \xrightarrow{\tau}_a \\ \langle s\ \rho\ [[\![\texttt{case}\ \langle x_1\ \ldots,x_i \rangle\ \texttt{of}\ \mathit{clauses}\ \texttt{end}]\!] \mid c]\ \sigma \rangle \\ \textbf{where}\ [\![p\ \texttt{when}\ g\ \texttt{->}\ e]\!] = \mathit{clause} \\ \qquad \langle v_1,\ldots,v_i \rangle = \langle \rho(x_1)\ \ldots,\rho(x_i) \rangle\end{array}} \tag{5.4}$$

The if intermediate is nondeterministic if the topmost value of the value stack is $\top$, since we do not know whether the guard would have evaluated to `true` or `false`.

$$\langle [\top \mid s]\ \rho\ [\mathsf{if}([c_1,\ldots,c_i],c') \mid c]\ \sigma \rangle \xrightarrow{\tau}_a \langle s\ \rho\ [c_1,\ldots,c_i \mid c]\ \sigma \rangle \tag{5.5}$$

$$\langle [\top \mid s]\ \rho\ [\mathsf{if}(c',[c_1,\ldots,c_i]) \mid c]\ \sigma \rangle \xrightarrow{\tau}_a \langle s\ \rho\ [c_1,\ldots,c_i \mid c]\ \sigma \rangle \tag{5.6}$$

5.5 Function Expressions

For function expressions the abstract semantics differs from the concrete in two respects. First the function or module in which it resides may be unknown. Secondly we have imposed a maximum call depth, since we do not know what this depth is (in the general case) the following rule of approximation of an `apply` expression is always applicable. Note that if the function or module in which it resides is unknown this is the only applicable rule. Furthermore it can be choosen anytime rules rule (4.17), rule (4.18) are applicable.

5.5.1 Apply

The first `apply` expression rule deals with the case where either the function is unknown or the maximum call depth is reached; then the call is transformed into an approx intermediate expression, and the "unknown" side effect is published.

$$\begin{aligned} &\langle s\ \rho\ [[\![\mathtt{apply}\ x^f\ (x_1,\ldots,x_i)]\!] \mid c]\ \sigma \rangle \xrightarrow{\top}_a \\ &\langle s\ \rho\ [\mathsf{approx}() \mid c]\ \sigma' \rangle \\ &\textbf{where } \sigma' = \mathsf{publish}(\top,\top,\sigma) \end{aligned} \tag{5.7}$$

5.5.2 Call

The first `call` expression rule deals with the case where either module or function is unknown or the maximum depth is reached, where the call is transformed into a approx intermediate. rule (5.8) is analogous to rule (5.7) for `call` expressions.

$$
\begin{array}{ll}
(5.8) & \langle\, s\ \rho\ [\,[\![\,\texttt{call}\ x^m{:}x^f\,(x_1,\ldots,x_i)\,]\!]\mid c\,]\ \sigma\,\rangle \xrightarrow{\top}_a \\
& \langle\, s\ \rho\ [\,\mathsf{approx}()\mid c\,]\ \sigma'\,\rangle \\
& \textbf{where}\ \sigma' = \mathsf{publish}(\top,\top,\sigma)
\end{array}
$$

5.6 Message Retrieval

The handling of the `receive` expression and rec intermediate expression is simpler in the abstract semantics where we do not consider any explicit message queue or input stream, instead all clauses are investigated. The patterns of a clause are matched against the approximation, and the timeout clause is investigated unless the timeout is set to `infinity`. In all the rules the message queues are replaced by an approximation.

The first rec rule replaces rule (4.40) and is always applicable since we do not know what the elements in the message queue are, and then the first clause's pattern might match the first element. We have to replace the variables in the pattern so that it can serve as an argument for the message intermediate. Otherwise the rule is identical to the concrete rule.

$$
\begin{array}{ll}
(5.9) & \langle\, s\ \rho\ [\,\mathsf{rec}(\top,\top,cls,x_t,e_t)\mid c\,]\ \sigma\,\rangle \xrightarrow{\tau}_a \langle\, s\ \rho'\ [\,g,\mathsf{if}(c_1,c_2)\mid c\,]\ \sigma\,\rangle \\
& \quad \textbf{where}\ [\,[\![\,p\ \texttt{when}\ g\ \texttt{->}\ e\,]\!]\mid cls'\,] = cls \\
& \qquad \rho' = \mathsf{update}\,(\,\mathsf{match}\,(\,p,\top\,),\rho\,) \\
& \qquad n = \mathsf{current}\,(\,\rho\,) \\
& \qquad p' = \mathsf{subst}\,(\,p,\mathsf{match}\,(\,p,\top\,)\,) \\
& \qquad c_1 = [\,\mathsf{message}(p'),e,\mathsf{restore}(n)\,] \\
& \qquad c_2 = [\,\mathsf{restore}(n),\mathsf{rec}(\top,\top,cls,x_t,e_t)\,]
\end{array}
$$

The second rec rule, replaces rule (4.41) and is always applicable since we do not know what the elements in the message are, and then the first clause's pattern might not match the first element. Since the queues are approximated the configuration is not changed, this rule is included to make the simulation of the concrete rules simpler to understand.

$$
\begin{array}{ll}
(5.10) & \langle\, s\ \rho\ [\,\mathsf{rec}(\top,\top,cls,x_t,e_t)\mid c\,]\ \sigma\,\rangle \xrightarrow{\tau}_a \\
& \langle\, s\ \rho\ [\,\mathsf{rec}(\top,\top,cls,x_t,e_t)\mid c\,]\ \sigma\,\rangle
\end{array}
$$

The third rec rule, replaces rule (4.42) and is always applicable since we do not know what the elements in the message are, and then the queue might be empty.

$$\langle s\ \rho\ [\mathsf{rec}(\top,\top,[cl \mid cls],x_t,e_t) \mid c]\ \sigma\rangle \xrightarrow{\tau}_a \langle s\ \rho\ [\mathsf{rec}(\top,\top,cls,x_t,e_t) \mid c]\ \sigma\rangle \tag{5.11}$$

The fourth rec rule, replaces rule (4.43) and it deals with the rec intermediate when the queue is empty, there are no more clauses and the timeout x_t is not `infinity`.

$$\frac{\rho(x_t) \neq \texttt{'infinity'}}{\langle s\ \rho\ [\mathsf{rec}(\top,\top,[\,],x_t,e_t) \mid c]x\ \sigma\rangle \xrightarrow{\texttt{timeout}}_a \langle s\ \rho\ [e_t \mid c]\ \sigma\rangle} \tag{5.12}$$

5.7 Relation between the Abstract and Concrete Configuration

In this section we will show that the abstract semantics is an abstraction of the concrete semantics, in the sense that the abstract semantics simulates the concrete semantics.

We want to show that for each possible sequence of transitions of the concrete semantics the abstract semantics can perform a simulating sequence of transitions that results in a state that abstracts the state resulting from the concrete sequence of transitions. To show this we want to show that the concrete semantics is monotonic, and for each transition rule in the concrete semantics that if the rule changes a configuration Γ_1^c to configuration Γ_2^c, and Γ_1^a is an abstraction of Γ_1^c, then there is a rule in the abstract semantics that will transform Γ_1^a into some Γ_2^a that abstracts Γ_2^c. The reason is that this means that any transition by a rule in the concrete semantics can be simulated by itself.

We will first present and define what we mean by saying that a configuration abstracts another and then continue to show the simulation property.

5.7.1 Abstraction relation

We will in this section define, explain and motivate the abstraction relation used. The abstraction relation is denoted by $\sqsupseteq$.

For all variables over parts of the configurations we will add superscripts c and a to denote that they belong to the concrete or abstract configuration respectively.

The abstraction relation is binary for the topmost domains, but quaternary for some domains. The reason for the quaternary relation is that for domains that have elements that can contain variables, the abstraction relation must be defined in relation to the contexts. The quaternary relation is denoted $\varphi^a \sqsupseteq_{(\rho^a,\rho^c)} \psi^c$ where φ^a abstracts ψ^c when the variables in φ^a are replaced by values in the current environment in ρ^a and correspondingly for ψ^c.

We will now define the relations for each domain, bottom up starting with the values.

5.7.1.1 Value

A literal value is abstracted by itself.

$$l \sqsupseteq_{(\rho^a,\rho^c)} l$$

All values are abstracted by the value domain's top element.

$$\top \sqsupseteq_{(\rho^a,\rho^c)} v^c$$

A function closure is abstracted if the function and module are identical and the environment references is abstracted. Environment references abstract if the environments they reference abstract. How environments abstract is described in the next subsection.

$$\begin{aligned} \langle\langle f^a, n^a, m^a \rangle\rangle \sqsupseteq_{(\rho^a,\rho^c)} \langle\langle f^c, n^c, m^c \rangle\rangle \quad \triangleq \quad & f^a = f^c \\ \wedge \quad & \rho^a(n^a) \sqsupseteq \rho^c(n^c) \\ \wedge \quad & m^a = m^c \end{aligned}$$

A list, tuple or sequence of values are abstracted if they contain the same number of elements and all the elements abstract pairwise.

$$[v_1^a, \ldots, v_i^a] \sqsupseteq_{(\rho^a,\rho^c)} [v_1^c, \ldots, v_i^c] \triangleq v_1^a \sqsupseteq_{(\rho^a,\rho^c)} v_1^c \wedge \quad \wedge v_i^a \sqsupseteq_{(\rho^a,\rho^c)} v_i^c$$

$$\{v_1^a, \ldots, v_i^a\} \sqsupseteq_{(\rho^a,\rho^c)} \{v_1^c, \ldots, v_i^c\} \triangleq v_1^a \sqsupseteq_{(\rho^a,\rho^c)} v_1^c \wedge \quad \wedge v_i^a \sqsupseteq_{(\rho^a,\rho^c)} v_i^c$$

$$\langle v_1^a, \ldots, v_i^a \rangle \sqsupseteq_{(\rho^a,\rho^c)} \langle v_1^c, \ldots, v_i^c \rangle \triangleq v_1^a \sqsupseteq_{(\rho^a,\rho^c)} v_1^c \wedge \quad \wedge v_i^a \sqsupseteq_{(\rho^a,\rho^c)} v_i^c$$

5.7.1.2 Env

An environment is abstracted if it for all variables, in the domain, they are mapped to values that are abstracted.

$$env^a \sqsupseteq env^c \triangleq (\forall\, x^c \in \mathtt{dom}(env^c))(env^a(x^c) \sqsupseteq env^c(x^c))$$

5.7.1.3 Value Stack

The empty value stack $[\,]$ is abstracted by itself.

$$[\,] \sqsupseteq_{(\rho^a,\rho^c)} [\,]$$

A nonempty value stack is abstracted if the top element is abstracted and the remainder of the stack is abstracted.

$$\begin{array}{rcll} [\,v^a \mid s^a\,] \sqsupseteq_{(\rho^a,\rho^c)} [\,v^a \mid s^a\,] & \stackrel{\triangle}{=} & v^a \sqsupseteq_{(\rho^a,\rho^c)} v^c \\ & \wedge & s_t^a \sqsupseteq_{(\rho^a,\rho^c)} s_t^c \end{array}$$

5.7.1.4 Context

A context is abstracted when the application of the context to the current environment name is abstracted.

$$\begin{array}{l} \langle n^a, envmap^a, k^a \rangle \sqsupseteq \langle n^c, envmap^c, k^c \rangle \stackrel{\triangle}{=} \\ \quad \langle n^a, envmap^a, k^a \rangle(n^a) \sqsupseteq \langle n^c, envmap^c, k^c \rangle(n^c) \end{array}$$

A context applied to environment name is the environment mapped to that name in the context's environment map.

$$\begin{array}{lcl} \rho\,(n_1) & \stackrel{\triangle}{=} & envmap\,(n_1) \\ & \texttt{where} & \rho = \langle n, envmap, k \rangle \end{array}$$

5.7.1.5 Code Stack

The empty code stack is abstracted by itself.

$$[\,] \sqsupseteq_{(\rho^a,\rho^c)} [\,]$$

The nonempty code stack is abstracted if the top element are either identical or abstract and the remainder of the stack is abstracted.

$$\begin{array}{rcll} [\,c_1^a \mid c^a\,] \sqsupseteq_{(\rho^a,\rho^c)} [\,c_1^c \mid c^c\,] & \stackrel{\triangle}{=} & (& c_1^a = c_1^c \\ & & \vee & c_1^a \sqsupseteq_{(\rho^a,\rho^c)} c_1^c \quad) \\ & \wedge & & c^a \sqsupseteq_{(\rho^a,\rho^c)} c^c \end{array}$$

The message and exception intermediate expressions are abstracted if the arguments of the expressions are abstracted.

$$\begin{array}{lcll}
\mathsf{message}(v^a) \sqsupseteq_{(\rho^a,\rho^c)} \mathsf{message}(v^c) & \stackrel{\triangle}{=} & v^a \sqsupseteq_{(\rho^a,\rho^c)} v^c \\
\mathsf{exception}(v_1^a, v_2^a) \sqsupseteq_{(\rho^a,\rho^c)} \mathsf{exception}(v_1^c, v_2^c) & \stackrel{\triangle}{=} & v_1^a \sqsupseteq_{(\rho^a,\rho^c)} v_1^c \\
& & \wedge \quad v_2^a \sqsupseteq_{(\rho^a,\rho^c)} v_2^c
\end{array}$$

The restore intermediate expression is abstracted if the name of the environment which is restored is abstracted in the current context.

$$\mathsf{restore}(n^a) \sqsupseteq_{(\rho^a,\rho^c)} \mathsf{restore}(n^c) \stackrel{\triangle}{=} \rho^a(n^a) \sqsupseteq \rho^c(n^c)$$

The return intermediate expression is abstracted if the stack is abstracted, the returned environment is abstracted and the module is identical.

$$\begin{array}{lcll}
\mathsf{return}(s^a, n^a, m^a) \sqsupseteq_{(\rho^a,\rho^c)} \mathsf{return}(s^c, n^c, m^c) & \stackrel{\triangle}{=} & s^a \sqsupseteq_{(\rho^a,\rho^c)} s^c \\
& & \wedge \quad \rho^a(n^a) \sqsupseteq \rho^c(n^c) \\
& & \wedge \quad m^a = m^c
\end{array}$$

The catch intermediate expression is abstracted if the stack is abstracted, the returned environment is abstracted and the module is identical.

$$\begin{array}{l}
\mathsf{catch}(x_1, x_2, e, s^a, n^a, m^a) \sqsupseteq_{(\rho^a,\rho^c)} \mathsf{catch}(x_1, x_2, e, s^c, n^c, m^c) \stackrel{\triangle}{=} \\
\quad s^a \sqsupseteq_{(\rho^a,\rho^c)} s^c \\
\wedge \quad \rho^a(n^a) \sqsupseteq \rho^c(n^c) \\
\wedge \quad m^a = m^c
\end{array}$$

The rec intermediate expression is abstracted if the queue arguments in the abstract rec intermediate is approximated with $\top$. The queue arguments may be approximated since the abstract semantics does not model the input queues. If the more abstract code stacks element is a rec intermediate which queue arguments are not approximated, then it the queues to be identical to the abstracted intermediate

$$\mathsf{rec}(\top, \top, cls, x_t, e_t) \sqsupseteq_{(\rho^a,\rho^c)} \mathsf{rec}(q_1^c, q_2^c, cls, x_t, e_t)$$

5.7.1.6 Effects

The empty sequence of effects is abstracted by itself.

$$[\,] \sqsupseteq_{(\rho^a,\rho^c)} [\,]$$

A sequence of effects that has a first element that has the domain top element as key is abstracted when the rest of the abstract sequence of effects abstracts a suffix of the concrete sequence of effects.

$$\begin{array}{l}[\,\langle\,\top, v_2^a\,\rangle \mid \pi_t^a\,] \sqsupseteq_{(\rho^a,\rho^c)} \pi^c \stackrel{\triangle}{=} \\ \quad (\exists\, \pi_1^c, \ldots, \pi_i^c)(\pi^c = [\,\pi_1^c, \ldots, \pi_i^c \mid \pi_t^c\,] \wedge \pi_t^a \sqsupseteq_{(\rho^a,\rho^c)} \pi_t^c))\end{array}$$

A sequence of effects that does not have a first element that has the domain top element as key is abstracted when the first key and value pair $\langle\, v_1^a, v_2^a\,\rangle$ is abstracted and the remainder of the sequence is abstracted.

$$\begin{array}{lll}[\,\langle\, v_1^a, v_2^a\,\rangle \mid \pi_t^a\,] \sqsupseteq_{(\rho^a,\rho^c)} [\,\langle\, v_1^c, v_2^c\,\rangle \mid \pi_t^c\,] & \stackrel{\triangle}{=} & v_1^a \neq \top \\ & \wedge & v_1^a \sqsupseteq_{(\rho^a,\rho^c)} v_1^c \\ & \wedge & v_2^a \sqsupseteq_{(\rho^a,\rho^c)} v_2^c \\ & \wedge & \pi_t^a \sqsupseteq_{(\rho^a,\rho^c)} \pi_t^c\end{array}$$

5.7.1.7 State

A state is abstracted when the abstract state's queue is either $\top$ or identical to the concrete, the current module and modules identical and the effects abstracted.

$$\begin{array}{lll}\langle\, m^a, q^a, ms^a, \pi^a\,\rangle \sqsupseteq_{(\rho^a,\rho^c)} \langle\, m^c, q^c, ms^c, \pi^c\,\rangle & \stackrel{\triangle}{=} & m^a = m^c \\ & \wedge & (\quad q^a = \top \\ & & \quad \vee\ q^a = q^c) \\ & \wedge & ms^a = ms^c \\ & \wedge & \pi^a \sqsupseteq_{(\rho^a,\rho^c)} \pi^c\end{array}$$

5.7.1.8 Configuration

The empty configuration `NIL` is abstracted by itself.

$$\texttt{NIL} \sqsupseteq \texttt{NIL}$$

There are three cases of abstraction for the nonempty configuration. First if the first element of the abstract code stack is not the approx intermediate, then the configuration abstracts if the stacks, environments, code stacks and states abstract.

$$\langle s^a\ \rho^a\ [\,c_1^a \mid c^a\,]\ \sigma^a\rangle \sqsupseteq \langle s^c\ \rho^c\ c^c\ \sigma^c\rangle \stackrel{\triangle}{=} \begin{array}{lll} & c_1^a \neq \mathsf{approx}() \\ \wedge & s^a \sqsupseteq_{(\rho^a,\rho^c)} s^c \\ \wedge & \rho^a \sqsupseteq \rho^c \\ \wedge & [\,c_1^a \mid c^a\,] \sqsupseteq_{(\rho^a,\rho^c)} c^c \\ \wedge & \sigma^a \sqsupseteq_{(\rho^a,\rho^c)} \sigma^c \end{array}$$

Secondly if the first element of the abstract code stack is the approx intermediate and the concrete code stack has the return intermediate then the configuration is abstracted if either the returned stack and environment are abstracted, the remaining code stacks is abstracted and the resulting state is abstracted. This is the case where the return matches the return from the actual call that has been approximated. Or the concrete configuration without the top most return intermediate is abstracted by the abstracted configuration. This is the case where the abstract semantics stutter.

$$\begin{array}{l} \langle s^a\ \rho^a\ [\,\mathsf{approx}() \mid c^a\,]\ \sigma^a\rangle \sqsupseteq \langle s^c\ \rho^c\ [\,\mathsf{return}(s_1^c,n^c,m^c) \mid c^c\,]\ \sigma^c\rangle \stackrel{\triangle}{=} \\ \quad (\quad \rho_1^c = \mathsf{set}\,(n^c,\rho^c\,) \\ \quad\ \wedge\ \langle m_1^c,\sigma_1^c\rangle = \mathsf{set\_module}\,(\sigma^c,m^c\,) \\ \quad\ \wedge\ s^a \sqsupseteq_{(\rho^a,\rho_1^c)} s_1^c \\ \quad\ \wedge\ \rho^a \sqsupseteq \rho_1^c \\ \quad\ \wedge\ c^a \sqsupseteq_{(\rho^a,\rho_1^c)} c^c \\ \quad\ \wedge\ \sigma^a \sqsupseteq_{(\rho^a,\rho_1^c)} \sigma_1^c) \\ \vee\quad \langle s^a\ \rho^a\ [\,\mathsf{approx}() \mid c^a\,]\ \sigma^a\rangle \sqsupseteq \langle s_1^c\ \rho^c\ c^c\ \sigma^c\rangle \end{array}$$

Finally if the first element of the abstract code stack is the approx intermediate and the concrete code stack does not have the return intermediate then the configuration is abstracted if the concrete without the top most element is abstracted.

$$\begin{array}{l} \langle s^a\ \rho^a\ [\,\mathsf{approx}() \mid c^a\,]\ \sigma^a\rangle \sqsupseteq \langle s^c\ \rho^c\ [\,c_1^c \mid c^c\,]\ \sigma^c\rangle \stackrel{\triangle}{=} \\ \qquad c_1^c \neq \mathsf{return}(s_1^c,n^c,m^c) \\ \quad \wedge\quad \langle s^a\ \rho^a\ [\,\mathsf{approx}() \mid c^a\,]\ \sigma^a\rangle \sqsupseteq \langle s^c\ \rho^c\ c^c\ \sigma^c\rangle \end{array}$$

5.8 Proof of Simulation

We will prove the simulation by showing for each concrete rule that it can be simulated by rules from the abstract semantics. When proving that a rule can be simulated we will assume that the abstract configuration abstracts the

concrete configuration, and proceed to show that after the application of the rules the abstraction is preserved.

The parts of the configurations will be superscripted with a for abstract and c for concrete, e.g., $s^a \sqsupseteq s^c$ stating that the abstracts value stack abstract the concrete value stack.

For each of the rules in the concrete semantics we will show three parts:

Monotonicity That the rule is monotonic.

Simulation Which rules are used to simulate the rule (in many cases the rule itself).

Approximation That the simulation holds if we use a rule that introduces an abstraction.

But first we will establish a few useful lemmas that we will use througout the proof.

5.8.1 Lemmas

We start by proving a few lemmas that will simplify the following proof of simulation:

Lemma 1

$$
\begin{array}{l}
(\forall \rho^a, \rho^c \in \text{Context}) \\
(\rho^a \sqsupseteq \rho^c \Rightarrow (\forall x \in \texttt{dom}(env^c))(\rho^a(x) \sqsupseteq \rho^c(x))) \\
\quad \texttt{where } \rho^c = \langle n^c, envmap^c, k^c \rangle \\
\qquad\quad env^c = \rho^c(n^c)
\end{array}
$$

Proof: Follows directly from the definitions of the abstraction relations. □

Lemma 2

$$
\begin{array}{l}
(\forall \rho^a, \rho^c \in \text{Context } p \in \text{Pattern } v \in \text{Value}^\top) \\
(env^c = \mathsf{match}(p, v) \neq \bot \wedge \rho^a \sqsupseteq \rho^c \Rightarrow \mathsf{subst}(p, env^a) \sqsupseteq_{(\rho^a, \rho^c)} v) \\
\quad \texttt{where } \rho^c = \langle n^c, envmap^c, k^c \rangle \\
\qquad\quad env^c = \rho^c(n^c) \\
\qquad\quad \rho^a = \langle n^a, envmap^a, k^a \rangle \\
\qquad\quad env^a = \rho^a(n^a)
\end{array}
$$

Proof: By induction over the structure of p.

Induction Hypothesis If $p_1, \ldots, p_i$ is of nesting depth no greater than j and p is either $x = p_1$, $[p_1 \mid p_2]$, $\{p_1, \ldots, p_i\}$ or $\langle p_1, \ldots, p_i \rangle$ then $env^c = \mathsf{match}(p, v) \wedge \rho^a \sqsupseteq \rho^c \Rightarrow \mathsf{subst}(p, env^a) \sqsupseteq_{(\rho^a, \rho^c)} v$

If p is x From the definitions of the functions we have that $\mathsf{match}(x, v) = env^c = \{x \mapsto v\}$ which means by the definition of $\sqsupseteq_{(\rho^a, \rho^c)}$ we have that if $env^a \sqsupseteq_{(\rho^a, \rho^c)} env^c$ then $env^a(x) \sqsupseteq_{(\rho^a, \rho^a)} v$ but then $\mathsf{subst}(x, env^a(x)) \sqsupseteq_{(\rho^a, \rho^a)} v$.

If p is $literal[v]$ From the definitions of the functions we have that $\mathsf{match}(literal[v], v) = \{\}$ which means $\mathsf{subst}(literal[v], \{\}) = v$ but that abstract v.

If p is $[p_1 \mid p_2]$ From the definitions of the functions we have that if $\rho^a = \mathsf{match}([p_1 \mid p_2], [v_1 \mid v_2])$ then $\mathsf{subst}([p_1 \mid p_2], \rho^a) = [\mathsf{subst}(p_1, \rho^a) \mid \mathsf{subst}(p_2, \rho^a)]$. From the induction hypothesis it follwos that $[\mathsf{subst}(p_1, \rho^a) \mid \mathsf{subst}(p_2, \rho^a)] \sqsupseteq_{(\rho^a, \rho^c)} [v_1 \mid v_2]$.

If p is $\{p_1, \ldots, p_i\}$ Analogous to the cons case.

If p is $\langle p_1, \ldots, p_i \rangle$ Analogous to the cons case.

□

Lemma 3

$$(\forall \rho^a, \rho^c \in \text{Context})(\rho^a \sqsupseteq \rho^c \Rightarrow \rho^a(\mathsf{current}(\rho^a)) \sqsupseteq \rho^c(\mathsf{current}(\rho^c)))$$

Proof: Follows directly from the definitions of the abstraction relations. □

Lemma 4

$$(\forall \sigma^a, \sigma^c \in \text{State}\ \forall \rho^a, \rho^c \in \text{Context})$$
$$(\sigma^a \sqsupseteq_{(\rho^a, \rho^c)} \sigma^c \Rightarrow \mathsf{module}(\sigma^a) = \mathsf{module}(\sigma^c))$$

Proof: Follows directly from the definitions of the abstraction relations. □

Lemma 5

$$(\forall \rho^a, \rho^c \in \text{Context}, c_1^a, \ldots, c_i^a, c_1^c, \ldots, c_i^c \in \text{Code})$$
$$(\rho^a \sqsupseteq \rho^c \wedge c_1^a \wedge s^a \sqsupseteq_{(\rho^a, \rho^c)} s^c \wedge m^a = m^c \wedge \sqsupseteq c_1^c \wedge \quad \wedge c_i^a \sqsupseteq c_i^c \Rightarrow$$
$$[c_1^a, \ldots, c_i^a, \mathsf{return}(s^a, n^a, m^a) \mid c^a] \sqsupseteq_{(\rho^a, \rho^c)} [c_1^c, \ldots, c_i^c, \mathsf{return}(s^c, n^c, m^c) \mid c^c])$$
where $n^a = \mathsf{current}(\rho^a)$
$n^c = \mathsf{current}(\rho^c)$

Proof: From Lemma 3 we have that $\rho^a(\mathsf{current}(\rho^a)) \sqsupseteq \rho^c(\mathsf{current}(\rho^c))$ and if $\mathsf{current}(\rho^a) = n^a$ then from the previous and the definition of $\sqsupseteq$ over code we have that $[\mathsf{return}(s^a, n^a, s^m) \mid c^a] \sqsupseteq_{(\rho^a,\rho^c)} [\mathsf{return}(s^c, n^c, m^c) \mid c^c]$. If now $c_1^a \sqsupseteq_{(\rho^a,\rho^c)} c_1^c \wedge \quad \wedge c_i^a \sqsupseteq_{(\rho^a,\rho^c)} c_i^c$, it follows from the definition of $\sqsupseteq$ over code that $[c_1^a, \ldots, c_i^a, \mathsf{return}(s^a, n^a, m^a) \mid c^a] \sqsupseteq_{(\rho^a,\rho^c)} [c_1^c, \ldots, c_i^c, \mathsf{return}(s^c, n^c, m^c) \mid c^c]$. □

Lemma 6

$$\left(\begin{array}{l}(\forall s^a, s^c \in \text{Stack } \rho^a, \rho^c \in \text{Context } c^a, c^c \in \text{Code} \\ \quad c_1^c, \ldots, c_i^c \in \text{Exprs} \cup \text{AuxExprs} \\ \quad \sigma^a, \sigma^c \in \text{State } n_1^c \in \text{EnvName } m_1^c \in \text{ModuleName}) \\ \left(\begin{array}{ll} & \langle s^a\ \rho^a\ [\mathsf{approx}() \mid c^a]\ \sigma^a \rangle \sqsupseteq \langle s^c\ \rho^c\ [c_i^c, \ldots, c_1^c \mid c^c]\ \sigma^c \rangle \\ \wedge & \{c_1^c, \ldots, c_i^c\} \cap \{\mathsf{return}(s^c, n_1^c, m_1^c)\} = \emptyset \end{array}\right) \Rightarrow \\ \langle s^a\ \rho^a\ [\mathsf{approx}() \mid c^a]\ \sigma^a \rangle \sqsupseteq \langle s^c\ \rho^c\ c^c\ \sigma^c \rangle \end{array}\right)$$

Proof: By induction over i.

Induction Hypothesis

$$\left(\begin{array}{l}(\forall s^a, s^c \in \text{Stack } \rho^a, \rho^c \in \text{Context } c^a, c^c \in \text{Code} \\ \quad c_1^c, \ldots, c_i^c \in \text{Exprs} \cup \text{AuxExprs} \\ \quad \sigma^a, \sigma^c \in \text{State } n_1^c \in \text{EnvName } m_1^c \in \text{ModuleName}) \\ \left(\begin{array}{ll} & \langle s^a\ \rho^a\ [\mathsf{approx}() \mid c^a]\ \sigma^a \rangle \sqsupseteq \langle s^c\ \rho^c\ [c_i^c, \ldots, c_1^c \mid c^c]\ \sigma^c \rangle \\ \wedge & \{c_1^c, \ldots, c_{i\ 1}^c\} \cap \{\mathsf{return}(s^c, n_1^c, m_1^c)\} = \emptyset \end{array}\right) \Rightarrow \\ \langle s^a\ \rho^a\ [\mathsf{approx}() \mid c^a]\ \sigma^a \rangle \sqsupseteq \langle s^c\ \rho^c\ c^c\ \sigma^c \rangle \end{array}\right)$$

If i is 0 We have to show that $\langle s^a\ \rho^a\ [\mathsf{approx}() \mid c^a]\ \sigma^a \rangle \sqsupseteq \langle s^c\ \rho^c\ c^c\ \sigma^c \rangle \Rightarrow \langle s^a\ \rho^a\ [\mathsf{approx}() \mid c^a]\ \sigma^a \rangle \sqsupseteq \langle s^c\ \rho^c\ c^c\ \sigma^c \rangle$ but that holds trivially.

If i is larger than zero We can assume that $\langle s^a\ \rho^a\ [\mathsf{approx}() \mid c^a]\ \sigma^a \rangle \sqsupseteq \langle s^c\ \rho^c\ [c_i^c, \ldots, c_1^c \mid c^c]\ \sigma^c \rangle$ and $\{c_1^c, \ldots, c_i^c\} \cap \{\mathsf{return}(s^c, n_1^c, m_1^c)\} = \emptyset$. But then it follows from the definition of abstraction over configurations that $\langle s^a\ \rho^a\ [\mathsf{approx}() \mid c^a]\ \sigma^a \rangle \sqsupseteq \langle s^c\ \rho^c\ [c_{i\ 1}^c, \ldots, c_1^c \mid c^c]\ \sigma^c \rangle$ and $\{c_1^c, \ldots, c_{i\ 1}^c\} \cap \{\mathsf{return}(s^c, n_1^c, m_1^c)\} = \emptyset$, but then using the induction hypothesis we can show that the lemma holds.

□

Lemma 7

$$
\begin{array}{l}
(\forall s^a, s^c \in \mathrm{Stack}\ \rho^a, \rho^c \in \mathrm{Context}\ c^a, c^c \in \mathrm{Code} \\
\quad c^c_1, \ldots, c^c_i \in \mathrm{Exprs} \cup \mathrm{AuxExprs} \\
\quad \sigma^a, \sigma^c \in \mathrm{State}\ n^c_1 \in \mathrm{EnvName}\ m^c_1 \in \mathrm{ModuleName}) \\
\left(\begin{array}{ll} & \langle s^a\ \rho^a\ [\mathsf{approx}() \mid c^a]\ \sigma^a \rangle \sqsupseteq \langle s^c\ \rho^c\ c^c\ \sigma^c \rangle \\ \wedge & \{c^c_1, \ldots, c^c_i\} \cap \{\mathsf{return}(s^c, n^c_1, m^c_1)\} = \emptyset \end{array} \right) \Rightarrow \\
\langle s^a\ \rho^a\ [\mathsf{approx}() \mid c^a]\ \sigma^a \rangle \sqsupseteq \langle s^c\ \rho^c\ [c^c_i, \ldots, c^c_1 \mid c^c]\ \sigma^c \rangle
\end{array}
$$

Proof: By induction over i.

Induction Hypothesis

$$
\begin{array}{l}
(\forall s^a, s^c \in \mathrm{Stack}\ \rho^a, \rho^c \in \mathrm{Context}\ c^a, c^c \in \mathrm{Code} \\
\quad c^c_1, \ldots, c^c_i \in \mathrm{Exprs} \cup \mathrm{AuxExprs} \\
\quad \sigma^a, \sigma^c \in \mathrm{State}\ n^c_1 \in \mathrm{EnvName}\ m^c_1 \in \mathrm{ModuleName}) \\
\left(\begin{array}{ll} & \langle s^a\ \rho^a\ [\mathsf{approx}() \mid c^a]\ \sigma^a \rangle \sqsupseteq \langle s^c\ \rho^c\ c^c\ \sigma^c \rangle \\ \wedge & \{c^c_1, \ldots, c^c_{i\ 1}\} \cap \{\mathsf{return}(s^c, n^c_1, m^c_1)\} = \emptyset \end{array} \right) \Rightarrow \\
\langle s^a\ \rho^a\ [\mathsf{approx}() \mid c^a]\ \sigma^a \rangle \sqsupseteq \langle s^c\ \rho^c\ [c^c_{i\ 1}, \ldots, c^c_1 \mid c^c]\ \sigma^c \rangle
\end{array}
$$

If i is 0 We have to show that $\langle s^a\ \rho^a\ [\mathsf{approx}() \mid c^a]\ \sigma^a \rangle \sqsupseteq \langle s^c\ \rho^c\ c^c\ \sigma^c \rangle \Rightarrow \langle s^a\ \rho^a\ [\mathsf{approx}() \mid c^a]\ \sigma^a \rangle \sqsupseteq \langle s^c\ \rho^c\ c^c\ \sigma^c \rangle$ but that holds trivially.

If i is larger than zero We can assume that $\langle s^a\ \rho^a\ [\mathsf{approx}() \mid c^a]\ \sigma^a \rangle \sqsupseteq \langle s^c\ \rho^c\ c^c\ \sigma^c \rangle$ and $\{c^c_1, \ldots, c^c_{i\ 1}\} \cap \{\mathsf{return}(s^c, n^c_1, m^c_1)\} = \emptyset$, then from the induction hypothesis we have that $\langle s^a\ \rho^a\ [\mathsf{approx}() \mid c^a]\ \sigma^a \rangle \sqsupseteq \langle s^c\ \rho^c\ [c^c_{i\ 1}, \ldots, c^c_1 \mid c^c]\ \sigma^c \rangle$. But from the definition of abstraction over configurations and that $c^c_i \neq \mathsf{return}(s^c, n^c_1, m^c_1)$ we can conclude that $\langle s^a\ \rho^a\ [\mathsf{approx}() \mid c^a]\ \sigma^a \rangle \sqsupseteq \langle s^c\ \rho^c\ [c^c_i, \ldots, c^c_1 \mid c^c]\ \sigma^c \rangle$.

□

5.9 Transition Rules

5.9.1 Stuttering

It follows trivially from Lemma 6 and Lemma 7 that for all rules (4.1) to (4.44), with the exception of (4.20), that when $c^a = [\mathsf{approx}() \mid c^a_t]$ the rule can be simulated by rule (5.1). We show how rule (4.20) is simulated in this case in Section 5.9.9.5.

$$
(5.1) \quad \langle s^a\ \rho^a\ [\mathsf{approx}() \mid c^a]\ \sigma^a \rangle \xrightarrow{\tau}_a \langle s^a\ \rho^a\ [\mathsf{approx}() \mid c^a]\ \sigma^a \rangle
$$

5.9.2 Normal Termination

$$(4.1)\quad \langle s\ \rho\ [\,]\ \sigma \rangle \xrightarrow{\texttt{exit(normal)}}_{c} \texttt{NIL}$$

Monotonicity
Rule (4.1) is trivially monotonic.

Simulation
Rule (4.1) is simulated by itself and monotonicity ensures that the simulation preserves abstraction.

Approximation
It follows from the definition of $\sqsupseteq$ over the code stack that: if the abstraction relation holds the code stack of the abstract state is $[\,]$, and consequently can not be approximated.

5.9.3 Simple Expressions

5.9.3.1 Variable

$$(4.2)\quad \langle s\ \rho\ [[\![x]\!]\mid c]\ \sigma \rangle \xrightarrow{\tau}_{c} \langle [\rho(x)\mid s]\ \rho\ c\ \sigma \rangle$$

Monotonicity
That the rule is monotonic follows from Lemma 1 and the definition of $\sqsupseteq$ over configuration, values, code and value stacks.

Simulation
Rule (4.2) is simulated by itself and monotonicity ensures that the simulation preserves abstraction.

Approximation
In the case where the rule (4.2) is approximated by rule (5.1) it follows from Lemma 6 that abstraction is preserved.

5.9.3.2 Literal

$$(4.3)\quad \langle s\ \rho\ [[\![\mathit{literal}[l]]\!]\mid c]\ \sigma \rangle \xrightarrow{\tau}_{c} \langle [l\mid s]\ \rho\ c\ \sigma \rangle$$

Monotonicity
That the rule is monotonic follows from Lemma 1 and the definition of $\sqsupseteq$ over configuration, values, code and value stacks.

Simulation
Rule (4.3) is simulated by itself and monotonicity ensures that the simulation preserves abstraction.

Approximation
In the case where the rule (4.3) is approximated by rule (5.1) it follows from Lemma 6 that abstraction is preserved.

5.9.4 Compound Expressions

5.9.4.1 Tuples

$$(4.4)\quad \langle\, s\ \rho\ [\,[\![\,\{x_1,\ldots,x_i\}\,]\!]\mid c\,]\ \sigma\,\rangle\ \xrightarrow{\tau}_c \langle\, [\,\{\rho\,(x_1),\ldots,\rho\,(x_i)\}\mid s\,]\ \rho\ c\ \sigma\,\rangle$$

Monotonicity
That the rule is monotonic follows from Lemma 1 and the definition of $\sqsupseteq$ over configuration, values, code and value stacks.

Simulation
Rule (4.4) is simulated by itself and monotonicity ensures that the simulation preserves abstraction.

Approximation
In the case where the rule (4.4) is approximated by rule (5.1) it follows from Lemma 6 that abstraction is preserved.

5.9.4.2 Lists

$$(4.5)\quad \langle\, s\ \rho\ [\,[\![\,[x_1\mid x_2\,]\,]\!]\mid c\,]\ \sigma\,\rangle\ \xrightarrow{\tau}_c \langle\, [\,[\,\rho\,(x_1)\mid\rho\,(x_2)\,]\mid s\,]\ \rho\ c\ \sigma\,\rangle$$

Monotonicity
That the rule is monotonic follows from Lemma 1 and the definition of $\sqsupseteq$ over configuration, values, code and value stacks.

Simulation
Rule (4.5) is simulated by itself and monotonicity ensures that the simulation preserves abstraction.

Approximation
In the case where the rule (4.5) is approximated by rule (5.1) it follows from Lemma 6 that abstraction is preserved.

5.9.4.3 Values

$$(4.6)\quad \langle\, s\ \rho\ [\,[\![\langle x_1,\ldots,x_i\rangle]\!]\mid c\,]\ \sigma\,\rangle \xrightarrow{\tau}_c \langle\, [\,\langle \rho(x_1),\ldots,\rho(x_i)\rangle \mid s\,]\ \rho\ c\ \sigma\,\rangle$$

Monotonicity
That the rule is monotonic follows from Lemma 1 and the definition of $\sqsupseteq$ over configuration, values, code and value stacks.

Simulation
Rule (4.6) is simulated by itself and monotonicity ensures that the simulation preserves abstraction.

Approximation
In the case where the rule (4.6) is approximated by rule (5.1) it follows from Lemma 6 that abstraction is preserved.

5.9.5 Funs

$$(4.7)\quad \langle\, s\ \rho\ [\,[\![\mathtt{fun}(x_1,\ldots,x_i)=e]\!]\mid c\,]\ \sigma\,\rangle \xrightarrow{\tau}_c \langle\, [\,v \mid s\,]\ \rho\ c\ \sigma\,\rangle$$

$$\begin{aligned}\textbf{where}\ & n = \mathsf{current}(\rho)\\ & m = \mathsf{module}(\sigma)\\ & v = \langle\langle [\![\mathtt{fun}(x_1,\ldots,x_n)=e]\!], n, m\rangle\rangle\end{aligned}$$

Monotonicity
That the rule is monotonic follows from Lemma 3 and Lemma 4 and the definition of $\sqsupseteq$ over configuration, values, code and value stacks.

Simulation
Rule (4.7) is simulated by itself and monotonicity ensures that the simulation preserves abstraction.

Approximation
In the case where the rule (4.7) is approximated by rule (5.1) it follows from Lemma 6 that abstraction is preserved.

5.9.6 Binding Expressions

5.9.6.1 Let

$$(4.8)\quad \begin{array}{l}\langle\, s\ \rho\ [\![\, \texttt{let}\ \langle x_1,\ldots,x_i\rangle = \langle e_1,\ldots,e_i\rangle\ \texttt{in}\ e\,]\!] \mid c\,]\ \sigma\,\rangle \xrightarrow{\tau}_C \\ \langle\, s\ \rho\ [\, e_1,\ldots,e_i,\mathsf{def}(x_1,\ldots,x_i),e,\mathsf{restore}(n) \mid c\,]\ \sigma\,\rangle \\ \textbf{where}\ n^c = \mathsf{current}\,(\,\rho^c\,)\end{array}$$

Monotonicity
That the rule is monotonic follows from Lemma 5 and the definition of $\sqsupseteq$ over configuration and code stack.

Simulation
Rule (4.8) is simulated by itself and monotonicity ensures that the simulation preserves abstraction.

Approximation
In the case where the rule (4.8) is approximated by rule (5.1) it follows from Lemma 6 and Lemma 7 that abstraction is preserved.

5.9.6.2 The def intermediate

$$(4.9)\quad \langle\,[\,v_i,\ldots,v_1 \mid s\,]\ \rho\ [\,\mathsf{def}(x_1,\ldots,x_i) \mid c\,]\ \sigma\,\rangle \xrightarrow{\tau}_C \langle\, s\ \rho_1\ c\ \sigma\,\rangle$$
$$\textbf{where}\ \rho_1 = \mathsf{bind}(\langle x_1,\ldots,x_i\rangle,\langle v_1,\ldots,v_i\rangle,\rho\,)$$

Monotonicity
That the rule is monotonic follows from that bind is monotonic and the definition of $\sqsupseteq$ over configurations and contexts.

Simulation
Rule (4.9) is simulated by itself and monotonicity ensures that the simulation preserves abstraction.

Approximation
In the case where the rule (4.9) is approximated by rule (5.1) it follows from Lemma 6 and that bind is monotonic that abstraction is preserved.

5.9.6.3 The restore intermediate

$$(4.10)\quad \langle\, s\ \rho\ [\,\mathsf{restore}(n) \mid c\,]\ \sigma\,\rangle \xrightarrow{\tau}_C \langle\, s\ \rho_1\ c\ \sigma\,\rangle$$
$$\textbf{where}\ \rho_1 = \mathsf{set}\,(\,n,\rho\,)$$

Monotonicity
That the rule is monotonic follows from the definition of set and the definition of $\sqsupseteq$ over configurations, contexts and code stacks.

Simulation
Rule (4.10) is simulated by itself and monotonicity ensures that the simulation preserves abstraction.

Approximation
In the case where the rule (4.10) is approximated by rule (5.1) it follows from Lemma 6 and Lemma 7 that abstraction is preserved.

5.9.6.4 LetRec

$$
\begin{array}{ll}
(4.11) & \langle\, s\ \rho\ [\,[\![\, \texttt{letrec}\ x_1^f = e_1^f\ \ldots\ x_i^f = e_i^f\ \texttt{in}\ e \,]\!]\mid c\,]\ \sigma\,\rangle\ \xrightarrow{\tau}_c \\
& \langle\, s\ \rho_1\ [\,e, \mathsf{restore}(n)\mid c\,]\ \sigma\,\rangle \\
& \textbf{where}\ n = \mathsf{current}\,(\,\rho\,) \\
& \qquad\quad m = \mathsf{module}\,(\,\sigma\,) \\
& \qquad\quad \rho_1 = \mathsf{bindf}\left(\langle x_1^f,\ldots,x_i^f\rangle, \langle e_1^f,\ldots,e_i^f\rangle, \rho, m\right)
\end{array}
$$

Monotonicity
That the rule is monotonic follows from that the functions current, module and bindf are monotonic and the definition of $\sqsupseteq$ over configurations, contexts and code stacks.

Simulation
Rule (4.11) is simulated by itself and monotonicity ensures that the simulation preserves abstraction.

Approximation
In the case where the rule (4.11) is approximated by rule (5.1) it follows from Lemma 6 and Lemma 7 and that the functions current, module and bindf are monotonic that abstraction is preserved.

5.9.7 Sequencing Expressions

$$
(4.12)\quad \langle\, s\ \rho\ [\,[\![\, \texttt{do}\ x_1\ x_2 \,]\!]\mid c\,]\ \sigma\,\rangle\ \xrightarrow{\tau}_c \langle\, [\,\rho\,(x_2)\mid s\,]\ \rho\ c\ \sigma\,\rangle
$$

Monotonicity

That the rule is monotonic follows from Lemma 1 and the definition of $\sqsupseteq$ over configuration, values, code and value stacks.

Simulation

Rule (4.12) is simulated by itself and monotonicity ensures that the simulation preserves abstraction.

Approximation

In the case where the rule (4.12) is approximated by rule (5.1) it follows from Lemma 6 that abstraction is preserved.

5.9.8 Conditional Expressions

5.9.8.1 First clause matches

$$(4.13)\quad \frac{\bot \notin \mathsf{match}(p, \langle \rho(x_1), \ldots, \rho(x_i) \rangle)}{\begin{array}{l}\langle s\ \rho\ [\![\texttt{case}\ \langle x_1, \ldots, x_i \rangle\ \texttt{of}\ \mathit{clause}\ \mathit{clauses}\ \texttt{end}]\!] \mid c]\ \sigma \rangle \xrightarrow{\tau}_c \\ \langle s\ \rho_1\ [g, \mathsf{if}(c_1, c_2) \mid c]\ \sigma \rangle \\ \textbf{where}\ [\![p\ \texttt{when}\ g\ \texttt{->}\ e]\!] = \mathit{clause} \\ \qquad \rho_1 = \mathsf{update}(\mathsf{match}(p, \langle \rho(x_1), \ldots, \rho(x_i) \rangle), \rho) \\ \qquad n = \mathsf{current}(\rho) \\ \qquad c_1 = [e, \mathsf{restore}(n)] \\ \qquad c_2 = [\mathsf{restore}(n), [\![\texttt{case}\ \langle x_1, \ldots, x_i \rangle\ \texttt{of}\ \mathit{clauses}\ \texttt{end}]\!]]\end{array}}$$

Monotonicity

That the rule is monotonic follows from that the functions current, update and match are monotonic and the definition of $\sqsupseteq$ over configurations, contexts and code stacks.

Simulation

Rule (4.13) is simulated by itself and monotonicity ensures that the simulation preserves abstraction.

Approximation

In the case where the rule (4.13) is approximated by rule (5.1) it follows from Lemma 6 and Lemma 7 and that the functions current, update and match are monotonic that abstraction is preserved.

5.9.8.2 First clause does not match

$$(4.14)\quad \frac{\bot \in \mathsf{match}(p, \langle \rho(x_1), \ldots, \rho(x_i) \rangle)}{\begin{array}{l}\langle s\ \rho\ [\, [\![\texttt{case}\ \langle x_1, \ldots, x_i \rangle\ \texttt{of}\ \mathit{clause}\ \mathit{clauses}\ \texttt{end}]\!] \mid c \,]\ \sigma \rangle \xrightarrow{\tau}_c \\ \langle s\ \rho\ [\, [\![\texttt{case}\ \langle x_1, \ldots, x_i \rangle\ \texttt{of}\ \mathit{clauses}\ \texttt{end}]\!] \mid c \,]\ \sigma \rangle \\ \textbf{where}\ [\![p\ \texttt{when}\ g\ \texttt{->}\ e]\!] = \mathit{clause}\end{array}}$$

Monotonicity
That the rule is monotonic follows from that the function match is monotonic and the definition of $\sqsupseteq$ over configurations and code stacks.

Simulation
Rule (4.14) is simulated by rule (5.4), and monotonicity ensures that the simulation preserves abstraction since the only difference is that rule (5.4) is applicable in more configurations.

$$(5.4)\quad \frac{\bot \in \mathsf{match}(p, \langle v_1, \ldots, v_i \rangle) \vee \mathsf{contain\_top}(\langle v_1, \ldots, v_i \rangle)}{\begin{array}{l}\langle s\ \rho\ [\, [\![\texttt{case}\ \langle x_1\ \ldots, x_i \rangle\ \texttt{of}\ \mathit{clause}\ \mathit{clauses}\ \texttt{end}]\!] \mid c \,]\ \sigma \rangle \xrightarrow{\tau}_a \\ \langle s\ \rho\ [\, [\![\texttt{case}\ \langle x_1\ \ldots, x_i \rangle\ \texttt{of}\ \mathit{clauses}\ \texttt{end}]\!] \mid c \,]\ \sigma \rangle \\ \textbf{where}\ [\![p\ \texttt{when}\ g\ \texttt{->}\ e]\!] = \mathit{clause} \\ \qquad \langle v_1, \ldots, v_i \rangle = \langle \rho(x_1) \ldots, \rho(x_i) \rangle\end{array}}$$

Approximation
In the case where the rule (4.14) is approximated by rule (5.1) it follows from Lemma 6 that abstraction is preserved.

5.9.8.3 The if intermediate

We will show monotonicity and simulation for the first rule rule (4.15), the proof for the second rule if (4.16) is analogous.

$$(4.15)\quad \begin{array}{l}\langle [\texttt{'true'} \mid s]\ \rho\ [\mathsf{if}([c_1, \ldots, c_i], c_0) \mid c]\ \sigma \rangle \xrightarrow{\tau}_c \\ \langle s\ \rho\ [c_1, \ldots, c_i \mid c]\ \sigma \rangle\end{array}$$

Monotonicity
That the rule is monotonic follows from the definition of $\sqsupseteq$ over configurations, value and code stacks.

Simulation

Rule (4.15) is simulated by itself or by rule (5.5). In the case where it is simulated by itself it follows from monotonicity that the simulation preserves abstraction as does the case where the rule (4.15) is simulated by rule (5.5) since they have identical right hand sides.

$$(5.5)\quad \langle\, [\,\top \mid s\,]\,\rho\ [\,\mathsf{if}([c_1,\ldots,c_i],c')\mid c\,]\,\sigma\,\rangle \xrightarrow{\tau}_a \langle s\,\rho\ [\,c_1,\ldots,c_i \mid c\,]\,\sigma\,\rangle$$

Approximation

In the case where the rule (4.15) is approximated by rule (5.1) it follows from Lemma 6 and Lemma 7 that abstraction is preserved.

5.9.9 Function Expressions

5.9.9.1 Apply

For the concrete rules there are three type of values that the variable x^f can be bound to: an atom, a closure, any other value (except approximation since they cannot occur in the concrete semantics). These three cases are handled by the three application rules of the concrete semantics: rule (4.17), rule (4.18) and rule (4.19). In the abstract semantics there exist a fourth case where the variable x^f is bound to an approximation, this is handled by rule (5.7). The rule (5.7) is also the rule responsible for abstracting a function call and we show that any of the rules: rule (4.17), rule (4.18), and rule (4.19) can be simulated by rule (5.7).

5.9.9.2 Application of an atom

$$(4.17)\quad \frac{\rho\,(x^f) \in \text{Atom}}{\begin{array}{l}\langle s\,\rho\ [\,[\![\,\mathtt{apply}\ x^f\,(x_1,\ldots,x_i)\,]\!]\mid c\,]\,\sigma\,\rangle \xrightarrow{\mu}_c \\ \langle\,[\,]\,\rho'\ [e,\mathsf{return}\,(s,n,m)\mid c\,]\,\sigma\,\rangle \\ \textbf{where}\ n = \mathsf{current}\,(\rho) \\ \qquad m = \mathsf{module}\,(\sigma) \\ \qquad \langle\langle\,[\![\,\mathtt{fun}(x'_1,\ldots,x'_i) = e\,]\!],0,m^c\,\rangle\rangle = \mathsf{func}\,\big(m,\rho\,(x^f),i,\sigma\big) \\ \qquad \rho' = \mathsf{bind}(\langle x'_1,\ldots,x'_i\rangle,\langle\rho\,(x_1),\ldots,\rho\,(x_i)\rangle,\mathsf{set}\,(0,\rho)) \\ \qquad \mu = m : \rho\,(x^f)\,(\rho\,(x_1),\ldots,\rho\,(x_i))\end{array}}$$

Monotonicity

That the rule is monotonic follows from that the functions current, module, func, set, and bind are monotonic, and the definition of $\sqsupseteq$ over configurations, contexts, value and code stacks.

Simulation

Rule (4.17) is simulated by itself or by rule (5.7). In the case where it is simulated by itself it follows from monotonicity that the simulation preserves abstraction. In the case where the rule (4.17) is simulated by rule (5.7) follows by Lemma 6 and the definition of $\sqsupseteq$ over configurations.

$$
\begin{array}{ll}
(5.7) & \langle\, s\ \rho\ [\,[\![\,\mathtt{apply}\ x^f\,(x_1,\ldots,x_i)\,]\!]\mid c\,]\ \sigma\,\rangle \overset{\top}{\rightarrow} a \\
& \langle\, s\ \rho\ [\,\mathsf{approx}()\mid c\,]\ \sigma_1\,\rangle \\
& \textbf{where}\ \sigma_1 = \mathsf{publish}(\top,\top,\sigma)
\end{array}
$$

Approximation

In the case where the rule (4.17) is approximated by rule (5.1) it follows from the definition of $\sqsupseteq$ over configurations and Lemma 7 that abstraction is preserved.

5.9.9.3 Application of a Closure

Without the requirement concerning the `func` function the proof is identical to that of rule (4.17).

$$
\begin{array}{ll}
(4.18) & \dfrac{\rho\,(x^f) = \langle\langle\,[\![\,\mathtt{fun}(x'_1,\ldots,x'_i) = e\,]\!], n', m\,\rangle\rangle}{\langle\, s\ \rho\ [\,[\![\,\mathtt{apply}\ x^f\,(x_1,\ldots,x_i)\,]\!]\mid c\,]\ \sigma\,\rangle \overset{\mu}{\rightarrow} c} \\
& \langle\,[\,]\ \rho'\ [e, \mathsf{return}\,(s,n,m)\mid c\,]\ \sigma\,\rangle \\
& \textbf{where}\ n = \mathsf{current}\,(\rho\,) \\
& \qquad m = \mathsf{module}\,(\sigma\,) \\
& \qquad \rho' = \mathsf{bind}(\langle x'_1,\ldots,x'_i\rangle, \langle\rho\,(x_1),\ldots,\rho\,(x_i)\rangle, \mathsf{set}\,(n',\rho\,)) \\
& \qquad \mu = m : \rho\,(x^f)\,(\rho\,(x_1),\ldots,\rho\,(x_i)\,)
\end{array}
$$

5.9.9.4 Application type mismatch

The proof is analogous to that of rule (4.17).

$$(4.19)\quad \frac{\rho(x^f) \notin \mathrm{Atom} \cup \mathrm{Closure} \cup \{\top_{\mathrm{Value}}\}}{\begin{array}{l}\langle s\ \rho\ [[\![\texttt{apply}\ x^f\ (x_1,\ldots,x_i)]\!]\mid c]\ \sigma\rangle \xrightarrow{\mu}_c \\ \langle [\,]\ \rho'\ [[\![\texttt{primop}\ atom[\texttt{'raise'}](x')]\!], \mathsf{return}\,(s,n,m)\mid c]\ \sigma\rangle \\ \textbf{where}\ n = \mathsf{current}\,(\rho) \\ \qquad m = \mathsf{module}\,(\sigma) \\ \qquad d_1 = \langle s,n,m\rangle \\ \qquad \rho' = \mathsf{bind}(\langle x'\rangle, \langle\,\texttt{'badarg'}\,\rangle, \rho) \\ \qquad \mu = m : \rho(x^f)\,(\rho(x_1),\ldots,\rho(x_i))\end{array}}$$

5.9.9.5 The return intermediate

$$(4.20)\quad \langle [v]\ \rho\ [\mathsf{return}\,(s,n,m)\mid c]\ \sigma\rangle \xrightarrow{\tau}_c \langle [v\mid s]\ \rho'\ c\ \sigma'\rangle$$

$$\textbf{where}\ \rho' = \mathsf{set}\,(n,\rho)$$
$$\sigma' = \mathsf{set\_module}\,(m,\sigma)$$

Monotonicity

That the rule is monotonic follows from that the functions set and set_module are monotonic, and the definition of $\sqsupseteq$ over configurations, contexts, states, value and code stacks.

Simulation

Rule (4.20) is simulated by itself and monotonicity ensures that the simulation preserves abstraction.

Approximation

The approximation of the concrete rule for the return intermediate rule (4.20) requires two abstract rules, rule (5.1) and rule (5.2). The two abstract rules correspond to two possible situations when a call in the concrete semantics has been approximated. The two situations are:

- The return approximated is one belonging to a call at a deeper call level (or in other words a nested call) than that of the approximated call. This case is approximated by rule (5.1).
- The return approximated is a normal return that corresponds to the approximated call. This case is approximated by rule (5.2).

In the case where the return simulated is one belonging to a call at a deeper call level than that of the approximated call it is approximated by rule (5.1).

It follows from the monotonicity of the functions set and set_module, and the definition of $\sqsupseteq$ over configurations, that abstraction is preserved.

$$(5.1) \quad \langle s\ \rho\ [\mathsf{approx}() \mid c]\ \sigma \rangle \xrightarrow{\tau}_a \langle s\ \rho\ [\mathsf{approx}() \mid c]\ \sigma \rangle$$

In the case where the return simulated is a normal return that corresponds to the approximated call it is approximated by rule (5.2). Then follows from that the functions set and set_module are monotonic, and the definition of $\sqsupseteq$ over configurations, values, states, contexts, value and code stacks, that abstraction is preserved.

$$(5.2) \quad \langle s\ \rho\ [\mathsf{approx}() \mid c]\ \sigma \rangle \xrightarrow{\tau}_a \langle [\top \mid s]\ \rho\ c\ \sigma \rangle$$

5.9.9.6 Call

The proof for rule (4.21), rule (4.22), rule (4.23) and rule (4.24) follows from the proof of rule (4.17) and the proof for rule (4.25) follows from the proof of rule (4.19). That the proofs for the apply and call rules should resemble (or rather follow trivially from) stems from the fact that the apply construct is merely a special case of the call construct.

5.9.9.7 Raise

$$(4.26) \quad \begin{array}{l} \langle s\ \rho\ [[\![\mathtt{primop}\ atom[\mathtt{'raise'}](x)]\!] \mid c]\ \sigma \rangle \xrightarrow{\mu}_c \\ \langle [\,]\ \rho\ [\mathsf{exception}(\mathtt{'EXIT'}, \rho(x)) \mid c]\ \sigma \rangle \\ \textbf{where}\ \mu = \mathtt{primop\ raise}(\rho(x)) \end{array}$$

Monotonicity

That the rule is monotonic follows from definition of $\sqsupseteq$ over configuration and value and code stacks.

Simulation

Rule (4.26) is simulated by itself and monotonicity ensures that the simulation preserves abstraction.

Approximation

In the case where the rule (4.26) is approximated by rule (5.1) it follows from Lemma 6 and Lemma 7 that abstraction is preserved.

5.9.9.8 Throw

The proof is analogous to that of rule (4.26)

$$
\begin{array}{ll}
(4.27) & \langle\, s\ \rho\ [\,[\![\texttt{primop}\ atom[\texttt{'throw'}](x)]\!]\mid c\,]\ \sigma\,\rangle \xrightarrow{\mu}_c \\
& \langle\, [\,]\ \rho\ [\,\mathsf{exception}\,(\texttt{'THROW'},\rho(x))\mid c\,]\ \sigma\,\rangle \\
& \textbf{where}\ \mu = \texttt{primop throw}\,(\rho(x))
\end{array}
$$

5.9.9.9 Read

In order for the read and publish rules to preserve abstraction the functions read and publish must preserve abstraction. This is achieved by publish becoming an identity operation once a key/value pair has been published where the key contains an approximation, and that once such a pair has been published all read operations will result in an approximation.

$$
\begin{array}{ll}
(4.28) & \langle\, s\ \rho\ [\,[\![\texttt{primop}\ atom[\texttt{'read'}](x_1,x_2)]\!]\mid c\,]\ \sigma\,\rangle \xrightarrow{\mu}_c \\
& \langle\, [\,v\mid s\,]\ \rho\ c\ \sigma\,\rangle \\
& \textbf{where}\ v = \mathsf{read}\,(\rho(x_1),\rho(x_2),\mathsf{effects}\,(\sigma\,)) \\
& \qquad\quad \mu = \texttt{primop 'read'}\ (\rho(x_1),\rho(x_2))
\end{array}
$$

Monotonicity

That the rule is monotonic follows from that the functions read and effects are monotonic and the definition of $\sqsupseteq$ over configuration and value and code stacks.

Simulation

Rule (4.28) is simulated by itself and monotonicity ensures that the simulation preserves abstraction.

Approximation

In the case where the rule (4.28) is approximated by rule (5.1) it follows from Lemma 6 and that the functions read and effects are monotonic that abstraction is preserved.

5.9.9.10 Publish

$$
\begin{array}{ll}
(4.29) & \langle\, s\ \rho\ [\,[\![\texttt{primop}\ atom[\texttt{'publish'}](x_1,x_2)]\!]\mid c\,]\ \sigma\,\rangle \xrightarrow{\mu}_c \\
& \langle\, [\texttt{'true'}\mid s]\ \rho\ c\ \sigma_1\,\rangle \\
& \textbf{where}\ \sigma_1 = \mathsf{publish}\,(\rho(x_1),\rho(x_2),\sigma\,) \\
& \qquad\quad \mu = \texttt{primop publish}\,(\rho(x_1),\rho(x_2))
\end{array}
$$

Monotonicity
That the rule is monotonic follows from that the function publish is monotonic and the definition of $\sqsupseteq$ over configuration and value and code stacks.

Simulation
Rule (4.29) is simulated by itself and monotonicity ensures that the simulation preserves abstraction.

Approximation
In the case where the rule (4.29) is approximated by rule (5.1) it follows from Lemma 6 and that the function publish is monotonic that abstraction is preserved.

5.9.9.11 General PrimOp Rules

For the two general primop rules it is necessary to show that each definition of the function primop that is used is monotonic.

$$(4.30)\quad \frac{a \notin \{\texttt{'raise'},\texttt{'throw'},\texttt{'read'},\texttt{'publish'}\} \wedge v \neq \bot}{\begin{array}{l}\langle s\ \rho\ [[\![\texttt{primop}\ atom[a](x_1,\ldots,x_i)]\!]\ |\ c]\ \sigma\rangle \xrightarrow{\mu}_c \\ \langle [v\ |\ s]\ \rho\ c\ \sigma'\rangle \\ \textbf{where}\ \{v,\sigma'\} = \mathsf{primop}(a,\langle \rho(x_1),\ldots,\rho(x_i)\rangle,\sigma) \\ \quad \mu = \texttt{primop}\ a(\rho(x_1),\ldots,\rho(x_i))\end{array}}$$

Monotonicity
That the rule is monotonic follows from that the function primop is monotonic and the definition of $\sqsupseteq$ over configuration, states and value and code stacks.

Simulation
Rule (4.30) is simulated by itself and monotonicity ensures that the simulation preserves abstraction.

Approximation
In the case where the rule (4.30) is approximated by rule (5.1) it follows from Lemma 6 and that the function primop is monotonic that abstraction is preserved.

$$(4.31)\quad \frac{a \notin \{\texttt{'raise'},\texttt{'throw'},\texttt{'read'},\texttt{'publish'}\} \wedge v = \bot}{\begin{array}{l}\langle s\ \rho\ [[\![\texttt{primop}\ atom[a](x_1,\ldots,x_i)]\!] \mid c]\ \sigma\rangle \xrightarrow{\mu}_c \\ \langle s\ \rho'\ [[\![\texttt{primop}\ atom[\texttt{'raise'}](x')]\!], \mathsf{return}() \mid c]\ \sigma\rangle \\ \textbf{where}\ \{v,\sigma'\} = \mathsf{primop}(a, \langle \rho(x_1),\ldots,\rho(x_i)\rangle, \sigma) \\ \qquad n = \mathsf{current}(\rho) \\ \qquad \rho' = \mathsf{bind}(\langle x'\rangle, \langle\ \texttt{'badarg'}\ \rangle, \rho) \\ \qquad \mu = \texttt{primop}\ a(\rho(x_1),\ldots,\rho(x_i))\end{array}}$$

Monotonicity

That the rule is monotonic follows from that the function primop is monotonic and the definition of $\sqsupseteq$ over configuration and value and code stacks.

Simulation

Rule (4.31) is simulated by itself and monotonicity ensures that the simulation preserves abstraction.

Approximation

In the case where the rule (4.31) is approximated by rule (5.1) it follows from Lemma 6 and Lemma 7 and that the function primop is monotonic that abstraction is preserved.

5.9.10 Exception

$$(4.32)\quad \frac{c_1 \notin \{\mathsf{catch}(x_1,x_2,e), \mathsf{return}(s,n,m)\}}{\begin{array}{l}\langle [\,]\ \rho\ [\mathsf{exception}(v_1,v_2), c_1 \mid c]\ \sigma\rangle \xrightarrow{\tau}_c \\ \langle [\,]\ \rho\ [\mathsf{exception}(v_1,v_2) \mid c]\ \sigma\rangle\end{array}}$$

Monotonicity

That the rule is monotonic follows from the definition of $\sqsupseteq$ over configuration and code stack.

Simulation

Rule (4.32) is simulated by itself and monotonicity ensures that the simulation preserves abstraction.

Approximation
In the case where the rule (4.32) is approximated by rule (5.1) it follows from Lemma 6 and Lemma 7 that abstraction is preserved.

5.9.10.1 Return

$$(4.33)\quad \begin{array}{l} \langle\, [\,]\ \rho\ [\mathsf{exception}(v_1,v_2),\mathsf{return}\,(s,n,m)\ \mid c]\ \sigma\,\rangle \xrightarrow{\tau}_c \\ \langle\, [\,]\ \rho'\ [\mathsf{exception}(v_1,v_2)\mid c]\ \sigma'\,\rangle \\ \textbf{where}\ \rho' = \mathsf{set}\,(n,\rho\,) \\ \qquad\quad \sigma' = \mathsf{set\_module}\,(m,\sigma\,) \end{array}$$

Monotonicity
That the rule is monotonic follows from the definition of $\sqsupseteq$ over configuration and code stacks.

Simulation
Rule (4.33) is simulated by itself and monotonicity ensures that the simulation preserves abstraction.

Approximation
The approximation of the concrete rule for the exception intermediate rule (4.33) requires two abstract rules, rule (5.1) and rule (5.3). The two abstract rules correspond to two possible situations when a call in the concrete semantics has been approximated. The two situations are:

- The return approximated is one belonging to a call at a deeper call level (or in other words a nested call) than that of the approximated call. This case is approximated by rule (5.1).
- The return approximated is the return that corresponds to the approximated call. This case is approximated by rule (5.3).

In the case where the return simulated is one belonging to a call at a deeper call level than that of the approximated 'call it is approximated by rule (5.1). Then follows from that the functions set and set_module are monotonic, and the definition of $\sqsupseteq$ over configurations that abstraction is preserved.

$$(5.1)\quad \langle s\ \rho\ [\mathsf{approx}()\mid c]\ \sigma\,\rangle \xrightarrow{\tau}_a \langle s\ \rho\ [\mathsf{approx}()\mid c]\ \sigma\,\rangle$$

In the case where the return simulated is the return that corresponds to the approximated call it is approximated by rule (5.3). Then follows from that the functions set and set_module are monotonic, and the definition of $\sqsupseteq$ over configurations, values, states, contexts, value and code stacks, that abstraction

is preserved.

$$(5.3)\quad \langle\, s\ \rho\ [\,\mathsf{approx}()\mid c\,]\ \sigma\,\rangle \xrightarrow{\tau}_a \langle\, [\,]\ \rho\ [\,\mathsf{exception}(\top,\top)\mid c\,]\ \sigma\,\rangle$$

5.9.10.2 Abnormal termination

$$(4.34)\quad \langle\, s\ \rho\ [\,\mathsf{exception}(v_1,v_2)\,]\ \sigma\,\rangle \xrightarrow{\texttt{exit(fail)}}_c \texttt{NIL}$$

Monotonicity
Rule (4.1) is trivially monotonic.

Simulation
Rule (4.34) is simulated by itself and monotonicity ensures that the simulation preserves abstraction.

Approximation
It follows from the definition of $\sqsupseteq$ over the code stack that: if the abstraction relation holds the code stack of the abstract state is $[\,]$, and consequently can not be approximated.

5.9.11 Exception Handling

5.9.11.1 Try–Catch

$$(4.35)\quad \begin{array}{l} \langle\, s\ \rho\ [\,[\![\,\texttt{try}\ e_1\ \texttt{catch}(x_1,x_2)\ \texttt{->}\ e_2\,]\!]\mid c\,]\ \sigma\,\rangle \xrightarrow{\tau}_c \\ \langle\, [\,]\ \rho\ [\,e_1,\mathsf{catch}(x_1,x_2,e_2),\mathsf{restore}(s,n,m)\mid c\,]\ \sigma\,\rangle \\ \textbf{where}\ n=\mathsf{current}(\rho) \\ \qquad\quad m=\mathsf{module}(\sigma) \end{array}$$

Monotonicity
That the rule is monotonic follows from Lemma 5, that the functions current and module are monotonic, and the definition of $\sqsupseteq$ over configurations, value and code stacks.

Simulation
Rule (4.35) is simulated by itself and monotonicity ensures that the simulation preserves abstraction.

Approximation
In the case where the rule (4.35) is approximated by rule (5.1) it follows from Lemma 6 and Lemma 7, that the functions current and module are monotonic and the definition of $\sqsupseteq$ over configurations, value and code stacks that abstraction is preserved.

5.9.11.2 The catch intermediate: normal execution

$$\langle\, s\ \rho\ [\,\mathsf{catch}(x_1,x_2,e,s_1,n,m) \mid c\,]\ \sigma\,\rangle \xrightarrow{\tau}_c \langle\, s\ \rho\ c\ \sigma\,\rangle \tag{4.36}$$

Monotonicity
That the rule is monotonic follows from the definition of $\sqsupseteq$ over configurations and code stack.

Simulation
Rule (4.36) is simulated by itself and monotonicity ensures that the simulation preserves abstraction.

Approximation
In the case where the rule (4.36) is approximated by rule (5.1) it follows from the definition of $\sqsupseteq$ over configurations and code stack that abstraction is preserved.

5.9.11.3 The catch intermediate: exception

$$\begin{aligned}
&\langle\,[\,]\ \rho\ [\,\mathsf{exception}(v_1,v_2),\mathsf{catch}(x_1,x_2,e,s,n,m) \mid c\,]\ \sigma\,\rangle \xrightarrow{\tau}_c \\
&\langle\, s\ \rho'\ [\,e \mid c\,]\ \sigma'\,\rangle \\
&\textbf{where}\ \rho' = \mathsf{bind}\,(\,\langle x_1,x_2\rangle,\langle v_1,v_2\rangle,\mathsf{set}\,(n,\rho\,)\,) \\
&\qquad\quad \sigma' = \mathsf{set_module}\,(m,\sigma\,)
\end{aligned} \tag{4.37}$$

Monotonicity
That the rule is monotonic follows from the definition of $\sqsupseteq$ over configurations and value and code stacks.

Simulation
Rule (4.37) is simulated by itself and monotonicity ensures that the simulation preserves abstraction.

Approximation
In the case where the rule (4.37) is approximated by rule (5.1) it follows from Lemma 6 and Lemma 7 and the definition of $\sqsupseteq$ over configurations and value and code stacks that abstraction is preserved.

5.9.12 Message Retrieval

5.9.12.1 Message delivery

$$(4.38) \quad \langle\, s\ \rho\ c\ \sigma \,\rangle \xrightarrow{\tau}_c \langle\, s\ \rho\ c\ \mathsf{deliver}(\sigma)\,\rangle$$

Monotonicity
That the rule is monotonic follows from that the deliver function is monotonic and the definition of $\sqsupseteq$ over configurations and states.

Simulation
Rule (4.38) is simulated by itself and monotonicity ensures that the simulation preserves abstraction.

Approximation
In the case where the rule (4.38) is approximated by rule (5.1) it follows from that the deliver function is monotonic and the definition of $\sqsupseteq$ over configurations and states that abstraction is preserved.

5.9.12.2 Transformation from `receive` to rec intermediate

$$(4.39) \quad \begin{array}{l} \langle\, s\ \rho\ [\,[\![\texttt{receive}\ clauses\ \texttt{after}\ x_t\ \texttt{->}\ e_t\ \texttt{end}]\!]\mid c\,]\ \sigma\,\rangle \xrightarrow{\tau}_c \\ \langle\, s\ \rho\ [\,\mathsf{rec}(q,q,cls,x_t,e_t)\mid c\,]\ \sigma\,\rangle \\ \textbf{where}\ q = \mathsf{queue}(\sigma) \\ \qquad clause_1 \ldots clause_i = clauses \\ \qquad cls = [\,clause_1,\ldots,clause_i\,] \end{array}$$

Monotonicity
That the rule is monotonic follows from that the queue function is monotonic and the definition of $\sqsupseteq$ over configurations and code stack.

Simulation
Rule (4.39) is simulated by itself and monotonicity ensures that the simulation preserves abstraction.

Approximation

In the case where the rule (4.39) is approximated by rule (5.1) it follows Lemma 6 and Lemma 7 that abstraction is preserved.

5.9.12.3 The rec intermediate: first message in queue match

$$(4.40)\quad \frac{\bot \notin \mathsf{match}\,(p,v)}{\begin{array}{l}\langle\, s\ \rho\ [\,\mathsf{rec}([\,v \mid q_1\,],q,cls,x_t,e_t) \mid c\,]\ \sigma\,\rangle \overset{\tau}{\rightarrow}_c \\ \langle\, s\ \rho_1\ [\,g,\mathsf{if}(c_1,c_2) \mid c\,]\ \sigma\,\rangle \\ \textbf{where}\ [\,[\![\,p\ \texttt{when}\ g\ \texttt{->}\ e\,]\!] \mid cls'\,] = cls \\ \qquad \rho_1 = \mathsf{update}\,(\,\mathsf{match}\,(\,p,v\,),\rho\,) \\ \qquad n = \mathsf{current}\,(\,\rho\,) \\ \qquad c_1 = [\,\mathsf{message}(v),e,\mathsf{restore}(n)\,] \\ \qquad c_2 = [\,\mathsf{restore}(n),\mathsf{rec}(q_1,q,cls,x_t,e_t)\,]\end{array}}$$

Monotonicity

That the rule is monotonic follows from that the functions update, match and current are monotonic and the definition of $\sqsupseteq$ over configurations, contexts and code stack.

Simulation

Rule (4.40) is simulated by rule (5.9), the‘ Lemma 2 and that the functions update, match and current are monotonic and from the definition of $\sqsupseteq$ over configurations, contexts and code stack ensures that the simulation preserves abstraction.

$$(5.9)\quad \langle\, s\ \rho\ [\,\mathsf{rec}(\top,\top,cls,x_t,e_t) \mid c\,]\ \sigma\,\rangle \overset{\tau}{\rightarrow}_a \langle\, s\ \rho_1\ [\,g,\mathsf{if}(c_1,c_2) \mid c\,]\ \sigma\,\rangle$$

$$\begin{array}{l}\textbf{where}\ [\,[\![\,p\ \texttt{when}\ g\ \texttt{->}\ e\,]\!] \mid cls'\,] = cls \\ \qquad \rho_1 = \mathsf{update}\,(\,\mathsf{match}\,(\,p,\top\,),\rho\,) \\ \qquad n = \mathsf{current}\,(\,\rho\,) \\ \qquad p_1 = \mathsf{subst}\,(\,p,\mathsf{match}\,(\,p,\top\,)\,) \\ \qquad c_1 = [\,\mathsf{message}(p_1),e,\mathsf{restore}(n)\,] \\ \qquad c_2 = [\,\mathsf{restore}(n),\mathsf{rec}(\top,\top,cls,x_t,e_t)\,]\end{array}$$

Approximation
In the case where the rule (4.40) is approximated by rule (5.1) it follows from Lemma 6 and Lemma 7 and that the function subst is monotonic that abstraction is preserved.

5.9.12.4 The rec intermediate: first message in queue does not match

$$(4.41)\quad \frac{\bot \in \mathtt{match}(p,v)}{\langle\, s\ \rho\ [\mathsf{rec}([v \mid q_1], q, cls, x_t, e_t) \mid c]\ \sigma\,\rangle \xrightarrow{\tau}_c \langle\, s\ \rho\ [\mathsf{rec}(q_1, q, cls, x_t, e_t) \mid c]\ \sigma\,\rangle}$$

Monotonicity
That the rule is monotonic follows from that the function match is monotonic and the definition of $\sqsupseteq$ over configurations and code stack.

Simulation
Rule (4.41) is simulated by rule (5.10) and the definition of $\sqsupseteq$ over configurations and code stack ensures that the simulation preserves abstraction.

$$(5.10)\quad \langle\, s\ \rho\ [\mathsf{rec}(\top, \top, cls, x_t, e_t) \mid c]\ \sigma\,\rangle \xrightarrow{\tau}_a \langle\, s\ \rho\ [\mathsf{rec}(\top, \top, cls, x_t, e_t) \mid c]\ \sigma\,\rangle$$

Approximation
In the case where the rule (4.41) is approximated by rule (5.1) it follows Lemma 6 and Lemma 7 that abstraction is preserved.

5.9.12.5 The rec intermediate: all elements in the queue has been matched against the first clause

$$(4.42)\quad \langle\, s\ \rho\ [\mathsf{rec}([\,], q, [cl \mid cls], x_t, e_t) \mid c]\ \sigma\,\rangle \xrightarrow{\tau}_c \langle\, s\ \rho\ [\mathsf{rec}(q, q, cls, x_t, e_t) \mid c]\ \sigma\,\rangle$$

Monotonicity
That the rule is monotonic follows from the definition of $\sqsupseteq$ over configurations and code stack.

Simulation
Rule (4.42) is simulated by rule (5.11) and the definition of $\sqsupseteq$ over configurations and code stack ensures that the simulation preserves abstraction.

$$(5.11)\quad \begin{array}{l}\langle\, s\ \rho\ [\mathsf{rec}(\top,\top,[cl \mid cls],x_t,e_t) \mid c]\ \sigma\,\rangle \\ \langle\, s\ \rho\ [\mathsf{rec}(\top,\top,cls,x_t,e_t) \mid c]\ \sigma\,\rangle\end{array} \xrightarrow{\tau} a$$

Approximation
In the case where the rule (4.42) is approximated by rule (5.1) it follows Lemma 6 and Lemma 7 that abstraction is preserved.

5.9.12.6 The rec intermediate: all elements in the queue has been matched against all clauses

$$(4.43)\quad \frac{\rho(x_t) \neq \texttt{'infinity'}}{\langle\, s\ \rho\ [\mathsf{rec}([\,],q,[\,],x_t,e_t) \mid c]\ \sigma\,\rangle \xrightarrow{\texttt{timeout}} c\langle\, s\ \rho\ [e_t \mid c]\ \sigma\,\rangle}$$

Monotonicity
That the rule is monotonic follows from the definition of $\sqsupseteq$ over configurations and code stack.

Simulation
Rule (4.43) is simulated by rule (5.12) and the definition of $\sqsupseteq$ over configurations and code stack ensures that the simulation preserves abstraction.

$$(5.12)\quad \frac{\rho(x_t) \neq \texttt{'infinity'}}{\langle\, s\ \rho\ [\mathsf{rec}(\top,\top,[\,],x_t,e_t) \mid c]x\ \sigma\,\rangle \xrightarrow{\texttt{timeout}} a\langle\, s\ \rho\ [e_t \mid c]\ \sigma\,\rangle}$$

Approximation
In the case where the rule (4.43) is approximated by rule (5.1) it follows Lemma 6 and Lemma 7 that abstraction is preserved.

5.9.12.7 The message intermediate

$$(4.44)\quad \begin{array}{l}\langle\, s\ \rho\ [\mathsf{message}(v) \mid c]\ \sigma\,\rangle \\ \langle\, s\ \rho\ c\ \mathsf{remove}(\,v,\sigma\,)\rangle\end{array} \xrightarrow{\texttt{receive}\ v} c$$

Monotonicity
That the rule is monotonic follows from that the function remove is monotonic and the definition of $\sqsupseteq$ over configurations, states and code stack.

Simulation

Rule (4.44) is simulated by itself and monotonicity ensures that the simulation preserves abstraction.

Approximation

In the case where the rule (4.44) is approximated by rule (5.1) it follows Lemma 6 and that the function remove is monotonic that abstraction is preserved.

This concludes the abstract semantics. We have defined the abstract semantics in terms of how it differs from the concrete semantics and shown that it cam simulate the concrete semantics.

6. Process Structure Extraction

In this chapter we will describe how the generic technique of extracting the supervision structure from the source code, as given by the abstract semantics presented in Chapter 5, has been realised in the tool. The tool employs a symbolic evaluator, and we show how this evaluator is devised based on the abstract semantics. We will then detail some of the parameters of the tool that may be used to guide the extraction in order to make the extraction tractable. Finally we discuss the limitations of the implementation.

6.1 Background

The extraction of, or analysis of, the process structure described in this chapter was preceded by an incomplete and unsound version presented in [Nyström and Jonsson 2001]. This first attempt was a prototype intended to investigate if the extracted process structure could be precise enough for further analysis; encouraged by the initial results we decided to continue and devise a sound method of extracting an overapproximation of the process structure.

In order to arrive at a sound analysis we had to base it on a formal understanding of the language and its supporting runtime system, this is formally described in Chapter 4 with an abstraction to achieve the desired approximation as described in Chapter 5.

6.2 From Abstract Semantics to Extraction

We have not implemented the abstract machine defined in Chapter 5. Instead we have implemented a symbolic recursive descent evaluator based on that abstract machine. The ease with which one implements a recursive descent evaluator in a functional language gives rise to a number of differences between the evaluator and the abstract machine:

- For each of the CORE ERLANG constructs, our implementation contains one function that embodies all the rules of the abstract machine that describes this construct.
- The value and code stacks of the abstract machine are implicitly represented by the call stack of the evaluator. The dump is represented explicitly

using intermediates on the code stack in the semantics, this is also implicitly represented by call stack of the evaluator.

- Our evaluator does not assume that the programme has be transformed into normal form, but explicitly evaluates all subexpressions. The reason that we assume a normal form in the abstract machine is simply to make the rules clearer by removing unimportant details whenever possible, which of course is not necessary in an implementation.
- The evaluator has to deal explicitly with the nondeterminism caused by approximations, this is handled by letting the evaluation return sets of values and states rather than a single value and state as one would otherwise expect.
- The evaluator will represent an exception as a special value returned rather than an object on the code stack, since the evaluator implicitly represents the abstract machine's code stack by the call stack.

The relation between the abstract machine and the evaluator can be clarified by the following examples, where we discuss first the `let` construct and then the handling of the `apply` and `try-catch` constructs.

In the abstract machine the `let` construct is described by the following three rules:

$$\begin{array}{ll} (4.8) & \langle\, s\ \rho\ [\, [\![\texttt{let}\ \langle x_1,\ldots,x_i\rangle = \langle e_1,\ldots,e_i\rangle\ \texttt{in}\ e]\!]\ |\ c\,]\ \sigma\,\rangle \xrightarrow{\tau}_c \\ & \langle\, s\ \rho\ [\, e_1,\ldots,e_i,\mathsf{def}(x_1,\ldots,x_i),e,\mathsf{restore}(n)\ |\ c\,]\ \sigma\,\rangle \\ & \textbf{where}\ n = \mathsf{current}(\,\rho\,) \end{array}$$

$$\begin{array}{ll} (4.9) & \langle\, [\, v_i,\ldots,v_1\ |\ s\,]\ \rho\ [\,\mathsf{def}(x_1,\ldots,x_i)\ |\ c\,]\ \sigma\,\rangle \xrightarrow{\tau}_c \langle\, s\ \rho'\ c\ \sigma\,\rangle \\ & \textbf{where}\ \rho' = \mathsf{bind}(\langle x_1,\ldots,x_i\rangle,\langle v_1,\ldots,v_i\rangle,\rho) \end{array}$$

$$\begin{array}{ll} (4.10) & \langle\, s\ \rho\ [\,\mathsf{restore}(n)\ |\ c\,]\ \sigma\,\rangle \xrightarrow{\tau}_c \langle\, s\ \rho'\ c\ \sigma\,\rangle \\ & \textbf{where}\ \rho' = \mathsf{set}(\,n,\rho\,) \end{array}$$

Rule (4.8) transforms the `let` expression into the expressions to bind and the two intermediates def and restore interspaced with the body of the `let` expression. The `def` Rule (4.9) takes the result of the expressions to be bound from the value stack and constructs a new context with the added bindings. The `restore` Rule (4.10) restores the context to the one that was previous to the evaluation of the `let`, thus "undoing" the additional bindings performed by the def intermediate.

The corresponding function in the evaluator is described by the pseudo code in Figure 6.1 which realises the Rules (4.8) – (4.10) as follows: the function `eval_let` evaluates the subexpressions `Expressions` by a call to the function `eval_exprs`, which evaluates lists of expressions, this corresponds to pushing the expressions onto the code stack in the abstract semantics. The result `Values` of evaluating the expressions are bound to the variables `Vars` to create a new evaluation context `Cntxt1`, this corresponds to the `def` intermediate. The body `Expr` of the `let` expression is finally evaluated in the new context and the new state `State1` returned by the call to `eval_exprs`.

```
eval_let(Vars, Expressions, Expr, State, Cntxt) ->
  {Values, State1} =
    eval_exprs(Expressions, State, Cntxt),
  Cntxt1 = bind(Vars, Values, Cntxt),
  eval(Expr, State1, Cntxt1).
```

Figure 6.1: Psuedo Code for Extraction of Let Expression: Step One

In the `eval_let` function the state and context of the abstract machine 's configuration is represented explicitly, while the value stack, code stack and dump are represented implicitly.

- The value stack is represented by the return values of the evaluating functions as can be seen by the call to `eval_exprs` which will return a list of values `Values` that are the result of evaluating a list of expressions.
- The code stack is implicitly represented by the call stack of ERLANG where the addition of subexpressions and intermediate expressions to the code stack are represented by the sequencing within the evaluating functions. It is because of the implicit representation that the evaluating functions have to return not only the value resulting from the evaluation, but also the new state generated.

The context will however not be returned by the evaluating functions since new bindings can only be added to subexpressions and is only visible to those subexpressions and subsequent subexpressions. Since bindings propagate only in calls to evaluate subexpressions we do not have to have any functionality corresponding to the restore intermediate. The evaluation context consists of the variable bindings and the parameters of the evaluation described in Section 6.3.

In the evaluator we also have to take into account the nondeterministic nature of the semantics we are using. As a consequence, the evaluating functions cannot simply return $\langle value, state \rangle$ tuples but have to return sets of such tuples corresponding to the possible different results of the evaluation. We have modified the pseudo code for `let` in Figure 6.2 to account for the nondeterminism, where `eval_exprs` returns a set of $\langle value, state \rangle$ tuples. The

binding and subsequent evaluation of the `let` body have to be performed for each member of the $\langle value, state \rangle$ set and the resulting sets united.[1]

```
eval_let(Vars, Expressions, Expr, State, Cntxt) ->
  ValueStateSet =
    eval_exprs(Expressions, State, Cntxt),
  union(fun({Values, State1}) ->
            Cntxt1 = bind(Vars, Values, Cntxt),
            eval(Expr, State1, Cntxt1)
        end,
        ValueStateSet).
```

Figure 6.2: Pseudo Code for Extraction of Let Expression: with nondeterminism

We have so far ignored exceptions. In Figure 6.3 we have added handling of exceptions to the evaluation of `let` expressions. In the evaluator the exceptions are represented as values specially marked as exceptions. This because we do not have an explicit code stack as in the abstract machine. When the `eval_let` encounters an exception it does not construct a new context or evaluate the body of the `let`, but simply includes exception among the returned values. It should be noted that this is only exception in one of the possible evaluations and as a consequence no other members in the set are in any way affected. We present further examples of how exceptions are handled in the `try-catch` example below.
To show how the evaluator works when the implict dump would be used in the abstract machine we will exemplify with some of the rules handling `apply`:

$$(4.17)\quad \frac{\rho(x^f) \in \text{Atom}}{\begin{array}{l}\langle s\ \rho\ [[\![\texttt{apply}\ x^f\ (x_1,\ldots,x_i)]\!] \mid c]\ \sigma\rangle \xrightarrow{\mu}_c \\ \langle [\,]\ \rho'\ [e, \mathsf{return}(\mathsf{s},\mathsf{n},\mathsf{m}) \mid c]\ \sigma\rangle \\ \textbf{where}\ n = \mathsf{current}(\rho) \\ \qquad m = \mathsf{module}(\sigma) \\ \qquad \langle\langle [\![\texttt{fun}(x'_1,\ldots,x'_i) = e]\!], 0, m\rangle\rangle = \mathsf{func}\left(m, \rho(x^f), i, \sigma\right) \\ \qquad \rho' = \mathsf{bind}(\langle x'_1,\ldots,x'_i\rangle, \langle \rho(x_1),\ldots,\rho(x_i)\rangle, \mathsf{set}(0,\rho)) \\ \qquad \mu = m : \rho(x^f)\,(\rho(x_1),\ldots,\rho(x_i))\end{array}}$$

This is the main rule, where the function argument of `apply` is an atom naming the called function. The function is looked up and a new environment is constructed to bind the formal parameters to the actual parameters. The body

[1]The union construct takes a function as its first argument and applies it to all the elements making the union of all the results. In the pseudo code, the sets are implemented using lists (the operator ++ is append on lists in ERLANG).

```
eval_let(Vars, Expressions, Expr, State, Cntxt) ->
  ValueStateSet =
    eval_exprs(Expressions, State, Cntxt),
  union(fun({Values, State1}) ->
            case is_exception(Values) of
              true -> set(Values, State1);
              false ->
                Cntxt1 = bind(Vars, Values, Cntxt),
                eval(Expr, State1, Cntxt1)
            end
        end,
        ValueStateSet).

set(Value, State) -> [{Value, State}].
```

Figure 6.3: Psuedo Code for Extraction of Let Expression: with exceptions

of the function is pushed on top of the code stack and followed by a return intermediate, that contains the necessary information necessary to restore the state after the call.

$$(5.7)\quad \begin{array}{l} \langle\, s\ \rho\ [\,[\![\,\mathtt{apply}\ x^f\,(x_1,\ldots,x_i)\,]\!]\mid c\,]\ \sigma\,\rangle \overset{\top}{\rightarrow}_a \\ \langle\, s\ \rho\ [\,\mathsf{approx}()\mid c\,]\ \sigma'\,\rangle \\ \textbf{where}\ \sigma' = \mathsf{publish}(\top,\top,\sigma) \end{array}$$

This rule deals with the case where the called function is unknown or the evaluation has been cutoff at this point. An approx intermediate is placed on the stack and the approximation is published.

$$(4.20)\quad \begin{array}{l} \langle\,[v]\ \rho\ [\,\mathsf{return}(s,n,m)\mid c\,]\ \sigma\,\rangle \overset{\tau}{\rightarrow}_c \langle\,[v\mid s]\ \rho'\ c\ \sigma'\,\rangle \\ \quad \textbf{where}\ \rho' = \mathsf{set}\,(n,\rho\,) \\ \qquad\quad \sigma' = \mathsf{set\_module}\,(m,\sigma\,) \end{array}$$

This rule handles the return from a function call, the elements on the value stack are placed at the top of the restored stack and the environment and current module are restored to their values prior to the call.

$$(5.2)\quad \langle\, s\ \rho\ [\,\mathsf{approx}()\mid c\,]\ \sigma\,\rangle \overset{\tau}{\rightarrow}_a \langle\,[\top\mid s]\ \rho\ c\ \sigma\,\rangle$$

This rule mirrors the concrete return rule in the case where the call has been approximated. An approximated value is placed on the value stack.

$$(5.3)\quad \langle\, s\ \rho\ [\,\mathsf{approx}()\mid c\,]\ \sigma\,\rangle \xrightarrow{\tau}_a \langle\, [\,]\ \rho\ [\,\mathsf{exception}(\top,\top)\mid c\,]\ \sigma\,\rangle$$

This is similar to the previous rule but this mirrors the concrete case where an exception has occurred.
In Figure 6.4 we can see the pseudo code corresponding to these rules. We have for simplicity omitted the handling of multiple results and exceptions from the pseudo code, but the complete pseudo code can be found in Figure 6.6. The evaluation is performed as follows:

- The defining fun of that function is looked up in the current module as given by the context.
- In the `lookup_function Name` expression is evaluated, this can either be an atom or a fun (we have omitted the fun case for the sake of clarity) or it can result in the top element signifying an approximation. The call to evaluate the `Name` expression can not cause an exception. If the top element was returned, it is returned as the result of the lookup, otherwise the `ensure_loaded` function tries to lookup the function in the definition of the module. If the lookup in the module definition fails an approximation is made an the top element returned; a typical reason for the lookup to fail is that the module has not been written yet. If the lookup in the module definition succeeds the fun is returned.
- The actual parameters are evaluated. This could cause an exception, how that is handled can be seen in Figure 6.6.
- The supposed fun is examined to see whether it is a call to be approximated using `is_top` which returns true if the fun is the top element. If so, we approximate the call by returning the top element and the state `State1` as resulting from the evaluation of the actual parameters. If the fun is not the top element it is a tuple of `{Formal, Body}` of the formal parameters and the body of the function; the body of the function is evaluated in the context where the formal parameters have been bound to the actual parameters.

In the final example we look at how the `try-catch` construct is handled in order to understand how exceptions work in the evaluator.

$$(4.35)\quad \begin{array}{l} \langle\, s\ \rho\ [\,[\![\,\mathtt{try}\ e_1\ \mathtt{catch}\,(x_1,x_2)\ \mathtt{->}\ e_2\,]\!]\mid c\,]\ \sigma\,\rangle \xrightarrow{\tau}_c \\ \langle\, s\ \rho\ [\,e_1,\mathsf{catch}(x_1,x_2,e_2,s,n,m),\mathsf{restore}(n)\mid c\,]\ \sigma\,\rangle \\ \textbf{where}\ n=\mathsf{current}(\,\rho\,) \\ \qquad\quad m=\mathsf{module}(\,\sigma\,) \end{array}$$

```
eval_apply(Name, Arity, Args, State, Cntxt) ->
  Module = module(Cntxt),
  {Fun, State1} =
    lookup_function(Module,Name,Arity,State,Cntxt),
  {Actual, State2} =
    eval_exprs(Args, State1, Cntxt),
  case is_top(Fun) of
    true -> {top, State2};
    false ->
      {Formal, Body} = Fun,
      Cntxt1 = bind(Formal, Actual, Cntxt),
      eval(Body, State2, Cntxt1)
  end.

lookup_function(Module,Name,Arity,State,Cntxt) ->
  {Name, State1} = eval(Name, State, Cntxt),
  case is_top(Name) of
    true -> {top, State1};
    false ->
      case code:ensure_loaded(Module, Name) of
        error -> {top, State1};
        {ok, Fun} -> {Fun, State1}
      end,
  end.
```

Figure 6.4: Psuedo Code for Extraction of Apply Expression

The main rule that places the expression to be caught on the code stack followed by the catch and restore intermediates, they contain the information necessary to restore state should an exception by caught by the catch intermediate.

$$\langle s\ \rho\ [\mathsf{catch}(x_1,x_2,e,s_1,n,m) \mid c]\ \sigma\ \rangle \xrightarrow{\tau}_c \langle s\ \rho\ c\ \sigma\ \rangle \tag{4.36}$$

This rule handles the case where evaluation has reached a catch intermediate without having caused an exception, it is simply removed from the code stack.

$$\begin{aligned} &\langle [\,]\ \rho\ [\mathsf{exception}(v_1,v_2), \mathsf{catch}(x_1,x_2,e,s,n,m) \mid c]\ \sigma\ \rangle \xrightarrow{\tau}_c \\ &\langle s\ \rho'\ [e \mid c]\ \sigma'\rangle \\ &\textbf{where}\ \rho' = \mathsf{bind}(\langle x_1,x_2\rangle, \langle v_1,v_2\rangle, \mathsf{set}(n,\rho)) \\ &\qquad\quad \sigma' = \mathsf{set\_module}(m,\sigma) \end{aligned} \tag{4.37}$$

This rule handles the case where evaluation has reached a catch intermediate when an exception has been raised. An environment where the type and reason for the exception are bound are created and the the body of the `try-catch` is evaluated in that environment.
In Figure 6.5 we can see the pseudo code corresponding to these rules. We have for the sake of simplicity omitted the handling of multiple results from the pseudo code, but the complete pseudo code can be found in Figure 6.7. The expression to be caught is first evaluated, and if it results in an exception a new context binding the `try-catch` variables to the type and value of the exception is constructed before the body of the `try-catch` is evaluated. If the expression to be caught does not result in an exception, the result is simply returned.

```
eval_try(E1, [X1, X2], E2, State, Cntxt) ->
  {Value, State1} = eval(E1, State, Cntxt),
  case is_exception(Value) of
    true -> {T, V} = Value,
            Cntxt1 = bind([X1, X2], [T, V], Cntxt),
            eval(E2, State1, Cntxt1);
    false -> {Value, State1}
  end.
```

Figure 6.5: Psuedo Code for Extraction of Try Expression

```
eval_apply(Name,Arity,Formal,Args,State,Cntxt) ->
  Module = module(Cntxt),
  SetofFunState =
    lookup_function(Module,Name,Arity,State,Cntxt),
  union(
    fun({Fun, State1} ->
      SetofActualState =
        eval_exprs(Args, State1, Cntxt),
      union(fun({Actual, State2}) ->
              case {is_exception(Actual),
                    is_top(Fun)} of
                {true, _} -> set(Actual, State2);
                {false, true} ->
                  State3 = publish(top, State2),
                  union(set(top, State2),
                        set(mkexception(top),
                            State3));
                _ ->
                  {Formal, Body} = Fun,
                  Cntxt1 =
                    bind(Formal,Actual,Cntxt),
                  eval(Body, State2, Cntxt1)
              end,
            SetofActualState)
    end,
  SetofFunState).

lookup_function(Module,Name,Arity,State,Cntxt) ->
  SetofNameState = eval(Name, State, Cntxt),
  union(fun({Name, State1} ->
            case is_top(Name) of
              true -> set(top, State1);
              false ->
                case code:ensure_loaded(Module,
                                        Name) of
                  error -> set(top, State1);
                  {ok, Fun} -> set(Fun, State1)
                end
            end,
        SetofNameState).
```

Figure 6.6: Complete Psuedo Code for Extraction of Apply Expression

```
eval_try(E1, [X1, X2], E2, State, Cntxt) ->
  SetofValueState = eval(E1, State, Cntxt),
  union(fun({Value, State1}) ->
            case is_exception(Value) of
              true -> {T, V} = Value,
                      Cntxt1 =
                        bind([X1, X2],[T, V],Cntxt),
                      eval(E2, State1, Cntxt1);
              false -> set(Value, State1)
            end
        end,
        SetofValueState).
```

Figure 6.7: Complete Psuedo Code for Extraction of Try Expression

6.3 Extraction Parameters

To be able to deal with realistic code we have to be able to specify a number of parameters of the extraction. There are four distinct types of parameters: first the static setup of the evaluation providing parts of the environment; second to what maximal depth should the evaluator descend abstracting subsequent subexpressions; third how and what parts of the global state should be abstracted over; and fourth at what detail exceptions are dealt with.

6.3.1 Setup

There are a number of static parameters that have to be defined when calling the evaluator, these are presently:

- *Path* which directs where, and in which order, directories are searched for the source code of modules.
- *Inclusion Directories* which directs where, and in which order, directories are searched for included files.
- *Node* is the name of the ERLANG-node in which the extracting evaluator is perceived as executing. The name of the ERLANG-node may, and indeed does in some of the analysed applications, influence the behaviour of the application.

6.3.2 Evaluation Depth

In order to ensure that the evaluation terminates we have to set a maximum evaluation depth, i.e., the maximum number of recursive calls to the evaluator without a return. When the maximum evaluator depth is exceeded the function calls are abstracted.

This parameter can be tweaked to get the correct level of approximation, however it should be used with caution since should the parts abstracted have impact on the supervision structure the result may be too inexact. It can also paradoxically be the case that a higher maximum evaluation depth may result in a shorter execution time than a lesser maximum evaluation depth, because the higher maximum evaluation depth can provide higher precision and thus fewer alternatives may have to be investigated.

There is also a simple cycle detection mechanism that is either used or not. The cycle detection mechanism will record each recursive function call with its parameters; should two calls to the same function have the same parameters the second call is abstracted. That the same function is called with the same parameters does not have to mean that we have a cycle in the execution since the global state of the process may have changed. Conversely we can not do without a maximum evaluation depth when using the cycle detection since we may have infinite evaluations without cycles in the presence of potentially infinitely growing data, e.g., integers.

6.3.3 Global State

It is important for the precision of the extraction how we abstract the global state of an ERLANG-node. There are two obvious ways to deal with each part of the global state; either abstract it entirely by letting calls to functions that access this part return the top element, or emulate its state completely. There are also intermediate solutions, where a part of the global state is emulated to some level of precision. Our evaluator allows all parts of the global state to be abstracted away. We have also implemented emulation of some parts of the run-time system to various degrees of precision. Below, we describe these facilities in more detail.

6.3.3.1 Registry

The registration of names to processes, either locally within an ERLANG-node or globally, has been described in Section 2.3.2 and Section 2.4 respectively. It may be of great help to emulate fully the registration of names since if the registration of a process is performed in the initialisation of a process and all further references to this process is via this name, it is easy to conclude what process is referred to in communications and other actions requiring a process identity. On the other hand, if the identity of a process is supplied in another manner, such as messages or from ETS tables, it is likely that the analysis will fail to determine what process is affected by the action.

The registration of processes is either emulated in full or wholly abstracted, with the possibility to have processes not started by the application itself pre-registered by the evaluator, such as processes in the more important OTP libraries.

6.3.3.2 ERLANG Term Storage

The ERLANG Term Storage (ETS) is used both to store persistent data on an ERLANG-node basis or as a second means of communication between processes. The tables have associated access rights `private`, `protected` and `public`, with the meaning:

- A `private` table can only be manipulated and examined by the process that created it.
- A `protected` table can only be manipulated by the process that created it, but all process can examine it.
- A `public` table can be manipulated and examined by all processes, and it has to be explicitly destroyed in order to vanish, unlike the other two types of tables which will vanish if the creating process is terminated.

The persistent nature of the `public` tables makes them popular in fault tolerant applications since they enable the storage of configuration and other data relevant to a specific rôle in an application in a manner that will survive the restart of the process performing this rôle. The problem with `public` tables is that they can be in an inconsistent state after a process failure and that one is tempted to to use the tables for communication and thus breaking the encapsulation of data intended by the table.

Our evaluator provides several levels of precision for the emulation of ETS tables:

- Full emulation of ETS tables, taking an unknown environment into consideration. This implies that all access to `public` tables results in abstraction, since we do not know whether some process, that may not even be part of the application, has modified the tables.
- Full emulation of ETS tables, under a closed world assumption, that the tables created by the application only be modified by process in the application during the initialisation part of process until the initialisation of the application is completed.

Our handling of ETS tables has one more special feature namely, the special handling of the `public` table `ac_tab` used by the `application_master` process that is part of the runtime system. The `ac_tab` table contains information about the loaded and running application on an ERLANG-node. This table should not be directly accessed, but only used by the functions in the `application` module; however for historical reasons it is examined directly in some code. We have therefore fully emulated the `application` module's format and handling of the `ac_tab` table, but the `ac_tab` table is `private` so that even without the closed world assumption we can use the same framework as for the other tables.

The ETS tables can either be wholly abstracted, emulated under a closed world assumption, or fully emulated. There is the possibility to have `protected` or `public` tables already created and initialised at the start of simulation, and the `ac_tab` table is always present.

6.3.3.3 Mnesia

Mnesia is a distributed database that has a large number of configurations. For our purposes most of these options are not important, such as if a database table is stored in primary or secondary memory. Dealing with Mnesia is analogous to the simulation of ETS except on details where the manipulation and examination functions and internal formats differ. Unlike with ETS, for Mnesia we only support wholly abstracted or fully emulated evaluation.

6.3.3.4 File System

The file system is extensively used for configuration data and logging of various forms, which makes it important to emulate the file system in order to improve the precision of the extraction. The file system can either be wholly abstracted or fully emulated, with the possibility of configuring the file system with directories and files before the evaluation.

6.3.4 Exception Handling

Since we are dealing with a language where the advocated means of dealing with errors is to throw exceptions rather than writing defensive code, it is essential that we handle the propagation of exceptions correctly. There is however a problem of scalability with the combination of exceptions and abstraction. For many of the language constructs and built in functions we have to consider the case where an exception is raised when one of the components or arguments contain an abstraction, and this exception must also be abstracted. This gives rise to an exponential growth in the number of possible results since the construct or call will result in either a normal value or an exception.

In order that we may at least partially address the problem of exceptions we allow the user decide how to deal with exceptions. There are three distinct strategies and one further abstraction that can be made orthogonally to these:

- The first strategy is to keep all exceptions and hope that the growth in possible evaluation is not intractable.
- The second strategy is to ignore all exceptions that will not be caught by any outer catch; the rationale for this is that we are only interested in the fault handling behaviour that actually results in starting the application.
- The third strategy is to ignore all exceptions. This is an unsound approach but can be useful in cases where the second strategy is intractable and we would like to see the supervision structures merely to have a presentation of the application, although we can not base any analysis on the extracted model. Unfortunately even this choice does not guarantee that we will extract a supervision structure since in some cases exceptions are used for other than error handling, thus exceptions ignored contains vital information to derive the supervision structure.

Further means of reducing the number of different evaluations is to abstract the exceptions themselves, i.e., exceptions with unknown type and value. In the evaluator it can be chosen how many different exceptions will be evaluated before they are all approximated by an abstract exception.

6.4 Limitations

Currently, our implementation of the evaluator can handle applications that are syntactically correct, i.e., those that pass through the initial phases of the compiler which produces CORE ERLANG. Applications can be analysed even if only parts of the source code are available; missing parts are handled in the same manner as calls to functions in unsupported library modules.

The most noticeable limitation is that when many unsupported library calls are made, the number of possible executions becomes intractable. This is the case both for the OTP application `mnesia`[Ericsson Utvecklings AB 2000, Mattsson et al. 1999] and for the first tries to analyse applications from the real life system AXD 301[Blau et al. 1999]. Precision can be improved by supporting more OTP libraries and simulating a larger part of the ERLANG runtime system, as has been done in order to analyse the AXD 301 applications.

Another limitation is that we can effectively analyse only applications that have been designed according to the suggestions of the OTP documentation and using the OTP library behaviours. This includes the restriction that the essential parts of the supervision structure should be set up by the initialisation code. This is a way to encourage use of standard coding idioms. It is in general intractable to analyse automatically the behaviour of arbitrarily structured code.

7. Experiments

In this chapter we will present the experiments that have been conducted with the tool on a number of applications. First is detailed the general setup and configuration of the experiments, then in the following sections the results are presented, and finally one experiment is singled out and described in detail.

7.1 Setup

In the experiments we have applied the tool to several OTP-library applications and a subsystem of the AXD 301 ATM switch. We will briefly describe the OTP-library applications; complete description of these applications are to be found in the OTP documentation [Ericsson Utvecklings AB 2000]:

`os_mon` This application monitors the resources of the underlying operating system, namely CPU, RAM-memory and disk usage.

`mnemosyne` The Mnemosyne application is a query interface to the distributed database Mnesia, which is part of the OTP-libraries. Mnemosyne is an extension of the ERLANG language, i.e., queries are written embedded in ERLANG code.

`sasl` The System Architecture Support Libraries application, SASL, provides support for alarm and release handling

`megaco` The Megaco application is a framework for building applications on top of the Megaco/H.248 protocol.

`inets` Inets is a container for Internet clients and servers. Currently, a HTTP server and a FTP client have been incorporated in Inets.

`crypto` The Crypto application provides message digests MD5 and SHA, and CBC-DES encryption and decryption. The purpose of this application is to provide message digest and DES encryption for SMNPv3.

The OTP applications do not require any special configuration of the tool; they were all complete stand-alone applications with the source code to be found in the path of the ERLANG runtime system.

The other applications are part of the runtime configuration management subsystem of the AXD 301 ATM switch [Blau et al. 1999] as well as generic

applications extending the behaviours of OTP for use in the AXD 301. Access to this proprietary software was kindly granted by Ericsson for use in this work in order that the experiments would be performed on product quality software of the complexity expected in telecommunication systems. The applications are described only briefly below:

`rcm` The main application of the runtime configuration manager, including the `rcmInit`, `rcmKernel`, `rcmUtilities` and `rcmNtp`.

`rcmInit` Initialization application for runtime configuration manager. This is the application that is started first, it starts the rcmLocker application, creates mnesia and ETS tables, etc.

`rcmKernel` The kernel application of the runtime configuration manager, with children like the `rcmInit` application.

`rcmUtilities`
Provides a number Utility processes for the runtime configuration manager.

`rcmNtp` This application is responsible for the supervision of the Network Time Protocol server.

`sys1` Loading application for the generic applications extending the behaviours of OTP used in the AXD 301.

`sysCmd` AXD system utility processes that wrap ETS and Mnesia tables for enhanced upgrade support.

Since the AXD 301 applications analysed only constitute a subsystem, we had to configure the tool with additional code source paths and initialise various ETS and Mnesia tables that would normally contain configuration information used by the applications. We had to seed the ETS and Mnesia tables with configuration information describing on which processors a particular subsystem is running, and the filename of log files.

The experiments were performed on a 450MHz Pentium III with 256MB memory.

7.2 Results and Conclusions

The results of the experiments are summarised in Table 7.1, with the exception of `mnemosyne`, `megaco` and `inets` which are trivial. From this table of results and the source code of the applications we can draw conclusions regarding a number of important issues:

Code size The code sizes of the various applications analysed to determine if we can analyse production size applications, and runtimes of the applications.

Language constructs The language constructs and libraries used by the actual applications to see how much of the potential language we can analyse.

Incomplete source code Incomplete source code resulting from either applications at an early stage of development or where the whole system is too large to be completely analysed,

Precision The precision of the analysis to see how likely the analysis is to find potential problems.

It should be noted that initialisation is normally only a small part of the application. As an example, in the AXD 301 subsystem consisting of 57'310 lines of source code at most 1'068 out of these in 118 functions were executed in order to analyse one of its applications.

Code size

For larger applications our tool can symbolically execute in the order of 1000 lines of source code per second. The execution time seems to grow exponentially with the number of different process trees generated by the analysis, thus it is important for larger applications to have as high precision as possible in order to be able to analyse the application in reasonable time. One strategy that may be adopted when the system is too large to be analysed is to analyse subsystems independently, although for some of the subsystems, in actuality dependent on other subsystems for the configuration, the lacking information can cause loss of precision.

Language constructs

Inspecting the source code of the OTP and AXD 301 subsystem applications, we can conclude that they use all CORE ERLANG language constructs and a great many of the OTP libraries. This result gives us reason to believe that we can analyse most applications under the limitations detailed in Section 6.4.

Incomplete source code

When analysing the AXD 301 subsystem parts of the code were missing, as was the remainder of the system. The tool could analyse all the applications after initialisation of configuration tables, with some increase of approximations made.

Precision

For all OTP applications we have traced an execution of the application, and compared the process tree created with the trees created by our analysis. For

Name	Time	#Trees	#Lines	#Executed	#Total
OTP:					
os_mon	8.0s	200	260	620	2’588
sasl	1.1s	2	291	312	9’467
crypto	0.3s	1	52	65	337
AXD 301:					
rcm	180s	1’674	822	306’176	19’250
rcmInit	5.4s	1	650	5’320	3’744
rcmKernel	582s	3’708	1’068	570’976	16’272
rcmUtilities	1.7s	1	140	164	62
rcmNtp	1.9s	3	89	93	2’583
sys1	28.2s	8	897	58’395	33’227
sysCmd	0.2s	1	9	9	1’359

Table 7.1: *Analysis Statistics*

In the table:
Time = runtime of the analysis,
#Trees = number of process trees extracted,
#Lines = number of different source code lines used by initialisation,
#Executed = total number of lines executed, e.g, if some line is executed 7 times, this is counted as 7 lines
#Total = total number of lines in the entire application.

The rcm application only contain 63 lines of source code, however it includes the applications `rcmKernel`, `rcmUtilities` and `rcmNtp`, and together with them it makes a total of 19’250 lines.

all applications the tree generated by tracing an execution of the application was matched by one of the trees generated by our analysis.

For the AXD 301 applications we could not execute the incomplete subsystem. We have, however, shown the results to the senior engineers within the AXD 301 that provided us with the subsystem; they confirm that the analysis results correspond to the actual system.

The large number of trees generated by three of the applications (`os_mon`, `rcm` and `rcmKernel`) all arise from polling performed in the initialisation and results in a number of trees only differing in the number of sends and receives performed. If we ignore the number of differing send and receives only a few trees remain; in the case of `os_mon` only two out of the 200 trees would remain.

7.3 Extended Example: `Os_mon`

To illustrate the results of our analysis we will use the result of applying the tool to the OTP application `os_mon`, that monitors the underlying operating system for disk, memory and CPU usage. Snapshots from the tool after analysing `os_mon` is shown in Figure 7.1.

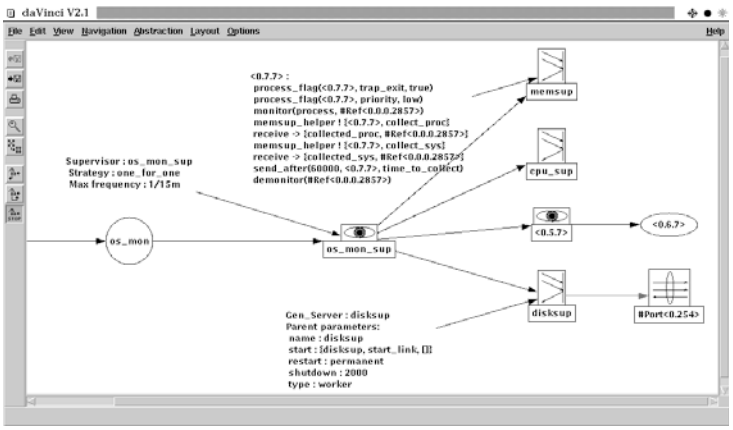


Figure 7.1: Creation tree for the OTP application `os_mon` with expanded information nodes.

The `os_mon` application running on a Linux system will have seven processes and one external port. Below we describe each process in the order of creation.

os_mon The application master process registered as `os_mon`.

os_mon_sup The supervisor of the processes that monitor the operating system resources; this process is registered as `os_mon_sup`. From its pa-

rameters one can see that it will allow at most 4 restarts per hour, and the restart strategy is `one_for_one`.

disksup The generic server registered as `disksup` is responsible for monitoring the disk usage. From the supervisor's parameters associated with this process we can see that it will be started by a call to `disksup:start_link()`, its restart strategy is `permanent`, it will be given 2000 milliseconds to shut down; and the type `worker`.

During initialisation `disksup` will start an external port which communicates with an external process which actually monitors the disk usage.

<0.5.7> This process, which is a supervisor bridge without a registered name, starts process `<0.6.7>` of which the analysis can not determine anything. Inspection of the source code reveals that the process `<0.6.7>`, that this process will dynamically be registered as `memsup_helper`, which `memsup` communicates with.

The reason that we do not know of the registration of `memsup_helper` process is, that instead of the usual method of letting the parent register it during its initialisation phase, it registers itself.

We could, when perusing the actual code, not determine any reason why the common design principles of ERLANG/OTP were not used. Since this process is in actuality started in much the same manner as the other children of the supervisor, our conclusion is that it is legacy code.

memsup The generic server registered as `memsup`, with process id `<0.7.7>`, is responsible for monitoring the memory usage of the ERLANG-node. In this case we have highlighted the sequence of actions that the process will perform:

```
process_flag(<0.7.7>, trap_exit, true)
process_flag(<0.7.7>, priority, low)
monitor(process, ⊤Term)
memsup_helper!{<0.7.7>, collect_proc}
receive {collected_proc, ⊤Term}
memsup_helper!{<0.7.7>, collect_sys}
receive {collected_sys, ⊤Term}
send_after(60000, <0.7.7>, time_to_collect)
demonitor(⊤Term)
```

The two first actions set process flags, followed by the monitoring of an unknown process. After these initial setups, the process will communicate with the registered process `memsup_helper`, sending and receiving twice. After these calls and responses it uses a built in fuction that

will send the message `time_to_collect` after 60'000 milliseconds. Finally it demonitors, i.e., removes the monitor, from an unknown process. Using monitors is an alterative to links for monitoring processes terminations.

The `monitor` built in function enables a process to monitor various aspects in the ERLANG-node, getting messages from the runtime system when something happens which affects the process.

cpu_sup The generic server registered as `cpu_sup` is responsible for monitoring the cpu usage of the system.

When analysing the `os_mon` application our tool generated 200 different trees, depending on approximations as mentioned in Chapter 6. The great multiplicity comes from a receive statement where the `memsup` process either gets information back from the `memsup_helper` process or a timeout occurs, this is repeated 10 times giving rise to 100 different trees. The final step from 100 trees to 200 is taken by a previous process, namely the anonymous `<0.5.7>`, that may fail during initialisation, but in such a manner that the remaining processes can be started. The tree presented here is the simplest where no retransmissions due to timeouts have to be performed, and the process `<0.5.7>` initialises successfully.

The automatic check performed by the tool for each tree shows that `os_mon` respects properties `P1` and `P2` provided that process `<0.6.7>` behaves correctly. This clearly indicates that in order to be confident in the failure recovery we need to examine `<0.6.7>`.

8. Thesis Summary

In this chapter we first summarise the achievements of this thesis and then present our conclusions that we have drawn from this work. We end the chapter with a description of how we intend to continue this work after the thesis.

8.1 Summary

In this thesis, we have presented a technique for automatically detecting deficiencies in the failure recovery mechanism of ERLANG applications, which are due to improperly designed supervision structures. The technique is structured into two distinct parts.

The first part consists of extracting the set of possible supervision structures by static analysis of the source code. The extracted set of possible supervision structures is a safe overapproximation of the static parts of supervision structures, extracted by symbolically executing the initialisation code of the application. By "the static part" we mean the processes started when the application is started and which are to remain running (possibly restarted to handle failures) until the application terminates. In the extraction we have assumed that the OTP libraries are used in the recommended way to set up the supervision structure, otherwise the precision becomes poor.

In order to ensure that the symbolic execution used to extract the supervision structure was correct and formally sound, we had to devise an operational semantics of the sequential part of CORE ERLANG that also dealt with the imperative side-effect mechanisms provided by the runtime system. Having devised an operational semantics we then devised an abstract version of the semantics that made the analysis tractable when analysing fault tolerance; we based the symbolic execution on this abstract operational semantics. The abstract semantics was shown to abstract the semantics in a precise way.

To evaluate the techniques we have implemented a tool, which automatically extracts sets of possible static supervision structures from source code and then can check the effects of a process failure in each supervision structure.

The tool can also perform several "sanity checks" to ensure that principles for the construction of good supervision structures are followed. If the principles are not followed strictly, the tool can check the effects of process failures;

this is useful when analysing legacy code applications, which may not have been designed using current design principles.

The second part of the technique involves analysing the effect of process failures on each of the extracted supervision structures, which entails analysing the effect of a particular process failure on the entire supervision structure. In order to be able to perform this analysis we have constructed a model of the supervisor's behaviour.

When formalising the properties we want to check we had two conflicting considerations: on the first hand the properties had to capture the essential aspects of fault tolerance of ERLANG applications; on the second hand the properties had to be possible to analyse with the precision provided by the extracted supervision structures.

The two essential properties we check are those of "repair" and "non concealment", which states that as system should be repaired with the restart of terminated processes and that a permanent error should not be concealed by the fault tolerance mechanism.

To be able to fulfil these main objectives we have also had to address a number of related issues. In order to be able to construct and explain our model of supervisor's behaviour we have had to make a detailed presentation of the fault tolerance mechanisms provided by the OTP libraries.

We have implemented a graphical tool that visualises the extracted supervision structure. This visualisation of the supervision structure can be presented to the designer for inspection with the possibility to choose different views depending on the information sought. As an example, the parts of the application that are affected by an abnormal process termination can be visualised.

In order to evaluate the abilities of the tool we have applied it to several OTP-library applications and a subsystem of the AXD 301 ATM switch. The visualisation was used to verify with the designers that the captured supervision structure was correctly captured.

8.2 Conclusions

In the view of the applications we have been able to analyse, we can conclude that using a combination of automatic model extraction for finding the structuring of the application and manually constructed models for the more elaborate runtime behaviour of library processes we can determine important aspects of the applications fault handling mechanisms. This shows that highly relevant properties of real life code can be determined for fault tolerance properties using a small amount of manual modelling.

The essential properties of the fault tolerance as well as the sanity checks can be fully automated. This together with the automated extraction of process structures can be developed into a push-button tool for checking important aspects of ERLANG programs from the source code.

The automatically extracted models of the applications can also be used by designers to understand the effects that fault tolerance mechanisms will have on their design, by providing simulations of the effects a fault will have on the application.

During the development of the tool we have noticed how important the correct level of abstraction is for keeping the model extraction and subsequent analysis tractable; this forces us to conclude that in some cases it will not suffice to add new manually constructed models of relevant libraries, but we would have to modify the model extraction to change what parts of the symbolic execution will be abstracted over.

We have indeed implemented such choices in the tool where in some cases the user can choose which side effects should be exactly modelled and which should be abstracted. In our tool the user controls whether ot not ets, mnesia and the registration of processes is modeled. If for example ets is used in a system analysed, but not included in the model, that leads to less precision in the result but we may still get all the possible supervision structures.

8.3 Further Work

There are two different and equally important courses that we will pursue in the continuation of this work; the first course of actions are to improve the quality of the current work, the second course is to extend the scope of what is addressed both in part of the source code covered but also in the context it is used.

As a driving force, and as a measure of success, in improving the techniques and tools we will try to analyse an increasing part of the Ericsson AXD 301 ATM switch as well as try to find applications from other sources to analyse. We will, in a similar manner as we have with the symbolic execution, convert the manually constructed model of the supervisor's behaviour to a formal model in some suitable formalism. The aim of formally presenting the model is both to present it in a clear disambiguated way, but also to enable the use of existing model checking tools such as SPIN[Holzmann 1997] to automate the checking of the fault tolerance properties.

To verify techniques in practise the actual runs of the analysed application could be checked against the possible traces generated by the symbolic execution. The objective would be to ensure that every actual run (there could be several different depending on system parameters) is abstracted by at least one of the runs generated by the symbolic execution.

To be able to navigate the possibly large number of extracted supervision structures we will have to define some order on the structures so that the set of structures can be easily navigated. The order of structures would have to reflect the regularities the construction of similar yet different supervision

structures, such as the number of retries needed before establishing contact between two subsystems.

When trying to extend our work the most important is how to find information beyond the predefined OTP behaviours and in the dynamic part of the applications. We intend to first investigate the approach of combining the symbolic execution of CORE ERLANG terms with finite state methods, where the dynamic parts are approximated by finite models extracted from the code.

Having a framework for showing and simulating the extracted supervision structures as well as showing properties of the structures, we can go the other way and let the designer specify the system in the same framework where it can be simulated and checked. The application can then be validated against the traces of the implementation and the extracted supervision structure can be compared to the specified.

In the furthest extension of this work we would like to use the manually constructed formal models of behaviours as a basis for discussion of how these might be improved and extended, with the ability to model check the model used to ensure that the new behaviours have the intended properties.

Acknowledgements

First and foremost I would like to thank the man of infinite patience: my supervisor Bengt Jonsson. He has over all these years served as an inspiration. My co-supervisor Sven-Olof Nyström has proved an invaluable help, especially during the last frantic part of this thesis creation.

Over the many years of writing this thesis I have had great support and much encouragement from my collegues at Uppsala University, Heriot-Watt University and now finally at Erlang Training and Consulting. You have all made my task so much easier. In particular my previous and current bosses Phil Trinder, Francesco Cesarini and Marcus Taylor.

A number of my friends and collegues have helped with feedback on my research and comments on the thesis, they are in no particular order: Gustaf Naeser, Kostis Sagonas, Martin Leucker, Richard Carlsson, Arnold Pears, Justin Pearson, Ulf Wiger, Thomas Arts and Greg Michaelson.

And finally I would like to thank my friends and family for never loosing faith in me.

Bibliography

[Agrawal 2000] AGRAWAL, G. 2000. Demand-Driven Construction of Call Graphs. In Proceedings of the 9^{th} International Conference on Compiler Construction (CC'00), Volume 1781 of Lecture Notes in Computer Science. Springer-Verlag, 125–140. {11}

[Allen 1978] ALLEN, J. 1978. Anatomy of Lisp. McGraw–Hill Computer Science Series, Feigenbaum, E.A. and Hamming, R.W., Editors. McGraw–Hill. {21, 62}

[Amiranashvili 2002] AMIRANASHVILI, V. 2002. A Rewriting Logic Formalization of Core Erlang Semantics. Master's thesis, Aachen University of Technology, Germany. {12}

[Armstrong 1996] ARMSTRONG, J. 1996. Erlang – A Survey of the Language and its Industrial Applications. In Proceedings of the 9^{th} International Symposium on Industrial Applications of Prolog (INAP'96), Hino, 16–18. {21}

[Armstrong 1997] ARMSTRONG, J. 1997. The Development of Erlang. In Proceedings of the 2^{nd} International Conference on Functional Programming (ICFP'97), Volume 32 of ACM SIGPLAN Notices. ACM Press, 106–203. {19}

[Armstrong 2003] ARMSTRONG, J. 2003. Making reliable distributed systems in the presence of software errors. PhD thesis, Department of Microelectronics and Information Technology, Royal Institute of Technology, Sweden. {9}

[Armstrong 2007] ARMSTRONG, J. 2007. Programming Erlang – Software for a Concurrent World. Pragmatic Programmer. {9}

[Armstrong and Virding 1990] ARMSTRONG, J. AND VIRDING, R. 1990. Erlang - An Experimental Thelephony Programming Language. In Proceedings of the 13^{th} International Switching Symposium, Stockholm. {19}

[Armstrong et al. 1992a] ARMSTRONG, J., VIRDING, R., AND WILLIAMS, M. 1992. Use of Prolog for Developing a new Programming Language. In Proceedings of the 1^{st} International

Conference on the Practical Application of Prolog (PAP'92), London. Association for Logic Programming. {19}

[Armstrong et al. 1992b] ARMSTRONG, J., DÄCKER, B., VIRDING, R., AND WILLIAMS, M. 1992. Implementing a Functional Language for Highly Parallel Real-Time Applications. In Proceedings of the Software Engineering for Telecommunication Switching Systems, Florence. {19}

[Armstrong et al. 1996] ARMSTRONG, J., VIRDING, R., WIKSTRÖM, C., AND WILLIAMS, M. 1996. Concurrent Programming in ERLANG, 2nd edition. Prentice Hall. {9}

[Arora et al. 1996] ARORA, A., GOUDA, M., AND VARGHESE, G. 1996. Constraint satisfaction as a Basis for Designing Nonmasking Fault-Tolerance. Journal of High Speed Networks 5, 3, 293–306. {7}

[Arts and Dam 1999] ARTS, T. AND DAM, M. 1999. Verifying a Distributed Database Lookup Manager Written in Erlang. In FM'99–Formal Methods, Volume I, Proceedings of the 1^{st} World Congress on Formal Methods in the Development of Computing Systems (FM'99), Wing, J.M., Woodcock, J., and Davies, J., Editors. Volume 1708 of Lecture Notes in Computer Science. Springer-Verlag, 682–700. {14, 51}

[Arts and Earle 2001] ARTS, T. AND EARLE, C.B. 2001. Development of a Verified ERLANG Program for Resource Locking. In Proceedings of the 6^{th} International Workshop on Formal Methods for Industrial Critical Systems (FMICS'01), Paris. {12}

[Arts and Giesl 1997] ARTS, T. AND GIESL, J. 1997. Automatically Proven Termination where Simplification Orderings Fail. In Proceedings of TAPSOFT: 7^{th} International Joint Conference on Theory and Practise of Software Development, Volume 1214 of Lecture Notes in Computer Science. Springer-Verlag. {14}

[Arts and Noll 2001] ARTS, T. AND NOLL, T. 2001. Verifying Generic Erlang Client-Server Implementations. In Proceedings of the 12^{th} International Workshop on the Implementation of Functional Languages (IFL'00), Volume 2011 of Lecture Notes in Computer Science. Springer-Verlag, 37–52. {12, 14}

[Arts et al. 2004a] ARTS, T., EARLE, C., , AND DERRICK, J. 2004. Deveolpment of a Verified Erlang Program for Resource Locking. International Journal on Software Tools for Technology Transfer 5, 2–3 (March), 205–220. {12, 15}

[Arts et al. 2004b] ARTS, T., EARLE, C., , AND PENAS, J. 2004. Translating Erlang to μCRL. In In Proceedings of the International Conference on Application of Concurrency to System Design (ACSD2004). {12}

[Aviziensis 1985] AVIZIENSIS, A. 1985. The N-version Approach to Fault-Tolerant Systems. IEEE Transactions on Software Engineering 11, 12, 1491–1501. {5}

[Aït-Kaci 1990] AÏT-KACI, H. 1990. The WAM: A (Real) Tutorial. Digital Press. {19}

[Barklund 1999] BARKLUND, J. 1999. Specification of the STANDARD ERLANG programming language, Final draft (0.6). Ericsson, Computer Science Laboratory. {15}

[Barklund and Virding 1999] BARKLUND, J. AND VIRDING, R. 1999. ERLANG 4.7.3 Reference Manual, Draft (0.7). Ericsson, Computer Science Laboratory, `http://www.erlang.org/download/erl_spec47.ps.gz`. {15}

[Beizer 1990] BEIZER, B. 1990. Software Testing Techniques, 2nd edition. Van Norstrand Reinhold. {2}

[Blau et al. 1999] BLAU, S., ROOTH, J., AXELL, J., HELLSTRAND, F., BUHRGARD, M., WESTIN, T., AND WICKLUND, G. 1999. AXD 301: A new generation ATM switching system. Computer Networks 31, 6, 559–582. {2, 11, 12, 20, 140, 141}

[Blom et al. 2001] BLOM, S., FOKKINK, W., GROOTE, J.F., VAN LANGEVELDE, I., LISSER, B., AND VAN DEN POL, J. 2001. μCRL: A Toolset for Analysing Algebraic Specifications. In Proceedings of the 13^{th} International Conference on Computer Aided Verification (CAV'01), Volume 2102 of Lecture Notes in Computer Science. Springer-Verlag, 250–254. {12}

[Borovanský et al. 1998] BOROVANSKÝ, P., KIRCHNER, C., KIRCHNER, H., P. E. MOREAU, AND RINGEISEN, C. 1998. An Overview of ELAN. In Proceedings of the International Workshop on Rewriting Logic and its Applications, Volume 15 of Electronic Notes in Theoretical Computer Science. Elsevier Science. {12}

[Brown and Sahlin 1999] BROWN, L. AND SAHLIN, D. 1999. Extending Erlang for Safe Mobile Code Execution. In Proceedings of the 2^{nd} International Conference on Information and Communication Security (ICICS'99), Volume 1726 of Lecture Notes in Computer Science. Springer-Verlag, 39–53. {21}

[Carlsson 2001] CARLSSON, R. 2001. An Introduction to Core Erlang. In Proceedings of PLI'01 Erlang Workshop, Florence, Italy, September. {11, 16, 51}

[Carlsson et al. 2000] CARLSSON, R., GUSTAVSSON, B., JOHANSSON, E., LINDGREN, T., NYSTRÖM, S.-O., PETTERSSON, M., AND VIRDING, R. 2000. Core ERLANG 1.0 language specification. Technical report 2000-03, Department of Information Technology, Uppsala University, Sweden. {51}

[Carlsson et al. 2006] CARLSSON, R., SAGONAS, K., AND WILHELMSSON, J. 2006. Message analysis for concurrent programs using message passing. ACM Transactions on Programming Languages and Systems (TOPLAS) 28, 4 (July), 715–746. {13, 16}

[Chandra and Toueg 1996] CHANDRA, J. AND TOUEG, S. 1996. Unreliable Failure Detectors for Reliable Distributed Systems. Journal of the ACM 43, 2, 225–267. {8}

[Chandy and Misra 1986] CHANDY, K.M. AND MISRA, J. 1986. How Processes Learn. Distributed Computing 1, 1, 40–52. {8}

[Cheheyl et al. 1981] CHEHEYL, M.H., GASSER, M., HUFF, G.A., AND MILLER, J.K. 1981. Verifying Security. ACM Computing Surveys 13, 3, 279–339. {2}

[Claessen and Svensson 2005] CLAESSEN, K. AND SVENSSON, H. 2005. A Semantics for Distributed Erlang. In In Proceedings of the ACM SIGPLAN 2005 Erlang Workshop, Tallinn, Estonia. {15, 16}

[Clarke and Wing 1996] CLARKE, E.M. AND WING, J.M. 1996. Formal Methods: State of the Art and Future Directions. ACM Computing Surveys 28, 4, 626–643. {2}

[Clavel et al. 2003] CLAVEL, MANUEL, DURÁN, FRANCISCO, EKER, STEVEN, LINCOLN, PATRICK, MARTÍ-OLIET, NARCISO, MESEGUER, JOSÉ, AND TALCOTT, CAROLYN. 2003. The Maude 2.0 System. In Rewriting Techniques and Applications (RTA 2003), Number 2706 in Lecture Notes in Computer Science. Springer-Verlag, 76–87. {12}

[Corbett 2000] CORBETT, J.C. 2000. Using Shape Analysis to Reduce Finite-State Models of Concurrent JAVA Programs. ACM Transactions on Software Engineering and Methodology 9, 1, 51–93. {12}

[Cristian 1991] CRISTIAN, F. 1991. Understanding Fault-Tolerant and Distributed Systems. Communications of the ACM 34, 2, 56–78. {6}

[Dam et al. 1998a] DAM, M., FREDLUND, L.-Å., AND GUROV, D. 1998. Compositional Verification of Erlang Programmes. In Proceedings of the 3^{rd} International Workshop on Formal Methods for Industrial Critical Systems (FMICS'98), Amsterdam. CWI. {15, 51}

[Dam et al. 1998b] DAM, M., FREDLUND, L.-Å., AND GUROV, D. 1998. Toward Parametric Verification of Open Distributed systema. In Revised Lectures of the International Symposium "Compositionality: the Significant Difference" (COMPOS'97), Volume 1536 of Lecture Notes in Computer Science. Springer-Verlag, 150–185. {15, 51}

[Dean et al. 1996] DEAN, D., FELTEN, E.W., AND WALLACH, D.S. 1996. Java Security: From HotJava to Netscape and Beyond. In Proceedings of the IEEE Symposium on Security and Privacy. IEEE Computer Society Press. {21}

[Dega 1996] DEGA, J.-L. 1996. The Redundancy Mechanisms of the Ariane 5 Operational Control Center. In Proceedings of the 26^{th} IEEE Symposium on Fault Tolerant Computing Systems (FTCS-26), IEEE Computer Society. IEEE Computer Society Press, 382–386. {5}

[Diehl et al. 2000] DIEHL, S., HARTEL, P., AND SESTOFT, P. 2000. Abstract machines for programming language implementation. Future Generation Computer Systems 16, 7, 739–751. {20}

[Dijkstra 1974] DIJKSTRA, E.W. 1974. Self-stabilizing Systems in Spite of Distributed Control. Communications of the ACM 17, 11, 643–644. {5}

[Dolev et al. 1987] DOLEV, D., DWORK, C., AND STOCKMEYER, L. 1987. On the Minimal Synchronism Needed for Distributed Consensus. Journal of the ACM 34, 1, 77–97. {8}

[Dybvig 1996] DYBVIG, R.K. 1996. The Scheme Programming Language: ANSI Scheme. Prentice-Hall. {21}

[Däcker 2000] DÄCKER, B. 2000. Concurrent Functional Programming for Telecommunications: A Case Study of Technology Introduction. Licentiate thesis, Computer Communication System Laboratory, Department Of Teleinformatics, Royal Institute of Technology, Sweden. {19, 21}

[Ericsson Utvecklings AB 2000] ERICSSON UTVECKLINGS AB. 2000. OTP Documentation. {62, 140, 141}

[Feely and Larosse 1998] FEELY, M. AND LAROSSE, M. 1998. Compiling Erlang to Scheme. In Principles of Declarative Programming, Proceedings of the 10^{th} International Conference on Programming Languages, Implementations, Logics and Programs (PLILP'98), Volume 1490 of Lecture Notes in Computer Science. Springer-Verlag, 300–318. {21}

[Feely et al. 1999] FEELY, M., PICHÉ, P., BEAULIEU, S., LAROSSE, M., AND LATENDRESSE, M. 1999. Status Report on the ETOS Erlang to Scheme Compiler. In Proceedings of the 5^{th} International ERLANG/OTP Users Conference (EUC'99). Ericsson Utvecklings AB. {21}

[Fernandez et al. 2000] FERNANDEZ, J.-C., GARAVEL, H., KERBRAT, A., MOUNIER, L., MATEESCU, R., AND SIGHIREANU, M. 2000. CADP: A Protocol Validation and Verification Toolbox. In Proceedings of the 8^{th} International Conference on Computer Aided Verification (CAV'96), Volume 1102 of Lecture Notes in Computer Science. Springer-Verlag, 437–440. {12}

[Fredlund 2001] FREDLUND, L.-Å. 2001. A Framework for Reasoning About ERLANG Code. PhD thesis, Department of Microelectronics and Information Technology, Royal Institute of Technology, Sweden. {14, 15, 16, 51}

[Fredlund and Earle 2006] FREDLUND, L.-Å. AND EARLE, C. B. 2006. Model Checking Erlang Programs: The Functional Approach. In In Proceedings of the ACM SIGPLAN 2006 Erlang Workshop, Portland, USA. {15, 16}

[Fredlund and Svensson 2007] FREDLUND, L.-Å. AND SVENSSON, H. 2007. McErlang: A Model Checker for a Distributed Functional Programming Language. In Proceedings of the ICFP '07 conference, Volume 42 of ACM SIGPLAN Notices. ACM Press, 125–136. {15, 16}

[Fredlund et al. 2003] FREDLUND, L.-Å., GUROV, D., NOLL, T., DAM, M., ARTS, T., AND CHUGUNOV, G. 2003. A verification tool for ERLANG. International Journal on Software Tools for Technology Transfer 4, 4, 405–420. {14}

[Gamma et al. 1998] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1998. Design Patterns. Addison-Wesley Professional Computing Series. Addison-Wesley. {31}

[Giesl and Arts 2001] GIESL, J. AND ARTS, T. 2001. Verification of Erlang Processes by Dependency Pairs. Journal of Applicable Algebra in Engineering, Communication and Computing 12, 1, 39–72. {14}

[Gonzáles 2007] GONZÁLES, S. R. 2007. Erlang Developments in LambdaStream. In Proceedings of the 13^{th} International ERLANG/OTP Users Conference (EUC'07). Ericsson Utveckling AB. {21}

[Gosling et al. 1996] GOSLING, J., JOY, B., AND STEELE, G. 1996. The Java Language Specification. Addison-Wesley. {12}

[Granbohm and Wiklund 1999] GRANBOHM, H. AND WIKLUND, J. 1999. GPRS - General Packet Radio Service. No 2, Ericsson Review. {21}

[Gustafsson 2007] GUSTAFSSON, P. 2007. How to program efficiently with Binaries and Bit Strings. In Proceedings of the 13^{th} International ERLANG/OTP Users Conference (EUC'07). Ericsson Utveckling AB. {20}

[Gärtner 1999] GÄRTNER, F.C. 1999. Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environmnets. ACM Computing Surveys 31, 1, 1–26. {5, 7}

[Hausman 1994] HAUSMAN, B. 1994. Turbo Erlang: Approaching the Speed of C. In Implementation of Logic Programming Systems, Tick, E. and Succi, G., Editors. Kluwer Academic Press, 119–135. {20}

[Hedqvist 1996] HEDQVIST, P. 1996. A Parallel and Multi-threaded Erlang Implementation. Master's thesis, Computing Science Department, Uppsala University, Sweden. {20}

[Herlihy and Wing 1991] HERLIHY, M.P. AND WING, J.M. 1991. Specifying Graceful Degradation. IEEE Transactions on Parallel Distributed Systems 2, 1, 93–104. {4}

[Hinde 2000] HINDE, S. 2000. Use of ERLANG/OTP as a Service Creation Tool for In Services. In Proceedings of the 6^{th} International ERLANG/OTP Users Conference (EUC'00). Ericsson Utvecklings AB. {21}

[Hoare 1969] HOARE, C.A.R. 1969. An Axoimatic Basis for Computer Programming. Communications of the ACM 12, 576–580. {51}

[Holzmann 1991] HOLZMANN, G.J. 1991. Design and Validation of Computer Protocol. Prentice-Hall International. {15}

[Holzmann 1997] HOLZMANN, G.J. 1997. The Model Checker SPIN. IEEE Transactions on Software Engineering 23, 279–295. {15, 151}

[Holzmann 2000] HOLZMANN, G.J. 2000. Logic Verification of ANSI-C Code with SPIN. In Proceedings of the 7^{th} International International SPIN Workshop (SPIN'00), Volume 1885 of Lecture Notes in Computer Science. Springer-Verlag, 131–148. {12}

[Holzmann and Smith 2000] HOLZMANN, G.J. AND SMITH, M.H. 2000. Automating software feature verification. Bell Labs Technical Journal 5, 2, 72–87. {12}

[Huch 1999] HUCH, F. 1999. Verification of ERLANG Programs using Abstract Interpretation and Model Checking. In Proceedings of the 4^{th} International Conference on Functional Programming (ICFP'99), Volume 34 of ACM SIGPLAN Notices. ACM Press, 261–272. {13, 16, 51}

[Huch 2001] HUCH, F. 2001. Model Checking ERLANG Programs – Abstracting the Context-Free Structure. In Proceedings of the 10^{th} International Workshop on Functional and Logic Programming (WFLP'01). {13, 51}

[Huch 2003] HUCH, F. 2003. Model Checking Erlang Programs – LTL-Propositions and Abstract Interpretation. In Proceedings of the 12^{th} International Workshop on Functional and (Constraint) Logic Programming (WFLP'03). {13}

[Intel 2002] INTEL, CORP. 2002. IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture. Intel Corporation, `http://www.intel.com/products/processor/manuals/`. {21}

[Ippolito 2008] IPPOLITO, B. 2008. Ad Serving in Erlang. In Proceedings of the 14^{th} International ERLANG/OTP Users Conference (EUC'08). Ericsson Utveckling AB. {21}

[Jalote 1994] JALOTE, P. 1994. Fault Tolerance in Distributed Systems. Prentice Hall. {6}

[Johansson et al. 2000] JOHANSSON, E., PETTERSON, M., AND SAGONAS, K. 2000. HIPE: A High Performance Erlang System. In Proceedings of the International Conference on Principles and Practises of Delcarative Programmong(PPDP'00), ACM SIGPLAN Notices, 32–43. {21}

[Kernighan and Ritchie 1978] KERNIGHAN, B.W. AND RITCHIE, D.M. 1978. The C Programming Language. Prentice-Hall. {19}

[Kozen 1983] KOZEN, D. 1983. Results on the Propositional μ-Calculus. Theoretical Computer Science 27, 333–354. {14}

[Kuhn 1997] KUHN, D.R. 1997. Sources of Failure in the Public Switched Telephone Network. IEEE Computer 30, 4, 31–36. {1}

[Lamport 1977] LAMPORT, L. 1977. Proving the Correctness of Multiprocess Programmes. IEEE Transactions on Software Engineering 3, 2, 125–143. {7}

[Lamport and Lynch 1990] LAMPORT, L. AND LYNCH, N. 1990. Distributed Computing: Models and Methods. In Handbook of Theoretical Computer Science (Vol. B): Formal Models and Semantics, van Leeuwen, J., Editor. MIT Press, 1157–1199. {8}

[Landin 1964] LANDIN, P.J. 1964. The mechanical evaluation of expressions. Computer Journal 6, 308–320. {62}

[Laprie 1995] LAPRIE, J.-C. 1995. Dependability - Its Attributes, Impairments and Means. In Predictability Dependable Computing Systems, Randell, R., Laprie, J.-C., Kopetz, H., and Littlewood, B., Editors. Basic Research Series. Springer-Verlag, 3–24. {1, 6}

[Laprie et al. 1990] LAPRIE, J.-C., ARLAT, J., BÉOUNES, C., AND KANOUN, K. 1990. Definition and analysis of Hardware-and-Software Fault-Tolerant Architectures. IEEE Computer 23, 7, 39–51. {5}

[Leucker and Noll 2001] LEUCKER, M. AND NOLL, T. 2001. A Distributed Model Checking Tool Tailored Erlang. In Proceedings of PLI'01 Erlang Workshop, Florence, Italy, September. {12}

[Lindahl and Sagonas 2004] LINDAHL, TOBIAS AND SAGONAS, KONSTANTINOS. 2004. Detecting Software Defects in Telecom Applications Through Lightweight Static Analysis: A War Story. In Programming Languages and Systems: Proceedings of the Second Asian Symposium (APLAS'04), Volume 3302 of LNCS. Springer, 91–106. {13}

[Lindahl and Sagonas 2006] LINDAHL, TOBIAS AND SAGONAS, KONSTANTINOS. 2006. Practical Subtype Inference Based on Success Typings. In In Proceedings of the Eight ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'06). ACM Press, 167–178. {13}

[Lindgren 1996] LINDGREN, A. 1996. A Prototype of a Soft Type System for Erlang. Master's thesis, Computing Science Department, Uppsala University, Sweden. {13, 22}

[Lundin 2008] LUNDIN, K. 2008. Inside the Erlang VM, focusing on SMP. In Proceedings of the 14^{th} International ERLANG/OTP Users Conference (EUC'08). Ericsson Utveckling AB. {20}

[Lynch 1996] LYNCH, N. 1996. Distributed Algorithms. Morgan Kaufmann. {8}

[Manna and Pnueli 1992] MANNA, Z. AND PNUELI, A. 1992. The Temporal Logic of Reactive and Concurrent Systems, 2nd edition. Springer-Verlag. {13}

[Marlow and Wadler 1997] MARLOW, S. AND WADLER, P. 1997. A practical subtyping system for Erlang. In Proceedings of the 2^{nd} International Conference on Functional Programming (ICFP'97), Volume 32 of ACM SIGPLAN Notices. ACM Press, 136–149. {13, 22}

[Mattsson et al. 1999] MATTSSON, H., NILSSON, H., AND WIKSTRÖMM, C. 1999. Mnesia - A Distributed Robust DBMS for Telecommunications Applications. In Proceedings of the 1^{st} International Workshop on Practical Aspects of Declarative Languages (PADL'99), Volume 1551 of Lecture Notes in Computer Science. Springer-Verlag, 152–163. {8, 20, 62, 140}

[Millroth 1999] MILLROTH, H. 1999. Mail Robustifier Product based on Erlang/OTP. In Proceedings of the 5^{th} International ERLANG/OTP Users Conference (EUC'99). Ericsson Utveckling AB. {21}

[Mills et al. 1987] MILLS, H.D., DYER, M., AND LINGER, R. 1987. Cleanroom Software Engineering. IEEE Software 4, 5, 19–25. {2}

[Mullaparthi 2005] MULLAPARTHI, C. 2005. Third Party Gateway. In Proceedings of the 11^{th} International ERLANG/OTP Users Conference (EUC'05). Ericsson Utveckling AB. {2, 21}

[Naeser 1997] NAESER, G. 1997. Your First Introduction to SafeErlang. Master's thesis, Computing Science Department, Uppsala University, Sweden. {21}

[Neuhäußer and Noll 2007] NEUHÄUSSER, M. AND NOLL, T. 2007. Abstraction and Model Checking of CORE ERLANG Programs in MAUDE. In Proceedings of the 6^{th} International Workshop on Rewriting Logic and its Applications (WRLA 2006), Volume 176 of Electronic Notes in Theoretical Computer Science, 147–163. {12}

[Nielson and Nielson 1992] NIELSON, H. AND NIELSON, F. 1992. Semantics with Application. Wiley and Sons. {51}

[Nielson et al. 1998] NIELSON, H. R., AMTOFT, T., AND NIELSON, F. 1998. Behaviour Analysis and Safety Conditions: A Case Study in CML. In Proceedings of the 1^{st} International Conference on Fundemantal Approaches to Software Engineering (FASE'98), Volume 1382 of Lecture Notes in Computer Science. Springer-Verlag, 255–269. {12}

[Nilsson and Wikström 1996] NILSSON, H. AND WIKSTRÖM, C. 1996. Mnesia - An Indutrial DBMS with Transactions, Distribution and a Logical Query Language. In Proceedings of the International Symposium on Co-operative Database Systems for Advanced Applications, Kyoto. {20}

[Noll 2001] NOLL, T. 2001. A Rewriting Logic Implementation of Erlang. In Proceedings of the 1^{st} International Workshop on Language Descriptions, Tools and Applications (ETAPS/LDTA'01), Volume 44 of Electronic Notes in Theoretical Computer Science. Elsevier Science. {12}

[Noll 2003] NOLL, T. 2003. Term Rewriting Models of Concurrency: Foundation and Applications. Habilitation thesis, Aachen University of Technology, Germany. {12}

[Noll and Roy 2005] NOLL, T. AND ROY, C.K. 2005. Modeling Erlang in the π-calculus. In In Proceedings of the ACM SIGPLAN 2005 Erlang Workshop, Tallinn, Estonia. {15}

[Nyblom 2000] NYBLOM, P. 2000. The Bit Syntax – The Released Version. In Proceedings of the 6^{th} International ERLANG/OTP Users Conference (EUC'00). Ericsson Utvecklings AB. {20}

[Nyström and Jonsson 2001] NYSTRÖM, J. AND JONSSON, B. 2001. Extracting the Process Structure of Erlang Applications. In Proceedings of PLI'01 Erlang Workshop, Florence, Italy, September. {127}

[Nyström 2003] NYSTRÖM, S.-O. 2003. A soft-typing system for Erlang. In In Proceedings of the ACM SIGPLAN 2003 Erlang Workshop, Uppsala, Sweden. {13}

[Nyström et al. 2008] NYSTRÖM, J.H., TRINDER, P.W., AND KING, D.J. 2008. High-level Distribution for the Rapid Production of Robust Telecoms Software: comparing C++ and Erlang. Concurrency and Computation: Practice & Experience 20, 8, 941–968. {20}

[Peyton Jones 1987] PEYTON JONES, S.L. 1987. The Implementation of Functional Programming Languages. Prentice-Hall International Series in Computer Science, Haore, C.A.R., Editor. Prentice-Hall International. {21}

[Plotkin 1981] PLOTKIN, G.D. 1981. A Structural Approach to Operational Semantics. Technical report DAIMI FN–19, Aarhus University, Denmark. {15, 50}

[Powell et al. 1995] POWELL, D., MARTINS, E., ARLAT, J., AND CROUZET, Y. 1995. Estimators for Fault-Tolerance Coverage Evaluation. In Predictability Dependable Computing Systems, Randell, R., Laprie, J.-C., Kopetz, H., and Littlewood, B., Editors. Basic Research Series. Springer-Verlag, 347–366. {3}

[Reppy 1993] REPPY, J.H. 1993. Concurrent ML: Design, Application and Semantics. In Functional Programming, Concurrency, Simulation and Automated Reasoning, Lauer, P.E., Editor. Volume 693 of Lecture Notes in Computer Science. Springer-Verlag, 165–198. {12}

[Roy et al. 2006] ROY, C.K., NOLL, T., ROY, B., AND CORDY, J.R. 2006. Towards automatic verification of Erlang programs by π-calculus translation. In In Proceedings of the ACM SIGPLAN 2006 Erlang Workshop, Portland, USA. {15}

[Rushby 1994] RUSHBY, J. 1994. Critical System Properties: Survey and Taxonomy. Reliability Engineering and System Safety 43, 2, 189–219. {1, 2}

[SPARC International 1994] SPARC INTERNATIONAL, INC. 1994. The SPARC Architecture Manual (Version 9). Prentice-Hall. {21}

[Sabry and Felleisen 1994] SABRY, A. AND FELLEISEN, M. 1994. Is Continuation-Passing USEFUL for Data Flow Analysis? In Proceedings of the International Conference on Programming Language Design and Implementation (PLDI'94). ACM Press, 1–12. {60}

[Sampath et al. 1995] SAMPATH, M., SENGUPTA, R., LAFORTUNE, S., SINNAMOHIDEEN, K., AND TEKENEKEKZIS, D. 1995. Diagnosability of Discrete-Event Systems. IEEE Transactions on Automatic Control 40, 9, 1555–1575. {5, 11}

[Schlichting and Schneider 1983] SCHLICHTING, R.D. AND SCHNEIDER, F.B. 1983. Fail Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. ACM Transactions on Computer Systems 1, 3, 222–238. {5}

[Schmidt 1986] SCHMIDT, D. A. 1986. Denotational Semantics: A Methodology for Language Development. Wm. C. Brown Publishers. {51}

[Schneider 1993a] SCHNEIDER, M. 1993. Self-stabilization. ACM Computer Surveys 25, 1, 46–67. {5}

[Schneider 1993b] SCHNEIDER, F.B. 1993. What Good are Models and What Models are Good? In Distributed Systems, Mullender, S., Editor. Addison-Wesley Longman Publications, 17–26. {6, 8}

[Schütt et al. 2008] SCHÜTT, T., SCHINTKE, F., AND REINEFELD, A. 2008. Scalaris: Reliable Transactional P2P Key/Value Store. In In Proceedings of the ACM SIGPLAN 2005 Erlang Workshop, Victoria, British Columbia, Canada. {21}

[Schütz 1995] SCHÜTZ, W. 1995. Testing Distributed Real-Time Systems: An Overview. In Predictability Dependable Computing Systems, Randell, R., Laprie, J.-C., Kopetz, H., and Littlewood, B., Editors. Basic Research Series. Springer-Verlag, 284–297. {3}

[Siewiorik and Swartz 1982] SIEWIORIK, D.P. AND SWARTZ, R.S. 1982. The Theory and Practice of Reliable System Design. Digital Press. {5}

[Singhai et al. 1998] SINGHAI, A., LIM, S.B., AND RADIA, S.R. 1998. The SunSCALR Framework for Internet Servers. In Proceedings of the 28^{th} IEEE Symposium on Fault Tolerant Computing Systems (FTCS-28), IEEE Computer Society. IEEE Computer Society Press, 108–117. {5, 7}

[Stenman 2006] STENMAN, E. 2006. Betting on FP (and winning?). In Proceedings of the 13^{th} International ERLANG/OTP Users Conference (EUC'06). Ericsson Utveckling AB. {2, 21}

[Sterling and Shapiro 1986] STERLING, S. AND SHAPIRO, E. 1986. The Art of Prolog: Advanced Programming Techniques. MIT Press. {19}

[Stoy 1977] STOY, J.E. 1977. Denotational Semantics: The Scott-Strachy approach to Programming Language Theory. MIT Press. {51}

[Theel and Gärtner 1998] THEEL, O. AND GÄRTNER, F.C. 1998. On Proving the Stability of Distributed Algorithms: Self-stabilization vs. Control Theory. In Proceedings of the International Computers Conference on System, Signals, Control (SSCC'98), 58–66. {5}

[Thévenod-Fosse et al. 1995] THÉVENOD-FOSSE, P., WAESELYNCK, H., AND CROUZET, Y. 1995. Software Statistical Testing. In Predictability Dependable Computing Systems, Randell, R., Laprie, J.-C., Kopetz, H., and Littlewood, B., Editors. Basic Research Series. Springer-Verlag, 253–272. {3}

[Torstendahl 1997] TORSTENDAHL, S. 1997. Open Telecom Platform. No 1, Ericsson Review. {9, 20}

[Warren 1983] WARREN, D.H.D. 1983. An Abstract Prolog Instruction Set. Technical report SRI Technical Note 309, SRI International, Menlo Park, USA. {19}

[Wiger 2001] WIGER, U. 2001. Four-fold Increase in Productivity and Quality. In Proceedings of the International Workshop Formal Design of Safety Critical Embedded Systems (FemSYS'01). {20}

[Wiklander 1999] WIKLANDER, C. 1999. Verification of Erlang Programmes using Spin. Technical report, Department Of Teleinformatics, Royal Institute of Technology, Sweden. {15}

[Wikström 1994] WIKSTRÖM, C. 1994. Distributed Programming in Erlang. In Proceedings of the 1st International Symposium on Parallel Symbolic Computation, Linz. {19}

[Winskel 1993] WINSKEL, G. 1993. The Formal Semantics of Programming Languages: An Introduction. MIT Press. {51}

[Wong 1998] WONG, G. 1998. Compiling Erlang via C. Technical report SERC–0079, Software Engineering Research Centre, Royal Melbourne Institute of Technology, Australia. {21}

[Wouters 2001] WOUTERS, A.G. 2001. Manual for the μCRL toolset (version 2.07). Technical report To appear???, CWI, Amsterdam. {12}

[Yau and Cheung 1975] YAU, S.S. AND CHEUNG, R.C. 1975. Design of Self-Checking Software. In Proceedings of the International Conference on Reliable Software. IEEE Computer Society Press, 450–457. {5}

[Ödling 1993] ÖDLING, K. 1993. New Technology for Prototyping New Services. No 2, Ericsson Review. {19}

Acta Universitatis Upsaliensis

Uppsala Dissertations from the Faculty of Science

Editor: The Dean of the Faculty of Science

1–11: 1970–1975

12. *Lars Thofelt:* Studies on leaf temperature recorded by direct measurement and by thermography. 1975.
13. *Monica Henricsson:* Nutritional studies on Chara globularis Thuill., Chara zeylanica Willd., and Chara haitensis Turpin. 1976.
14. *Göran Kloow:* Studies on Regenerated Cellulose by the Fluorescence Depolarization Technique. 1976.
15. *Carl-Magnus Backman:* A High Pressure Study of the Photolytic Decomposition of Azoethane and Propionyl Peroxide. 1976.
16. *Lennart Källströmer:* The significance of biotin and certain monosaccharides for the growth of Aspergillus niger on rhamnose medium at elevated temperature. 1977.
17. *Staffan Renlund:* Identification of Oxytocin and Vasopressin in the Bovine Adenohypophysis. 1978.
18. *Bengt Finnström:* Effects of pH, Ionic Strength and Light Intensity on the Flash Photolysis of L-tryptophan. 1978.
19. *Thomas C. Amu:* Diffusion in Dilute Solutions: An Experimental Study with Special Reference to the Effect of Size and Shape of Solute and Solvent Molecules. 1978.
20. *Lars Tegnér:* A Flash Photolysis Study of the Thermal Cis-Trans Isomerization of Some Aromatic Schiff Bases in Solution. 1979.
21. *Stig Tormod:* A High-Speed Stopped Flow Laser Light Scattering Apparatus and its Application in a Study of Conformational Changes in Bovine Serum Albumin. 1985.
22. *Björn Varnestig:* Coulomb Excitation of Rotational Nuclei. 1987.
23. *Frans Lettenström:* A study of nuclear effects in deep inelastic muon scattering. 1988.
24. *Göran Ericsson:* Production of Heavy Hypernuclei in Antiproton Annihilation. Study of their decay in the fission channel. 1988.
25. *Fang Peng:* The Geopotential: Modelling Techniques and Physical Implications with Case Studies in the South and East China Sea and Fennoscandia. 1989.
26. *Md. Anowar Hossain:* Seismic Refraction Studies in the Baltic Shield along the Fennolora Profile. 1989.
27. *Lars Erik Svensson:* Coulomb Excitation of Vibrational Nuclei. 1989.
28. *Bengt Carlsson:* Digital differentiating filters and model based fault detection. 1989.
29. *Alexander Edgar Kavka:* Coulomb Excitation. Analytical Methods and Experimental Results on even Selenium Nuclei. 1989.
30. *Christopher Juhlin:* Seismic Attenuation, Shear Wave Anisotropy and Some Aspects of Fracturing in the Crystalline Rock of the Siljan Ring Area, Central Sweden. 1990.
31. *Torbjörn Wigren:* Recursive Identification Based on the Nonlinear Wiener Model. 1990.
32. *Kjell Janson:* Experimental investigations of the proton and deuteron structure functions. 1991.
33. *Suzanne W. Harris:* Positive Muons in Crystalline and Amorphous Solids. 1991.
34. *Jan Blomgren:* Experimental Studies of Giant Resonances in Medium-Weight Spherical Nuclei. 1991.
35. *Jonas Lindgren:* Waveform Inversion of Seismic Reflection Data through Local Optimisation Methods. 1992.
36. *Liqi Fang:* Dynamic Light Scattering from Polymer Gels and Semidilute Solutions. 1992.
37. *Raymond Munier:* Segmentation, Fragmentation and Jostling of the Baltic Shield with Time. 1993.