Automatic checking of dynamic architectural rules

Verdaasdonk, R.C.A.

Award date:
2007

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

Automatic Checking of Dynamic Architectural Rules

By
R.C.A. Verdaasdonk

Supervisors:
Mark van den Brand (TU/e)
Aleksandra Tesanovic (Philips Research Eindhoven)
Tim Trew (NXP Semiconductors)

Eindhoven, July 2007

Abstract

The use of third-party software in the Philips Consumer Electronics software stack is increasing. Also the complexity of the software is increasing and the time to release new products should decrease. The combination of these factors has led to problems during integration. Especially, problems related to concurrency issues were hard to find during testing due to all possible interleavings. For the MG-R architecture, the architecture used for high-end and mid-range TV's, common made errors have previously been investigated and analyzed, resulting in problem reports. Using these problem reports, dynamic architectural rules were formulated. If these architectural rules hold for software created at Philips and for third-party software, integration would lead to much less problems. At this point there is a checklist containing the architectural rules at Philips Consumer Electronics that is checked manually by the programmers and testers. The goal of this project is to investigate if it is possible to automatically check these rules using model checking techniques. Requirements for the model checker are that it has to accept ANSI C as input and that it has to support concurrency. We have used the code of a mid-range TV from 2002 for the work presented in this thesis. After a series of experiments with the selected model checker, Copper, we can conclude that the current available tools are not mature enough to reach the goal. However, we were able to express several architectural rules in a temporal logic and check them on a model of a small part of the software stack using the model checker Spin.

Keywords Architectural Rules, Model Checking, Concurrency, ANSI C, Embedded Software

Acknowledgements

This thesis is the result of a MSc project done in cooperation with Philips Research Eindhoven. First of all I would like to thank my supervisors for helping and guiding me throughout the project and proofreading this thesis. My supervisor from TU/e was Mark van den Brand, who was always able to give advice when needed. My supervisors from Philips Research were Aleksandra Tesanovic and Tim Trew. Moving to a different group, and even a different company, couldn't stop them from supervising me during the project. Aleksandra always managed to stay positive and encouraged me to continue even after all the setbacks. If you ever need to know anything about Philips software, ask Tim. It seems like he knows everything. Thanks for all the explanations!

Also, I would like to thank my parents for supporting me throughout the years and giving me the opportunity to do this.

Finally, I'd like to thank Marjolein for her support. She had to live with me when I had to finish all kinds of work until late at night. I'm sorry I tortured you by making you proofread this thesis.

Eindhoven, July 2007

Ronald Verdaasdonk

Contents

Chapter 1

Introduction

In this chapter, section 1.1 contains the motivation for the project that led to the work presented in this thesis. The approach for the work done is given in section 1.2. In section 1.3, the outline of the thesis is given.

1.1 Motivation

TV software development for mid-range and high-end TV's at Philips Consumer Electronics (CE) is done using Koala [23] and MG-R [7]. Koala is the software component model and architectural description language that is developed by Philips Research. MG-R is the architecture used for mid-range and high-end TV's. The MG-R architecture consists of several sub-systems that are composed from Koala components. Products can be constructed rapidly by selecting re-usable components. Koala makes sure that the interface specifications of the components are syntactically compatible, but this is not enough to have high confidence in reliability [24]. Problems with reliability arise at system integration. Previously at Philips Research many problem reports (PR), which contain information about faults in the software, were analyzed to create guidelines for creating components with higher reliability after system integration. The created design guidelines were called Design for Plug 'n Play (DPnP). DPnP rules ensure that MG-R sub-systems are composed more rapidly and with high confidence in reliability.

The DPnP approach led to a checklist of architectural rules. These rules constrain the dynamic aspects of the software and must be satisfied by the software. Currently Philips CE uses this checklist during the software development process. The rules are checked manually by the programmers and later by the testers or architects. However, due to the size and complexity of the software it is not possible to check everything manually. It would be a significant improvement to be able to check these rules automatically. The automatical checking of architectural rules is the focus of this thesis work. The goal is to achieve easier integration of new components into products

and minimize the test obligations when requirements change.

The aim of the work performed for this thesis is to investigate the suitability and feasibility of implementing the automatic checking of dynamic architectural rules using model checking techniques.

1.2 Approach

To achieve the goal of the project we investigated available model checkers. Most of them are developed during research projects at universities and have not been evaluated in an industrial setting. Given that the objective is to automatically check architectural rules on existing Philips software, written in C, converting the C code into a model in the input language of the model checker is a lot of work. Moreover, learning how to express the behaviour of a program in the abstractions of the modelling language has a high learning curve for the software developers. Therefore the converting of C code into a model should be performed by the model checker itself in order to make model checking interesting for a company like Philips. Model checkers that have a long tradition, are stable and are frequently used, like NuSMV [8] and Spin [10], do not accept C as input. Since Philips software is mainly multi-threaded, the requirements for a model checker are: accept ANSI C as input and support concurrency through shared variables. There are very few model checkers that support both. Model checkers that accept C as input usually do not support threads or concurrency. The only two model checkers that claimed to comply to the outlined requirements are Copper 2.0 [3] and Blast 2.0 [1]. Copper supports both C code as input and multiple threads and it was claimed to be ready for industry. It was known, however, that there are some limitations on the elements of the C language that it can handle [2, 4]. Blast does not have the known limitations of Copper when it comes to C operations. However, it supports concurrency in a very limited way. Since handling multiple threads is important for checking Philips software, we have chosen to use Copper as our model checker.

To evaluate the capabilities of Copper, we conducted a number of experiments, which exposed new limitations of Copper. These limitations prohibited us to check the selected architectural rules on the entire Philips software stack and even on a scaled down version of part of the software. For this reason, in order to investigate if it was possible to express these rules in temporal logic and use them in a suitable environment, we had to use another model checker. As mentioned before, Spin and NuSMV are good choices if we ignore the requirement that the model checker must accept ANSI C as input. The main difference between Spin and NuSMV is that Spin uses LTL [17] as temporal logic and NuSMV uses CTL [17]. Copper uses LTL, therefore Spin was chosen. Using a model we were able to show that it is possible to check architectural rules with a model checker.

1.3 Outline

Chapter 2, Background introduces general information about the methods used and terminology needed to understand this thesis.

Chapter 3, Preliminaries introduces in-depth information about the tools used and the architectural rules.

Chapter 4, Model Checking C Code describes experiments done in order to find out which limitations have to be dealt with to be able to perform model checking on C code using Copper and the capabilities of Copper regarding model checking C code.

Chapter 5, Checking Architectural Rules on Philips Software describes experiments done to translate selected architectural rules into a temporal logic and check them on the Philips software stack.

Chapter 6, Conclusion gives an evaluation of the results of the experiments and suggestions for future work.

Chapter 2

Background

This chapter gives general information about model checking in section 2.1 and software development at Philips in section 2.2.

2.1 Model Checking

This section introduces the verification method called model checking. More detail is provided about the state space explosion problem and about temporal logic.

2.1.1 Introduction

Usually, software is checked for certain properties using simulation and testing. As size and complexity grow, it becomes much harder to verify the behaviour of software, because methods like simulation and testing cannot exercise all behaviours, particularly for multi-threaded code. If we look at embedded software for a TV, it is important that real-time constraints for video processing are not violated, since that would lead to low quality video streams and thus bad experiences for viewers. It is important that systems are reliable, but it is not possible to test everything using conventional methods. An alternative to simulation and testing is formal verification. Formal verification is a form of verification that is based on mathematics and logic and does an exhaustive exploration of all possible behaviour [17, 20].

Model checking is a method of formal verification by which a desired behavioural property of a reactive system is verified over a given system (the model) through exhaustive enumeration (explicit or implicit) of all the states reachable by the system and the behaviours that traverse through them [17].

Model checking has already been used successfully for verifying hardware systems and communication protocols. However, until recently, software

systems were too complex for model checking. The main reason for this, and also the main challenge of model checking, is the state space explosion problem. Namely, if there is a lot of interaction between components or if data structures can have many different values, the number of states in the model will be very big. The main topic in research on model checking is the reduction of the state space explosion [17]. Another problem for model checking software is the mismatch between the semantics of programming and modelling languages.

Model checking is done by a program called a model checker. In general the process of model checking consists of the following three tasks [17, 20].

1. **Modelling** The system is modelled as a finite state machine using a formalism accepted by the model checker, resulting in a model M.

2. **Specification** A claim is defined using the specification language accepted by the model checker, resulting in a formula φ.

   Claim A property of the system, which is often given in a temporal logic.

   Temporal Logic A logic that can describe the ordering of events in time without introducing time explicitly [17].

3. **Verification** The process of automatic checking whether M satisfies φ.

If a model checker is given model M and claim φ as input, it will give as output either the confirmation that M satisfies φ (written as $M \models \varphi$) or a counter example. The counter example is an execution trace that violates the claim φ.

False negative If a counter example is returned, this can also be caused by an error in the model M. This is called a false negative.

Due to the state space explosion problem it is also possible that M is too large to fit in the memory of the computer. If this happens, there will be no output. In order to be able to check if $M \models \varphi$ in this case, state space reduction techniques must be applied to M.

2.1.2 State Space Explosion

As described before, the main challenge in model checking is the state space explosion problem. Several techniques have been developed to be able to handle systems that would result in models that are too large to fit in the memory of the computer. E.g. abstraction and partial order reduction. Also a great improvement for model checking is symbolic model checking.

Symbolic model checking uses the Ordered Binary Decision Diagram (OBDD) representation for M and is based on the manipulation of boolean formulas [17].

OBDD A canonical form representation for boolean formulas which is often more compact than traditional normal forms and can be manipulated efficiently [17].

Symbolic model checking is not a technique to reduce the number of states. Using OBDD's, much larger state spaces can be handled by the model checker, since the canonical representation of boolean formulas is more efficiently than the normal representation.

The most important technique for handling state space explosion is abstraction. This technique requires a simplification of the model M. Two examples of abstraction are cone of influence reduction and data abstraction.

Cone of influence reduction restricts M to the variables that can influence the variables used in the claim φ.

Data abstraction is the mapping of complex data structures to abstract data structures.

Using data abstraction leads to fewer possible values for data structures and thus fewer states in the model, i.e. mapping integers to even and uneven numbers. The cone of influence technique leads to less variables used in the model M and thus less states. Both types of abstraction lead to a simplification of M. Note however that abstraction can lead to false negatives, because parts of the model are not taken into account. This means adjustments in the abstraction may be necessary [17, 20].

Another technique for handling state space explosion is partial order reduction.

Partial order reduction is aimed at reducing the size of the state space that needs to be searched by model checking algorithms by exploiting the commutativity of concurrently executed transitions, which result in the same state when executed in different orders [17].

Another example of a technique is compositional reasoning.

Compositional reasoning constructs a proof outline for the global correctness of a composed system that exploits the modular structure of the system by partitioning the verification into checks of individual modules [20].

2.1.3 Temporal Logic

It has already been mentioned before that the claim φ to be checked on the model M is often given in a temporal logic. A very powerful temporal logic is CTL* [17]. CTL* has two useful subsets, called Computational Tree Logic (CTL) and Linear Temporal Logic (LTL). CTL* formulas describe properties of computation trees and consist of path quantifiers and temporal operators.

Path quantifiers are used to describe the branching structure in a computation tree.

There are two path quantifiers [17]:

A means for all computation paths.

E means for some computation path.

Temporal operators describe properties of a path through the tree. There are five temporal operators [17]:

X ("next") requires that a property holds in the next state of the path.

F ("eventually" or "in the future") requires that a property holds in some state on the path.

G ("always" or "globally") requires that a property holds in all states on the path.

U ("until") combines two properties and requires that the second property holds in a state on the path and in every preceding state, the first property holds.

R ("release") combines two properties and requires that the second property holds along the path up to and including the first state where the first property holds. However the first property is not required to hold eventually.

The syntax of CTL* formulas consists of state formulas and path formulas and can be summarized as follows [26]:

- State formulas are:
  - Atomic propositions
  - Boolean combinations of state formulas
  - Quantified path formulas

- Path formulas are:
  - State formulas
  - Boolean combinations of path formulas
  - Temporal combinations of path formulas

CTL and LTL both are a subset of CTL*, so CTL* has the most expressive power. Unfortunately, CTL* also needs more complex algorithms for model checking. Although CTL and LTL do not have the expressive power of CTL*, they are more useful in practice because there exist efficient algorithms for them. The expressive power of CTL and LTL is incomparable, since they handle branching in the computation tree differently [17]. We focus on LTL here, since that is the temporal logic used by the model checkers we used during the project. The syntax of LTL formulas is given by [26]:

- State formulas are:
  - All-quantified path formulas
- Path formulas are:
  - Atomic propositions
  - Boolean combinations of path formulas
  - Temporal combinations of path formulas

2.2 TV Software Development at Philips CE

In this section it is described how development of TV software at Philips CE is set up. First the Koala component model is introduced and next the relevant parts of the MG-R architecture are briefly described.

2.2.1 Koala

There was a commercial need to both supporting increased diversity in products and shorten development time. Therefore the complexity of the software should decrease. A new strategy for software development was introduced. The strategy was to adopt a product line approach and to use and reuse software components that work within an explicit software architecture [23]. The architecture is the MG-R architecture that is described in the next section. Since existing architectural description languages (ADL) and component models did not satisfy the needs for industrial embedded software products, Philips created their own component model and ADL, the Koala component model and ADL [23].

A Koala model describes components, interfaces and modules, which together can form a configuration [23].

Components are units of design, development, and reuse. Components are depicted as boxes in figure 2.1. Component composition is hierarchical. Basic components are composed into compound components. Figure 2.2 shows a component that is part of the compound component illustrated in figure 2.1.

Interfaces are used by components to communicate with their environment. Three types of interfaces are defined:

- **Provides interfaces** are used by components to provide functionality to other components. These interfaces are shown as boxed arrows entering components in figure 2.1.
- **Requires interfaces** are used to obtain functionality provided by other components. They are depicted with boxed arrows going out of the components.
- **Diversity interfaces** are used to require properties, or parameters, from the configuration. A diversity interface is a special type of requires interface.

Modules represent the actual implementation of functions specified in the interfaces. They can also be considered as components without interfaces that glue interfaces together. In figure 2.1, modules that glue interfaces are depicted. In figure 2.2 a module that represents the actual implementation of functions is shown as a box with an "m" in it.

Configurations A configuration is a composition of components that represents a product or a test configuration. In a configuration all requires interfaces must be connected to one provides interface. A provides interface can be connected to zero or more requires interfaces.

The syntax of interfaces is defined using a simple Koala interface definition language (IDL). In interface definitions prototypes of the functions are listed. The semantics are described in interface data sheets (IDS) in natural language. Components are described using a Koala component description language (CDL). In component descriptions the requires and provides interfaces are listed. Also sub-components and connections between their interfaces are listed [23].

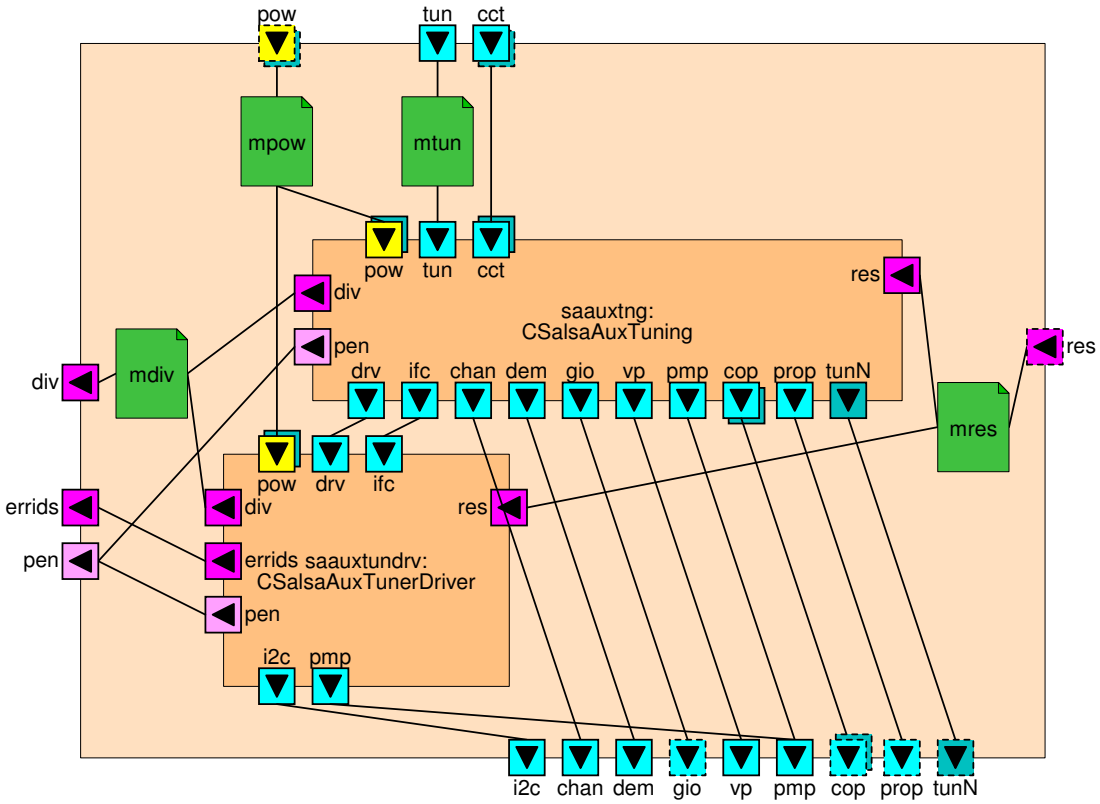

Figure 2.1: Example Koala compound component from 2002 [7]

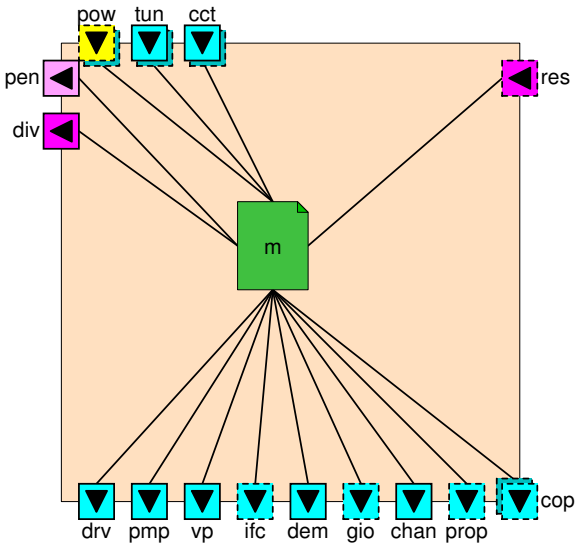

Figure 2.2: Example Koala component from 2002 [7]

2.2.2 MG-R Architecture

As mentioned before, mid-range and high-end TV's developed by Philips CE during the last years are based on Koala and the MG-R architecture. The software is component-based and multi-threaded. Figure 2.3 shows an overview of the sub-system composition for MG-R. As can be seen in the picture, this architecture roughly consists of four parts. The following list shows this division with the sub-systems in figure 2.3 quickly explained.

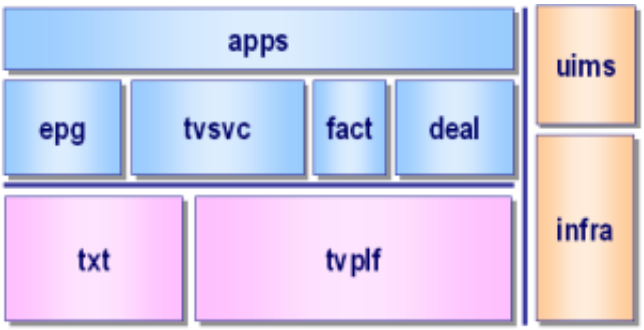

Figure 2.3: MG-R sub-system composition [24]

- Application consists of:
  - **apps**, which contains the applications used by the TV.
- Services consists of:
  - **epg**, which contains services for the Electronic Programming Guide (EPG),
  - **tvsvc**, which contains the services used by the TV,
  - **fact**, which contains services for testing and
  - **deal**, which contains services for dealers.
- Platform consists of:
  - **txt**, which implements Teletext and
  - **tvplf**, which represents the TV Platform, a hardware abstraction layer.
- Infrastructure consists of:
  - **uims**, which contains all functionality for the User Interface (UI) and
  - **infra**, which provides the operating system abstraction.

A sub-system represents a role in the architecture, each of which might be played by a variety of compound components. For instance, there are separate compound components for European and North American variants of tvsvc. The differences between these variants might be obtained by a combination of selecting different sub-components and by changing the settings of diversity interfaces.

Using MG-R it should be possible to easily create products by selecting and configuring components. Since components can be reused in various products, product families can be built faster. The notion of sub-systems ensures consistency between the different components that can play a role in the architecture, facilitating evolution and the allocation of development work to different groups.

The TV software contains many activities. Several threads are used to schedule the activities for execution. Because of resource limitations only a few threads can be used. Due to differences in platforms for TV's, the number of threads available can vary for each product. In order to map all activities to threads, logical threads are used by the components. These logical threads are mapped to physical threads. Using these logical threads, it is also possible to postpone the mapping of activities to physical threads during the development process. The components do not need information about the mapping. The idea of logical and physical threads is implemented using pumps and pump engines. Activities are usually initiated by sending messages to a pump.

Pump A logical message queue with a processing function coupled to it.

Pump engine A physical message queue on a thread that processes the messages.

Components can create pumps on pump engines. A pump engine has a queue of messages. There are two types of pumps: normal pumps and replacing pumps, that handle messages differently.

Normal pumps place messages in the queue.

Replacing pumps replace any previous message queued to the pump.

There are also two types of messages: plain messages and timed messages. A pump engine has a queue for both types of messages [7].

Plain messages are dispatched by the pump engine as soon as possible.

Timed messages are converted to plain messages when their dispatch time is reached.

If a message is sent to a pump, the pump engine on which the pump is created handles the message by using the pump's processing function. Note that if two components with a pump are mapped to the same pump engine, they are thus automatically synchronized [28, 27].

The system reacts to inputs from the user, e.g. the remote control, or hardware interrupts. This usually requires communication between the layers Applications, Services and Platform (see the system composition in figure 2.3). Components either make synchronous calls to other components or asynchronous calls using pumps.

Chapter 3

Preliminaries

This chapter contains detailed information about the model checkers used, Copper and Spin, and the selected architectural rules that are checked. Section 3.1 is about the model checker Copper. Section 3.2 contains information about Spin. Section 3.3 describes the architectural rules.

3.1 Copper

Copper is a model checker based on the tool called MAGIC, which stands for Modular Analysis of proGrams In C [5]. Both are developed by Software Engineering Institute (SEI) at Carnegie Mellon University (CMU). More specifically, Copper is a software model checker for concurrent C programs [3].

In this section we first present the internals of Copper in section 3.1.1. We use a running example to illustrate this. Next, we discuss the capabilities and limitations of Copper in section 3.1.2 as described in the existing Copper documentation [2, 4].

3.1.1 Internals

3.1.1.1 Input / Output

In general, Copper works as depicted in figure 3.1. It takes a C program, and a specification file, including an LTL claim, as input and returns either that the claim is valid for the given C program or that the claim is invalid with a counter example containing a trace that leads to the undesired behaviour. A program to be checked by Copper can be concurrent. Concurrent programs consist of threads or processes. The part of the program that is executed by a thread or process is called, in Copper terminology, a component of the program.

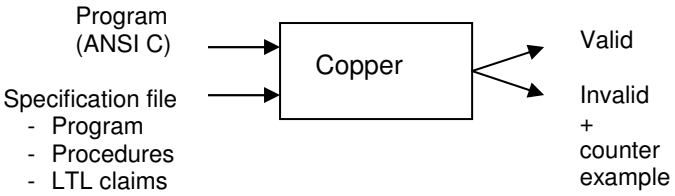

Figure 3.1: Copper input/output

The input to Copper can be summarized as follows:

1. **C Program** The files containing the C code. Note that it has to be ANSI C.

2. **Specification file** A file with the information about the behaviour of the code needed for model checking. The specification file can contain:
   - **Program** The entry points of the components are defined, i.e. the starting functions of the threads, and the specifications for tests are defined.
   - **Procedures** Labelled Transition Systems (LTS) are used to model procedures or library routines that are not included in the available code. Also, predicates can be supplied to guide Copper during model building.
   - **LTL Claims** Temporal logic formulas in LTL that describe the behaviour to be checked. They are coupled to tests in the program section.

As mentioned before, the C program to be model checked must be ANSI C. To analyze a C program that contains GCC or Microsoft Visual Studio C extensions, the tool CIL [22] is used first to translate the code to pure ANSI C [13]. CIL also simplifies the code for the analysis and cleans it up. We use the ANSI C program in listing 3.1 as an example to illustrate how Copper works. The program consists of a main function, lines 27 - 35, and three loops, loop1, loop2 and loop3, which can be seen as concurrent components. As indicated in the specification file in listing 3.2 in line 1, we let loop1 and loop2 run parallel to main. Hence, the program has three concurrent components in this example and loop3 is not used. Note the use of handshakes in line 4. This is a synchronization mechanism in Copper. If more than one component has a handshake with the same action in its code, the components synchronize on this handshake. For example, if loop1 reaches the handshake called "init" in line 4, it has to wait until loop2 and main reach lines 12 and 31 respectively. If a handshake occurs in only one

component, a handshake is seen as an internal action with a name. The use of handshakes as actions can be useful in order to make certain behaviour visible during model checking. The actions can be used in the LTL claims.

Listing 3.1: Example C Program

```
1  int my_global_1, my_global_2, my_global_3;
2
3  void loop1() {
4    __COPPER_HANDSHAKE__("init");
5    while (my_global_1 < 3) {
6      my_global_1++;
7    }
8    __COPPER_HANDSHAKE__("synch");
9  }
10
11 void loop2() {
12   __COPPER_HANDSHAKE__("init");
13   while (my_global_2 < 3) {
14     my_global_2++;
15   }
16   __COPPER_HANDSHAKE__("synch");
17 }
18
19 void loop3() {
20   __COPPER_HANDSHAKE__("init");
21   while(my_global_3 < 3) {
22     my_global_3++;
23   }
24   __COPPER_HANDSHAKE__("synch");
25 }
26
27 int main(void) {
28   my_global_1 = 0;
29   my_global_2 = 0;
30   my_global_3 = 0;
31   __COPPER_HANDSHAKE__("init");
32   __COPPER_HANDSHAKE__("synch");
33
34   exit(0);
35 }
```

In the specification file we have to identify the components and couple specifications to claims. In our example, we chose to run loop1 and loop2 in parallel with main, so we have to specify that. In listing 3.2 a specification file

is shown where we specified this in line 1. Each specification in the program block represents a test. It has a name, contains initial values for variables for each component and a claim. In the example we have two specifications. The first one, in lines 2-3, is called abs01, gives the global variables the value 0 and checks the claim called LTLSpec01. Note that the initial values for variables can be empty. In that case, a value of 1 is used, as in the second specification on line 5. Also note that this specification is a special one. The DefaultSpec can be used for checking special properties without a claim. These possibilities of Copper are further described in section 3.1.2.2. More detailed information about the specification file can be found in the Copper manual [2] and Copper tutorial [4].

The LTSs that model procedures use a notation like the Finite State Process [21] (FSP) notation. As mentioned before, predicates can be supplied for procedures to guide Copper during predicate abstraction. More information about predicates that guide Copper follows in section 3.1.1.3. An example of predicates attached to a procedure can be seen in listing 3.2, lines 10-12.

Besides the special specifications that use DefaultSpec, as on line 5, specifications are coupled to a claim. This claim is written in a special form of LTL, which is called state/event LTL; see standard LTL description in section 2.1.3. Using state/event LTL it is, as mentioned before, possible to use the names of the handshakes as events in claims. Efficient LTL model checking algorithms can be applied without extra costs in space or time for this LTL derivative [20].

Listing 3.2: Example Specification file

```
1  program main, loop1, loop2 {
2    specification abs01,{(my_global_1 == 0) && (my_global_2 == 0)
3                         && (my_global_3 == 0),1,1},LTLSpec01;
4
5    specification abs10,{1,1,1},DefaultSpec;
6  }
7
8  ltl LTLSpec01 { #G #F [my_global_1 == 3 && my_global_2 == 3]; }
9
10 procedure loop1 {
11   predicate (my_global_1 < 3);
12 }
13 procedure loop2 {
14   predicate (my_global_2 < 3);
15 }
16 procedure loop3 {
17   predicate (my_global_3 < 3);
18 }
```

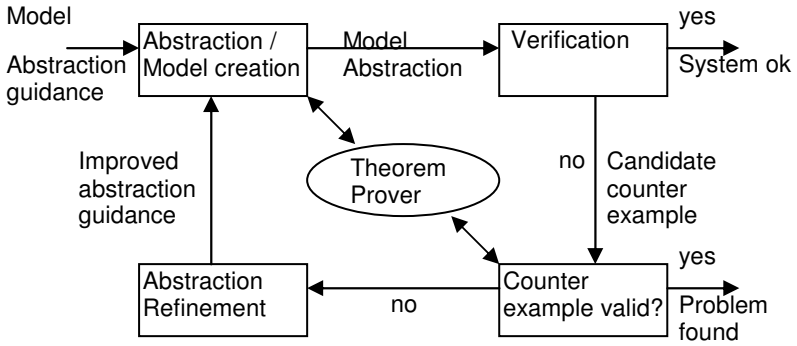

Figure 3.2: CEGAR Framework [3]

3.1.1.2 Abstraction, Verification and Refinement

For the abstraction, verification and refinement of C programs, Copper uses predicate abstraction and the Counter Example-Guided Abstraction Refinement (CEGAR) framework, which is illustrated in figure 3.2. As described in section 2.1, the main problem in formal verification is the state space explosion. In order to reduce this problem Copper uses a modular approach for abstraction, counter example validation and refinement. Also, two-level abstraction refinement is used and predicate minimization during predicate abstraction [13, 3].

The model constructed by Copper is guaranteed to be a conservative abstraction of the original system, meaning that each behaviour in the original system is represented by some behaviour in the model, although the model may, and usually will, contain more behaviour. As a result, if during verification the model satisfies the claim, so does the original system [15]. Since the model may contain more behaviour, counter examples for the abstract model might not be counter examples for the original model. In this case, the counter example is a false negative, or a spurious counter example. This is illustrated in figure 3.2.

If the claim is not satisfied by the model, the counter example is examined to find out if it is a real counter example or a spurious one. If it is a real counter example, Copper exits and returns the result. If it a spurious counter example, Copper uses CEGAR, as shown in figure 3.2.

Based on the type of counter example either predicate refinement is applied or action-guided refinement to eliminate the spurious counter example. In the first case, a new set of predicates is created using predicate minimization and afterwards also action-guided refinement is applied. In the second case, the refinement leads to a finer state aggregation [13].

3.1.1.3 Model Creation

In order to be able to run model checking algorithms on the C code, Copper has to create a suitable model from the code. In Copper the C code is translated into an LTS. The translation from the code to a model is done in several steps, which is established below.

1. The C program is parsed.
2. The program is divided into components.
3. A Control Flow Graph (CFG) is created for each component.
4. Predicate abstraction is applied to each component resulting in an abstract model for each component.
5. Action-guided abstraction is applied to each component resulting in an abstract model for each component.
6. The models of the components are put in parallel to complete the model for the overall system.

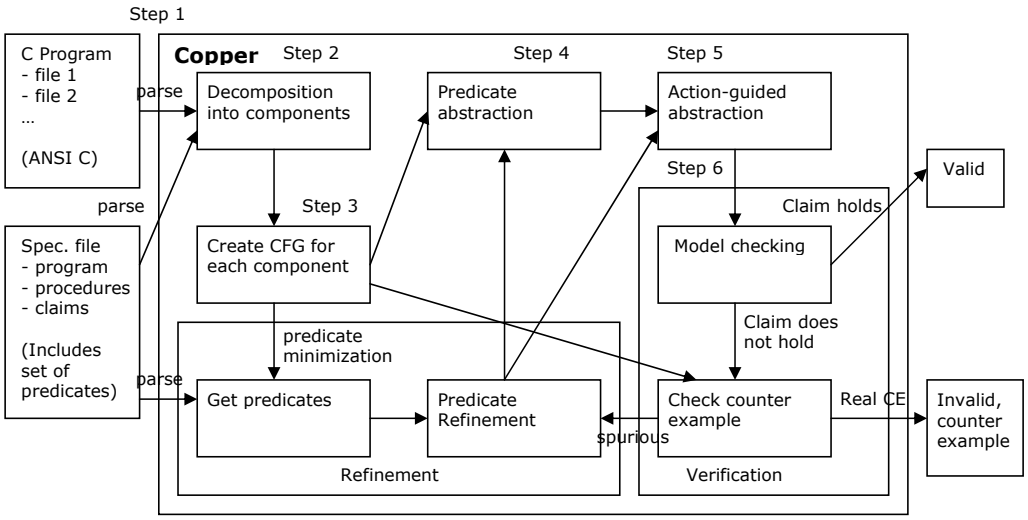

Figure 3.3: Copper overview

Each step of the model creation process is marked in figure 3.3. Also the parts of the internals of Copper related to verification and refinement are marked. Note that in order to keep the figure readable, there is no arrow indicating that the actual model checking needs a claim as input. Also, during refinement either predicate abstraction or action-guided abstraction is applied. In both cases information about the current model is needed, which

is not depicted. Further, predicate abstraction, of course, needs predicates as input. This is also not explicitly depicted for the first run. Finally, the theorem prover, as illustrated in figure 3.2, is not included here.

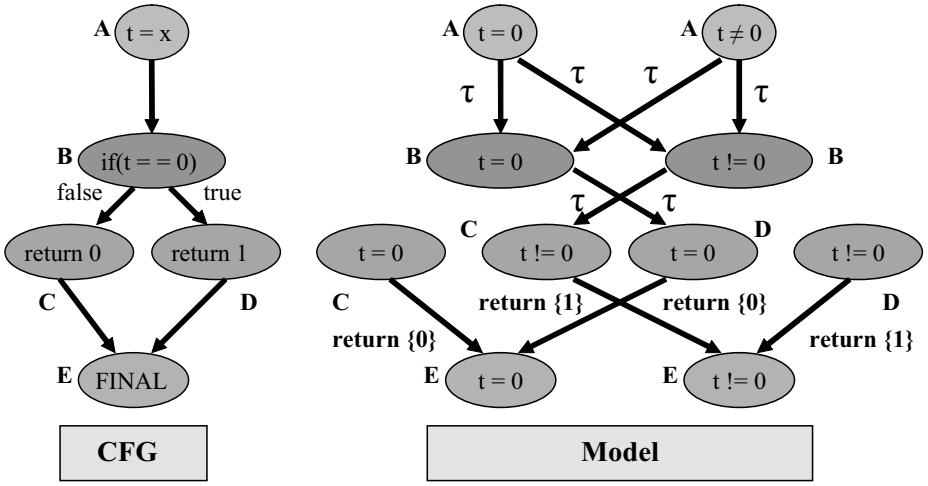

Figure 3.4: Model Extraction through Predicate Abstraction [20]

Step 1 The input C files are parsed using the standard ANSI C grammar and the specification file is parsed.

Step 2 In this step, the program is divided into components using the information about the program in the specification file. It is checked whether the ANSI C program is non-recursive, since recursive behaviour cannot be handled by Copper. Inlining is used to convert a set of procedures with a DAG-like call graph into a single procedure [13].

Step 3 A CFG is created for each component based on the parsed C files. Figure 3.4 contains an example of a CFG. On the left we see a CFG of a simple C program. Two language constructs are eliminated by Copper: indirect function calls via function pointers and indirect assignments to variables using pointer dereferences [13, 12].

Step 4 When Copper searches for the conditions under which state transitions occur, it examines the conditional expressions in the CFG [13, 12]. By default, Copper uses heuristics to select the smallest set of predicates necessary to create a model for which the claims hold. However, the user can supply predicates that they believe are essential to avoid spurious counter examples. The predicates are matched against the conditions on the branches in the code and are added to all the corresponding points in the model. Finding the smallest set of predicates

is also known as predicate minimization, a problem that is an active research field [14]. In the example in listings 3.1 and 3.2 we supply the predicate $my\_global\_1 < 3$ for the procedure loop1. Supplied predicates have to match branches in the code. An example of the resulting model after predicate abstraction can be seen in figure 3.4. If we define a predicate "$t = 0$", the resulting model after applying predicate abstraction on the example CFG from step 3, which is given on the left in figure 3.4, is given on the right in figure 3.4. The transitions are labelled with actions that represent synchronization events, return values of procedure calls or internal actions (τ) [20].

Step 5 This step contains the other level of the compositional 2-level abstraction refinement. The state space after predicate abstraction can still be very large when putting the components in parallel. During action-guided abstraction states are aggregated based on the possible action that can be performed. Together with predicate abstraction this step solves part of the state space explosion problem [13, 12].

Step 6 The model is completed by putting the models of the components resulting from the previous steps in parallel [13].

3.1.2 Limitations and Capabilities

In this section we discuss the limitations and capabilities of Copper when it comes to the actual model checking. They are obtained from the Copper documentation. Also, several interesting command line options are described.

3.1.2.1 Limitations

As we have already seen in sections 3.1.1.1 and 3.1.1.3, there are some limitations on the input that Copper can handle. The input must be pure ANSI C code and it cannot contain a cyclic call graph, since Copper inlines all procedures. There are some other limitations during the model checking as well. Namely, Copper does not support data types such as floats and doubles. Variables of such type are treated as integers. Also, pointers are handled in an unsound manner [4]. Furthermore, arrays are not fully supported at the moment. When using the standard theorem prover bundled with Copper, Simplify [9], not all operators are handled correctly because integers are treated as unbounded quantities. This last problem can be solved however, by using the theorem prover Cprover that comes with Copper. Cprover has one disadvantage: it runs much longer than Simplify on the same input [4, 2].

However, a technique has been developed, called SAT-based predicate abstraction, that makes it possible to model arithmetic overflow, bit vector manipulations, pointers and arrays [3]. This would mean part of the limitations described in the documentation is not applicable to Copper 2.0.

3.1.2.2 Capabilities

We are primarily interested in checking certain LTL claims. There are several possibilities when using Copper, as Copper supports the following types of verification [4]:

1. Simulation conformance between a program and an LTS specification.
2. Trace containment between a program and an LTS specification.
3. A program contains no trace that would cause an LTS specification to reach an ERROR state.
4. A program satisfies an LTL specification.
5. A program does not violate an assertion.
6. A program does not deadlock.

For more information about simulation conformance, trace containment and error state reachability, see [4]. These features can be used to check C code against models. Assertion violation and deadlock detection require the DefaultSpec specification as given in listing 3.2. We are mainly interested in the checking of temporal logic formulas.

3.1.2.3 Command Line Options

Copper is a command line tool. It can take several options to guide its behaviour. When checking an LTL claim, the –ltl option must be used. For assertion violation and deadlock detection, Copper uses a built-in claim. In these cases, the –assert or –deadlock option has to be used. The checking of assertion violations and deadlock detection are features of Copper that can be useful, but are not of interest for this thesis.

There is a command line option called –default, which implies the following command line options [2]:

–**stat** Display statistics at the end.

–**useAllSpecs** Inline all library routine specifications. Overrides the default rule that only library specifications that contain synchronization or global specification actions are inlined.

–**ceShowAll** Show actions and propositional constraints when displaying counter examples.

–**symbolic** Use symbolic predicate abstraction to reduce the number of theorem prover calls.

–**cegar** Do abstraction refinement without predicate optimization.

Other options that are of special interest are [2]:

–**inline** Inline all library routines whose code is available.

–**cprover** Use CProver as theorem prover.

–**noParAssign** Disable parallel assignments.

Without –inline, only the code of the procedures that are marked as the starting points of components in the specification file are used. Other procedures are seen as internal actions that have no effect. It is also possible to indicate which procedures should be inlined in the specification file. In section 3.1.2.1 we already mentioned the (dis)advantages of CProver as a theorem prover. By default Copper transforms a sequence of simple assignments into a single semantically equivalent parallel assignment [2]. –noParAssign makes sure this does not happen since it can lead to unsound behaviour.

Some interesting options that were available for MAGIC, but are not in the current manual of Copper could be used to create visualizations of the intermediate models; see section 4.2 for more information about the results in practice.

3.2 Spin

As mentioned in the introduction, the limitations encountered with Copper meant that we needed Spin to investigate if it was possible to translate the architectural rules described in section 3.3 into a temporal logic and verify them on a model.

Spin does not accept C code as input, but formal models built using Promela (PROcess MEta LAnguage). Promela is a non-deterministic language, loosely based on Dijkstra's guarded command language [16] notation and borrowing the notation for I/O operations from Hoare's CSP [18] language [10]. Processes run in parallel and can communicate using channels or shared memory. Channels can be rendezvous channels or asynchronous communication channels with a buffer for the messages. Using Spin it is possible to do simulations, several types of safety and liveness verification and check LTL properties. The structure of Spin can be summarized as depicted in figure 3.5 and is briefly explained here. It is possible to use Spin from the command line, but using the graphical interface XSpin is highly recommended. Unlike other model checkers, e.g. Copper and Blast, Spin does not feed a model to the actual model checker, but Spin generates an optimized on-the-fly verification program from the specification. This C program is then compiled and executed. If a counter example is found, it can be used as input to the simulator. This way counter examples can be visualized [19].

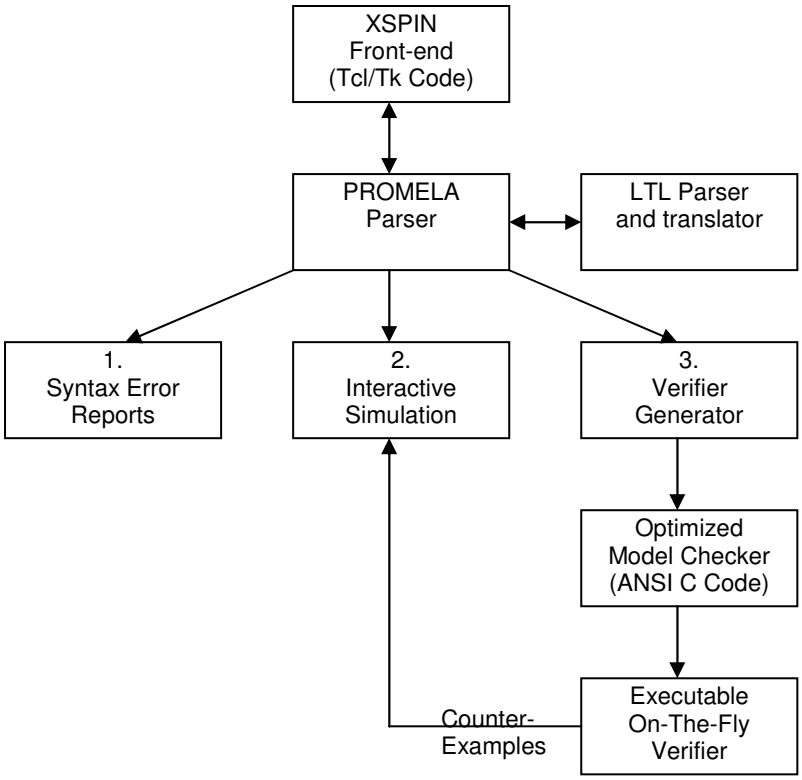

Figure 3.5: The structure of Spin simulation and verification [19]

3.3 Architectural Rules

In this section we describe the general idea of architectural rules and two rules in particular that were used for the work presented in the thesis. First we give information about how the architectural rules were introduced. Next the rules that have been used throughout the project are described in detail.

3.3.1 Introduction

As mentioned in the chapter 1, architectural rules were developed for DPnP. An architectural rule is defined by the property that it aims to maintain. Such a property is called an intent. Concrete rules for a specific architecture are expressed as policies that should meet these intents [24]. Intents are architecture independent (see figure 3.6). Interaction contexts are introduced to map the intents to situations familiar to the developers. Given an interaction context, e.g. function call, general intents are specialized into specific intents. Examples of an intent, a specialized intent and an interaction con-

text are given in section 3.3.2 when describing the selected architectural rules. Policies are created based on the specific intents, the architecture and non-functional requirements. An overview of the DPnP framework for architectural rules can be found in figure 3.6 [24, 25].

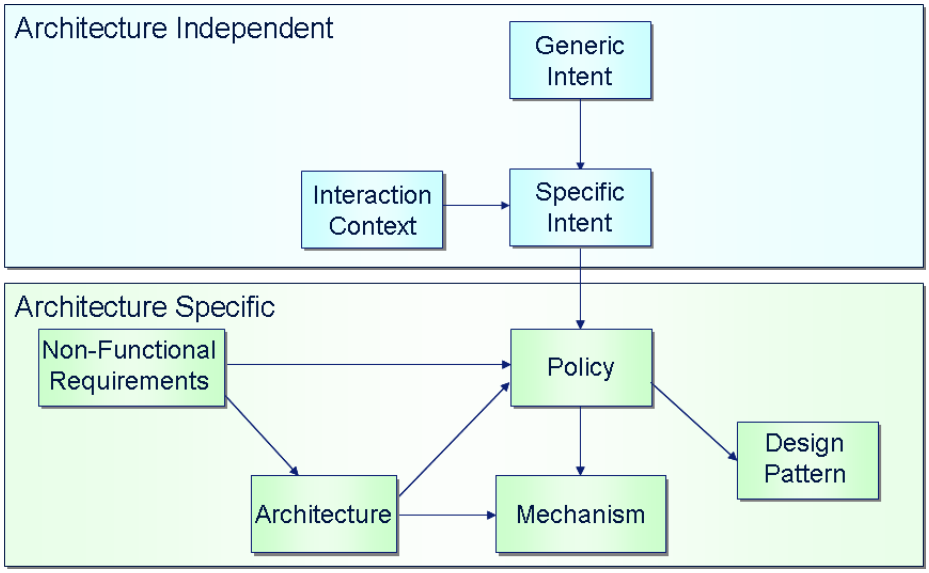

Figure 3.6: Design for Plug and Play: Framework [25]

The architectural rules are used during software development. We can give an overview of the software development life cycle where architectural rules influence the development process in figure 3.7.

Architectural rules are defined using the following steps [24, 25]:

1. Identify principles (intents) that must hold (necessary, but not sufficient) for a system to be reliable.
2. Specialize the intents for a particular interaction context.
3. Relate them to other non-functional requirements and document the design rationale.
4. Specialize them for a particular architecture.
5. Assess alternative design rationales for specific situations.

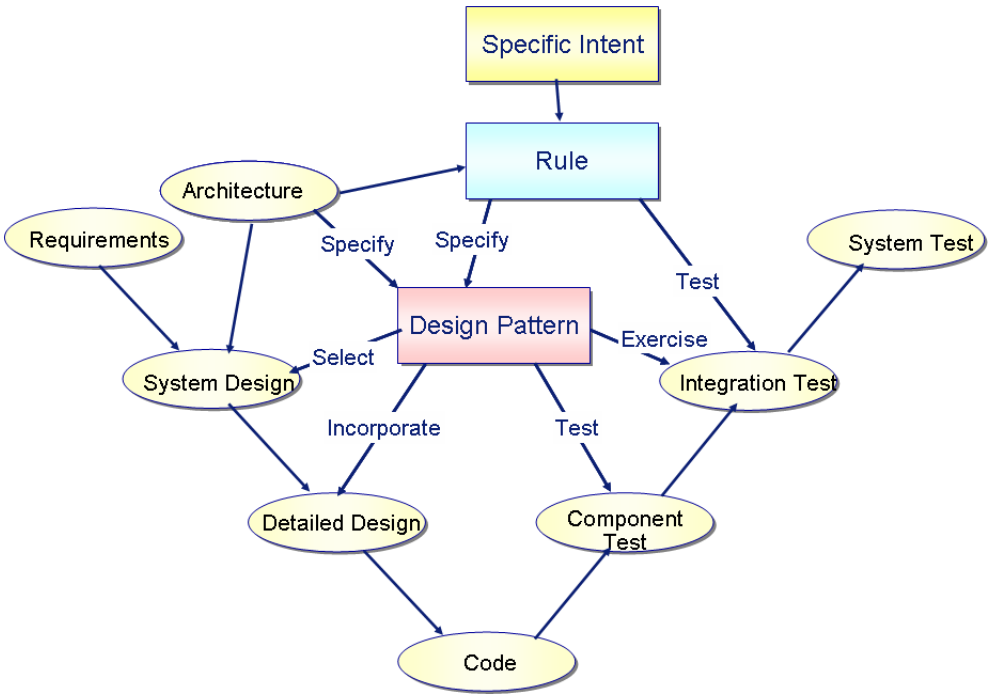

Figure 3.7: Design for Plug and Play: Life Cycle [25]

3.3.2 Example Rules

In the MG-R architecture it is possible that a function initiates an asynchronous activity, and then sets a variable that indicates the function has been called. This variable can then be read as the result of a notification generated by the function call. This mechanism fails if the notification is generated synchronously, e.g. if an optimization is made such that no asynchronous call is made if the activity is unnecessary. In that case, the notification is handled before the synchronous function returns, and thus the value read is incorrect [24]. This behaviour is illustrated in figure 3.8. The first architectural rule we are interested in is:

> A notification should never be generated synchronously on the load of the function that initiates the activity.

This rule follows from the interaction context "Notification generation", the generic intent "Variables must be initialized before they are used" and is related to the non-functional requirement "Predictability". The specific intent for this rule is "Variables that will be read by a notification handler must be set before the handler executes".

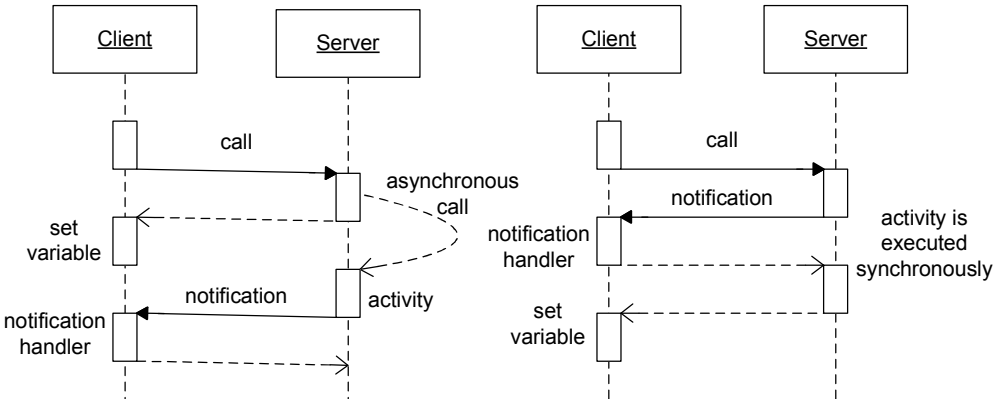

Figure 3.8: Left: Asynchronous call to notification handler. Right: Synchronous call to notification handler.

If a client makes a request to a server that does not lead to any changes, it is possible that the server does not generate a notification. If the client is waiting for this notification, this can lead to blocked behaviour of the system. Therefore, transactions should always get a complete/failure acknowledgement [24]. E.g., consider the MG-R architecture in figure 2.3. If the apps issues a command to change the state of the tvplf, but the tvplf is already in the requested state, it may be necessary for this notification to be generated by the tvsvc, which must detect that the tvplf will not respond. The second rule we are interested in is:

> Transactions should always get a completion/failure acknowledgement.

This rule follows from the interaction context "Function call", the generic intent "Policies of clients and servers must be matched" and is also related to the non-functional requirement "Predictability". The specific intent for the rule is "If a client requires acknowledgement, it must always be generated".

Chapter 4

Model Checking C Code

This chapter contains several experiments done throughout the project using the Copper model checker to further investigate its capabilities and limitations regarding model checking C code. We already know about some limitations, as described in section 3.1.2.1. The experiments are divided into several categories.

Basic tests to test basic C and model checking operations are described in section 4.1.

Command line option tests to test the influence of some optional command line options are described in section 4.2.

Large input tests to test how Copper handles large models are described in section 4.3.

Concurrency tests to test how Copper deals with multiple threads are described in section 4.4.

Language construct tests to compare the execution times of equivalent alternatives for the program used for the concurrency tests are described in section 4.5.

The experiments record Copper's execution time. Since processes running in the background can influence the performance of Copper, all tests are run several times. The average is used as result. All tests of the same category are executed under the same circumstances using a pc with an Intel Core 2 Duo E6400 cpu and 2048MB memory. The results shown are obtained using Copper 2.0.

4.1 Basic Tests

First we run several basic tests, using the program in listing 4.1. This program uses several pointer and array operations. As we have seen in section

3.1.2.1, Copper could have problems performing these operations. However, we would like to know more precisely how many problems arise when model checking C programs with these language constructs. Also, the basic behaviour for checking simple claims using the F (eventually, future) and G (always, global) operators is tested. For Test program 1 we run the tests that are presented in listing 4.2. Note that the handshake in line 3 is added. Even though it is possible to supply initial values in the specification file, Copper can give unexpected values to variables when it starts computing. Therefore, the handshake is added. The name of the action, init, can be used in the LTL claims to convert a claim ϕ into a claim $G(init \rightarrow \phi)$. This can be applied in general to create an initialization phase.

The tests are executed with the default options, the inline option and the noParAssign option. For each test a short summary is given to explain it. Detailed information about the time taken to run the tests can be found in table 4.1.

Listing 4.1: Test program 1

```
1  int main( void ) {
2      int a = 0, b = 0, n = 5, r = 0, *p;
3      __COPPER_HANDSHAKE__("init");
4      while( n > 0 )
5      {
6          /* function call testing */
7          b = triple(n);
8          /* some cast testing */
9          b = (long)b;
10         b = (int)b;
11         a = b + n;
12         /* some pointer testing */
13         p = &a;
14         decrease(p);
15         n--;
16     }
17     r = b + a;
18     exit(0);
19 }
20
21 int triple( int n ) {
22     int cons[5];
23     int i = 0;
24     /* some array testing */
25     for (i = 0; i < 5; i++) {
26         cons[i] = n;
27     }
```

```
28      return 3 * cons[3];
29  }
30
31  void decrease( int *n ) {
32      (*n)--;
33  }
```

Listing 4.2: Test specification 1

```
1   program main {
2       specification abs_01, {1}, Claim1;
3       specification abs_02, {1}, Claim2;
4       specification abs_03, {1}, Claim3;
5       specification abs_04, {1}, Claim4;
6       specification abs_05, {1}, Claim5;
7       specification abs_06, {1}, Claim6;
8       specification abs_07, {1}, Claim7;
9       specification abs_08, {1}, Claim8;
10      specification abs_09, {1}, Claim9;
11      specification abs_10, {1}, Claim10;
12      specification abs_11, {1}, Claim11;
13      specification abs_12, {1}, Claim12;
14      specification abs_13, {1}, Claim13;
15      specification abs_14, {1}, Claim14;
16  }
17
18  ltl Claim1  { #G (init => #G [P0::n >= 0]); }
19  ltl Claim2  { #G (init => #F [P0::n >= 0]); }
20  ltl Claim3  { #G (init => #G [P0::n == 0]); }
21  ltl Claim4  { #G (init => #F [P0::n == 0]); }
22  ltl Claim5  { #G (init => #G [P0::a == P0::b]); }
23  ltl Claim6  { #G (init => #F [P0::a == P0::b]); }
24  ltl Claim7  { #G (init => #G #F [P0::a == P0::b]); }
25  ltl Claim8  { #G (init => #F [P0::r == 6]); }
26  ltl Claim9  { #G (init => #F [P0::r == (P0::a + P0::b)]); }
27  ltl Claim10 { #G (init => #G #F [P0::r == (P0::a + P0::b)]); }
28  ltl Claim11 { #G (init => #G #F [P0::r == 6]); }
29  ltl Claim12 { #G (init => #G [P0::a >= P0::b]); }
30  ltl Claim13 { #G (init => #F [P0::n <  0]); }
31  ltl Claim14 { #G (init => ((#G [P0::n >= 0]) &
32                             (#G #F [P0::a >= P0::b]))); }
```

Claim1 This claim checks whether n is always greater than 0. This should be valid and indeed Copper returns valid.

Claim2 This claim should be trivially valid, as n is initially already is greater than 0, and should be checked faster than Claim1, which is true (see timing results in table 4.1).

Claim3 This claim should be invalid. The counter example should be computed very quickly, since initially n is set to 5 and thus not equal to 0. This is handled correctly.

Claim4 We now check if eventually n will become 0. It should be 0 after the loop, so this claim should be valid. This is the first test where Copper actually has to do some work. Because there are two loops involved, the main loop and the nested loop in the procedure triple, it takes longer for Copper to compute the result. The claim is said to be valid. Note that, until this point, no pointer or array operations are needed for the result.

Claim5 We check if a is always equal to b. The result should be invalid, as can be concluded from line 10 in listing 4.1. Copper takes rather long to check this (see computation times in table 4.1) and the counter example shows that the array operations are not handled correctly. The variable b is assigned the wrong values.

Claim6 Here we check if a will be equal to b at some point. This claim should be trivially true, since the initial values of a and b are 0. This holds, but the claim is checked slower than Claim2, which is of the same form. This might be caused by the presence of two variables in the claim instead of one.

Claim7 Here we test if at the end of the program, a is always equal to b. This should be true, since a should always evaluate to 3 and so should b. Here we really test the pointer and array operations. It takes a long time to check this claim (see table 4.1), but eventually the result is invalid. This is incorrect. The counter example shows this is due to the wrong values that are assigned to b.

Claim8 As mentioned at Claim7, both a and b should evaluate to 3 and from line 18 in listing 4.1 follows that this claim should be valid. Copper does not find an answer to this claim within an hour. If we look at the simplicity of the program and the results of the other claims, an answer should be given sooner.

Claim9 We check if r will be equal to $a + b$. This claim trivially holds because of the initial values. Copper returns valid, but takes rather long to check it. Again, perhaps this is caused because now three variables are involved.

Claim10 We now test if the claim, which is very similar to Claim9, still holds at the end of the program. It turns out the result is valid, as it should be. The pointer and array operations do not influence the output here, since we are not testing for a specific value. It does take a long time to check this (see table 4.1).

Claim11 We now check if the program always results in a variable r that evaluates to 6. We already know from Claim8 that this might give problems. Again, Copper is not able to give an answer within an hour.

Claim12 Here we check if a is always at least b. When b is assigned the value of triple(n) it should have a value larger than a, so the claim should be invalid. This result is also returned by Copper. However, due to the incorrect values assigned to b, this is correct for the wrong reason.

Claim13 That this claim, n will become less than 0, does not hold should follow rather quickly, but Copper takes a long time to get that result. From the counter example it is apparent that it checks the whole program.

Claim14 This last test could take a while if we look at the structure of the claim and the previous results. For both parts of the claim all states have to be checked. However, Copper manages to give the correct result almost instantly (see table 4.1). This might be for the wrong reason.

If we look at the summaries of the results of the claims listed above we can conclude that Copper does not fully support array operations. Although it might be caused by the array operations, the pointer operations on a seem to give incorrect results as well. The test program was modified to isolate these problems. Although toy example programs showed Copper was able to handle pointer and array operations, it turned out both language constructs were not handled correctly in this test program. Therefore, for now we cannot rely on results regarding pointers and array operations. The basic tests show that when it comes to the actual model checking, the temporal logic and initial values are handled correctly.

In order to make sure Copper can handle the program when we replace the array and pointer operations, a few tests have been redone with a modified version of the program in listing 4.1. The triple function now just returns three times the input parameter and the decrease function returns the value of the input parameter decreased by 1, which is assigned to the variable a. Information about the results is listed in table 4.2. Note that the loop in the function triple is now removed, which should lead to faster running times. All claims are now checked correctly and relatively quickly. The counter examples also show that the computations are correct.

Table 4.1: Results basic tests

Test	Result	Correct	CPU Time (ms)	Th. Prover calls
Claim1	Valid	Yes	218	34
Claim2	Valid	Yes	148	34
Claim3	Invalid	Yes	414	21
Claim4	Valid	Yes	5179	674
Claim5	Invalid	Yes*	19671	651
Claim6	Valid	Yes	757	131
Claim7	Invalid	No	132382	4960
Claim8	?	?	-	-
Claim9	Valid	Yes	2710	256
Claim10	Valid	Yes	435921	43185
Claim11	?	?	-	-
Claim12	Invalid	Yes*	11953	460
Claim13	Invalid	Yes	74773	1579
Claim14	Valid	Yes**	1913	176

* Correct, but for the wrong reason

** Correct, but CPU Time indicates it might be for the wrong reason

Table 4.2: Results modified basic tests

Test	Result	Correct	CPU Time (ms)	Th. Prover calls
Claim4	Valid	Yes	1249	456
Claim5	Invalid	Yes	3617	291
Claim7	Valid	Yes	4202	1702
Claim8	Valid	Yes	2710	1232
Claim10	Valid	Yes	42359	5822
Claim11	Valid	Yes	2627	1232
Claim12	Invalid	Yes	1289	268
Claim13	Invalid	Yes	18094	286
Claim14	Valid	Yes**	898	286

** Correct, but CPU Time indicates it might be for the wrong reason

4.2 Influence of Command Line Options

During the following tests we tried to investigate the influence of command line options on the performance of Copper. We use the modified version of the program in listing 4.1. For the basic tests, the following command line options were used: –default (which implies –useAllSpecs –ceShowAll –stat –cegar –symbolic), –inline and –noParAssign [2]. We concentrate on two tests from the specification file in listing 4.2 that are not trivial, Claim7 and Claim11. Again, a short summary for each test is given below and more information about the running time can be found in table 4.3.

We test the following list of interesting command line options [2]:

Test1 –bp instead of –symbolic

Test2 –autoPred

Test3 –cprover

Test4 no –inline

Test5 –drawPredAbsLTS and –outputModels

Claim4 and Claim10 were used for Test4, because they do not depend on the triple and decrease functions. This leads to the following results:

Test1 Instead of using symbolic predicate abstraction, Copper generates boolean programs via abstraction and uses BDDs for verification; see section 2.1.2 for information on BDDs.

Claim7 Using boolean programs turns out to be 9613% slower than symbolic predicate abstraction for Claim7. It also uses 503% more theorem prover calls.

Claim11 For this claim Copper took 251% more time and 166% more theorem prover calls.

The results are clear. If it has this much influence on a small program, the influence on a large program would be dramatic.

Test2 Without supplied predicates, Copper uses a predicate minimization algorithm to determine the set of predicates to be used. With –autoPred all branches in the code are added as seed predicates.

Claim7 This option has no noticeable influence on the behaviour of Copper.

Claim11 Also for this claim, there is hardly any difference in the results.

In general, more seed predicates means longer running time, because predicate abstraction requires exponential time and memory in the number of predicates, as we have seen in section 3.1.1.3. There is only one branch in the program. This may be the reason why the option –autoPred does not have much influence for this example. Using this option does not seem to be useful for larger programs with a lot of branches.

Test3 By default Copper uses the theorem prover Simplify. As mentioned in section 3.1.2.1, Cprover is an alternative.

Claim7 Claim7 takes much longer to be checked. Using Cprover, Copper is almost 74 times slower.

Claim11 Also for this claim Copper is much slower. It takes over 87 times longer.

It is clear that if Copper takes this long checking a simple C program, this option should not be used for larger programs, unless it is vital that, e.g., the use of floats is handled correctly.

Test4 Instead of inlining all procedures available, only inline the relevant procedures using the specification file. Since we are not interested in the triple and decrease procedures for Claim4 and Claim10, we do not inline the procedures.

Claim4 As expected, Copper is 31% faster if we do not inline all procedures. It also uses 25% less theorem prover calls.

Claim10 For Claim10, Copper is 73% faster and uses 58% less theorem prover calls.

In general, not using this option and specifying the procedures to be inlined in the specification file should lead to faster results. For very large programs, this might be a lot of work though, since all relevant functions should be listed manually.

Test5 Here we do not test the influence of the options on the results, but we try to get an overview of the intermediate models Copper uses internally. FSP processes, CFGs and the resulting LTS after predicate abstraction are drawn. Also the model created at each iteration is saved.

Claim7 We expect that the creation of the models takes some extra time and it does: Copper is 4% slower here.

Claim11 Here Copper is 2% slower

We were mainly interested in the extra output given by Copper. Ideally, the models should give an overview of what actually happens and how Copper handles predicate abstraction. The models after each iteration are not readable. The drawings give more information, but it is not clear how they are created or how they should be interpreted.

Table 4.3: Results command line option tests

Test	Result	Correct	CPU Time (ms)	Th. Prover calls
Test1				
Claim7	Valid	Yes	408124	10262
Claim11	Valid	Yes	9194	3274
Test2				
Claim7	Valid	Yes	4234	1702
Claim11	Valid	Yes	2726	1232
Test3				
Claim7	Valid	Yes	313390	0
Claim11	Valid	Yes	231311	0
Test4				
Claim4	Valid	Yes	859	344
Claim10	Valid	Yes	11593	2469
Test5				
Claim7	Valid	Yes	4367	1702
Claim11	Valid	Yes	2656	1234

4.3 Large Input

Until now we have tested a really simple and small program. We now select a few configurations from the Philips software code base. The configurations are based on the MG-R architecture, which is described in section 2.2.2. We are only interested in the number of files and the size of the code, and therefore the program itself is not of interest here. Information about the time taken for checking each claim can be found in table 4.4. We use the following tests:

Test1 The first test is checking a claim that is trivially valid. Here we see how long it takes to build the model.

Test2 In the second test, we check a claim that becomes true after some initializing procedures are finished. Here we test how Copper handles large models.

These tests are executed with the following systems:

Small system The first system is a relatively small configuration of 1.27MB that consists of 66 files.

Test1 It already takes a while to build the model, but the result is correct.

Test2 Quite soon after the model is created and the actual model checking starts, Copper crashes. This happens due to an error in the parser of Copper.

Medium size system The second system is quite a bit larger. It is a configuration of 8.16MB that consists of 395 files. Here we had to comment out a line because it caused a cyclic call graph.

Test1 Building the model again takes a while, but it is still acceptable. The claim is checked correctly.

Test2 The same error message shows up for the medium size program. It is not clear on what point it occurs and it would take a long time to find out without knowledge of the Copper source code.

Large system The last system is a configuration of 13.3MB that consists of 559 files.

Test1 Copper exposes unexpected behaviour when we feed Copper the 559 files. After having read exactly 512 files, Copper is not able to open and read the other files. The building of the model can thus not even start. We could check this was not due to memory limitations.

Test2 Since Test1 could not be started, Test2 was not executed. It leads to the same result.

If we look at the results above, it is hard to reason about how Copper handles larger input with a great amount of certainty. It cannot open more than 512 files for reading and the smaller programs both result in a crash when we check for something that is not as trivial as checking an initial value. Note that during these tests large parts of the code were not even inlined, because these parts are ignored due to function calls using pointers. This means that the actual models of the configurations tested here are much larger than the models constructed during the experiments.

Table 4.4: Results large input tests

Test	Result	Correct	CPU Time (ms)	Th. Prover calls
Small system				
Test1	Valid	Yes	281281	1144
Test2	Error	-	-	-
Medium size system				
Test1	Valid	Yes	595875	1132
Test2	Error	-	-	-
Large system				
Test1	Error	-	-	-
Test2	Error	-	-	-

4.4 Concurrency

Copper is specifically designed for concurrent message-passing C programs. Therefore we designed tests to check some concurrency issues. For the tests we use the program in listing 3.1. First we test the program, without supplied predicates, with one, two and three concurrent loops. The loops are then extended to count to ten instead of three. Again, the program is tested with one, two and three concurrent loops. The claim checks whether all loops have finished counting at the end of the program. The handshakes are used for synchronization here. We can also use them for actions in claims. This is tested as well. Finally, it is tested whether supplied predicates have an influence on the performance of Copper. As mentioned in section 3.1.1.3, predicates can be supplied to guide Copper during predicate abstraction [4]. Again, all these tests are done with the options –default, –inline and –noParAssign [2]. All results about the execution time can be found in table 4.5. The summary of the results is as follows.

Count to three First we only count to three in every loop. We check whether in the end all used global variables are equal to 3.

- **One loop** This is checked very quickly as expected.
- **Two loops** We already see quite an increase in the running time and the number of theorem prover calls here.
- **Three loops** Here we can really see the exponential increase in time in the number of concurrent components.

Count to ten Next we count to ten in every loop, which means more work for Copper. We check whether in the end all used global variables are equal to ten.

- **One loop** Again, using only one loop is checked rather quickly.
- **Two loops** Here we see a large increase in the running time and number of theorem prover calls.
- **Three loops** Here, the exponential increase in time occurs. Checking three loops takes a very long time. We stopped Copper after an hour. If this example is not checked within an hour, a full product or even a component with three threads would simply take much too long to check.

Besides the test with three loops counting to ten, all tests gave the correct result, as expected. The last test seems not very difficult to check, but Copper was not able to give an answer within an hour. It is very likely that Copper would eventually return the correct result, but we do not know it for sure. Regarding the increased running time the results are as expected.

We now check whether supplying predicates has a positive influence on the performance of Copper. Supplied predicates must be exactly in the same form as the branch it relates to, so we add $my\_global\_1 < 3$ for procedure loop1 e.g. Again, we use the program in listing 3.1. This leads to the following results. Information about the running times is also given in table 4.5.

Count to three Again, we check whether all loops finish counting to three.

One loop It takes almost the same amount of time, slightly less, as without predicates to check the claim with the predicates.

Two loops For two loops we again have a slightly faster running time.

Three loops The same result is shown for three loops.

Count to ten Also with predicates supplied, we let the loops count to ten.

One loop It takes quite a lot more time to check the claim as without the predicates.

Two loops Here, the result is almost equal to the result without supplied predicates

Three loops Again, it takes too long before Copper comes with a result, which makes if difficult to reason about the influence of the predicates.

In general, our experiments indicate that supplying predicates does not have a noticeable influence on the outcome and performance of Copper. It takes the same amount of time to check the claims in this case, although the supplied predicates are the branches of the crucial point in the program to check the claim.

Table 4.5: Results concurrency tests, without and with supplied predicates

Test	Result	Correct	CPU Time (ms)		Th. Prover calls	
Count to 3			Without	With	Without	With
1 Loop	Valid	Yes	499	282	460	282
2 Loops	Valid	Yes	2953	1403	2866	1403
3 Loops	Valid	Yes	21273	5048	21147	5026
Actions	Valid	Yes	929	438		
Count to 10			Without	With	Without	With
1 Loop	Valid	Yes	726	412	1906	311
2 Loops	Valid	Yes	181397	45278	180570	45278
3 Loops	?	?	-	-	-	-
Actions	Valid	Yes	7843	2601		

We do one last test for both counting to three and counting to ten with our program. We check if an "init" action is always followed by a "synch" action. Of course, this should be valid. This is tested using all loops. The results, which are listed in table 4.5, show that this kind of claims is checked much faster. Even the program that counts to ten in three loops gives the correct result after around 8 seconds.

4.5 Language Constructs

In order to be able to model check C programs containing arrays and pointers, we had to translate these statements such that the resulting program was equivalent. While preparing for the experiments described in chapter 5, it turned out that "while" loops with more than just a few iterations and "case" statements with several branches could slow down the process more than expected. In order to get an overview of which language constructs are most or least suitable for model checking with Copper, we changed the program in listing 3.1. During this experiment we also tested how bitwise operations are handled. These operations slow down the process during model checking as well. Counting was implemented for these language constructs and compared with counting using a list of statements. Again, all results about the running time can be found in table 4.6.

We had two counting functions in parallel with a main function similar to the program in listing 3.1. For all constructs, we had two tests: counting to three and counting to ten. As during the concurrency experiments in section 4.4, the claim checks whether all loops have finished counting at the end of the program. We compare the results during this experiment with the test that contains the unrolled loop to form a list of statements. This test will most likely be checked fastest. It led to the following results:

List of statements The loops are unrolled and replaced by three and ten incrementing operations respectively.

- **Count to three** As expected this is checked very fast by Copper.
- **Count to ten** Counting to ten instead of counting to three only leads to a linear increase in the number of statements for the running time, so it is still very fast.

While loops The original while loops.

- **Count to three** Using the while loops, the running time is much longer compared to the list of statements. It takes over twelve times more time to check the claim.

Count to ten Compared to the list of statements, the increase in running time for counting to ten is much larger for the while loops. It takes 62 times longer to count to ten than to count to three. Compared to the list of statements, this test takes even 187 times longer.

If statements The while loops are replaced by nested if statements.

Count to three This test is faster than the while loops test, but still over eight times slower than the test with the list of statements.

Count to ten Here the if statements are quite a bit slower than the while statements and the increase in running time and theorem prover calls is huge.

Bitwise operations Again the loops are replaced by a list of statements, but the incrementing is done using bitwise operations.

Count to three This test is by far the slowest. It takes over 33 times longer to check the claim using bitwise operations compared to normal increments.

Count to ten Checking the claim for counting to ten was not finished after an hour.

The list of statements is much faster than the other language constructs. This is as expected. Both the while loops and the nested if statements contain branches and thus contain choices in the computation path. The bitwise operations are handled correctly by Copper, but the time taken is much too long. It is not possible to model check programs containing bitwise operations within a decent time limit.

Table 4.6: Results language construct tests

Test	Result	Correct	CPU Time (ms)	Th. Prover calls
List of statements				
Count-3	Valid	Yes	218	135
Count-10	Valid	Yes	967	933
While loops				
Count-3	Valid	Yes	2874	1416
Count-10	Valid	Yes	181749	45469
Nested if statements				
Count-3	Valid	Yes	2061	994
Count-10	Valid	Yes	243906	119987
Bitwise operations				
Count-3	Valid	Yes	7327	1685
Count-10	?	?	-	-

Chapter 5

Checking Architectural Rules on Philips Software

In this chapter we discuss experiments done to investigate the possibilities for the automatic checking of architectural rules on Philips software. The selected rules are explained in general in section 3.3.2. Section 5.1 gives information about the test configuration used for the experiments. In section 5.2 we describe the context for the selected architectural rules and the translation of the rules to LTL formulas. The experimental setup is described in section 5.3. We then discuss the experiments done with Copper in section 5.4 and the experiments done with Spin in section 5.5.

5.1 Test Configuration

The original goal of the project was to check dynamic architectural rules on the software of a Philips CE product, a mid-range TV from 2002. Tests and the experiments described in chapter 4 showed that it is not possible to check architectural rules on the whole product using Copper. The software was too big, contained too many files, was too complex and contained too many threads for Copper to handle it. Still, we continue to use Copper to check the architectural rules on a smaller test configuration. We can work around the limitations found in chapter 4 by rewriting language constructs we know cause problems. If it is possible to check the rules on a smaller and modified configuration, it might be possible to use Copper in the future for checking larger configurations if limitations are fixed. After preprocessing the code with CIL, the full configuration contains 338.544 lines of code and consists of 559 files. As for all mentioned lines of code, this means without comments, whitespace, etc.

The software to check was scaled down to a test configuration containing the tvsvc, tvplf and infra sub-systems from the MG-R architecture illustrated in figure 2.3 to find out if Copper is able handle such large models. This configuration contains only two threads besides the main thread, contains 233.492 lines of code and consists of 395 files. Without rewriting, this test configuration could not be handled by Copper. Namely, a model can be built, but checking a relatively simple claim, i.e. checking whether a variable is set after a few initialization procedures, does not return a result after 24 hours. Only part of the model was built, since function calls using pointers are ignored. Function calls using pointers are used by the pump engines to execute pump functions (see section 2.2.2). This means checking the configuration with rewritten code would result in a much larger model. It is very well possible that problems found in chapter 4 have a huge impact on the execution time of Copper here. However, it is not an option to rewrite the whole test configuration.

To find out if it would be possible at all to use Copper for checking architectural rules, we decided to scale down the test configuration to a small component that does not depend much on pointers and array operations, to avoid rewriting work. The component chosen was a component for auxiliary tuner control. It offers a control over processing auxiliary RF antenna signals [7]. The component is part of the TV Platform (tvplf) subsystem in the MG-R architecture, which is illustrated in figure 5.1. The functionality of this component can be summarized as [7]:

- Basic tuning in terms of frequency and channel.
- Search functions to scan the band for the presence of stations.
- Supply information on channels and channel tables.

The original test configuration for the auxiliary tuner component contains 35.792 lines of code and consists of 66 files. A Koala model of part of this test configuration is illustrated in figure 2.1. We concentrated on the tuner component itself in this test configuration. Both selected architectural rules consider communication. Therefore, we also needed the pump engine component to handle the asynchronous communication. After removing parts of the code not necessary for checking the architectural rules and rewriting pointer and array operations, the test configuration was cut down to 9 files containing 2.632 lines of code.

5.2 Architectural Rules

The following two architectural rules were used to investigate whether it is possible to express them in LTL and check them on the test configuration.

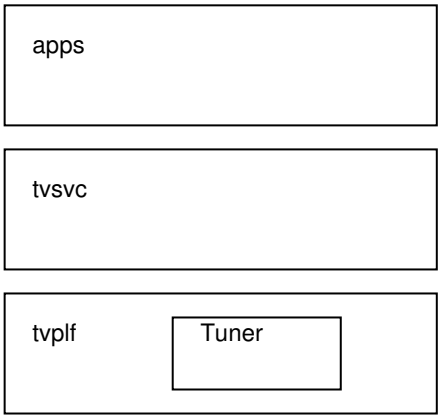

Figure 5.1: Location of the tuner component in the MG-R architecture.

Rule1 A notification should never be generated synchronously on the load of the function that initiates the activity.

Rule2 Transactions should always get a completion/failure acknowledgement.

We discuss Rule1 in section 5.2.1 and Rule2 in section 5.2.2.

5.2.1 First Architectural Rule

Rule1 is the minimal we should be able to check that is still interesting. For this rule we only need the tuner and pump engine components. The behaviour checked for the tuner component is:

> When the tuner receives the request to set a frequency, the notification to indicate whether a station is found on the requested frequency should not be generated synchronously.

The function that sets a frequency is called *setFrequency*. If a synchronous call is made to the notification handler, the notification handler is executed before *setFrequency* has finished. As explained in section 3.3.2, this could lead to undesired behaviour. If a notification is generated asynchronously, using a pump, *setFrequency* finishes before the notification handler is executed. This behaviour is illustrated in figure 5.2. The notification handler is indicated in gray.

In order to check Rule1, we have to translate the desired behaviour of the tuner component into an LTL formula. In Copper it is possible to use the names of actions in an LTL formula, using its state/event LTL. We introduced several handshake statements with actions that are not already

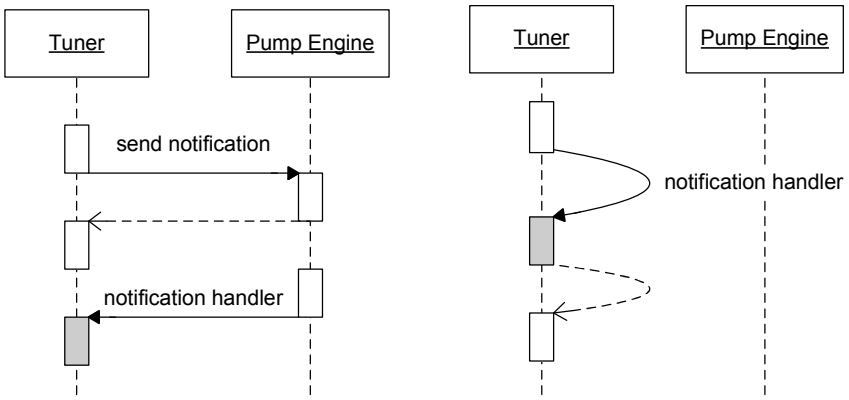

Figure 5.2: Left: An asynchronous call to the notification handler. Right: A synchronous call to the notification handler.

used in the code. If a handshake with the same action would exist in another thread, it would act as a synchronization handshake, which would lead to undesired behaviour. One of these handshakes is added at the start of *setFrequency*. Another is added at the end of the function. A third and fourth handshake are added at the point where the notification handler is executed. One handshake to indicate that a station is found and one to indicate that no station is found. If we look at only one function call to *setFrequency*, we can formulate the following LTL formula:

LTL1 $G(start \rightarrow F(done \wedge F(station\_found \vee station\_not\_found)))$.

The following atomic propositions are used:

start *setFrequency* is called, i.e. the handshake with action "start" is reached.

done *setFrequency* has finished executing, i.e. the handshake with action "done" is reached.

station_found The notification handler returns that a station has been found, i.e. the handshake in the notification handler with action "station_found" is reached.

station_not_found The notification handler returns that no station has been found, i.e. the handshake in the notification handler with action "station_not_found" is reached.

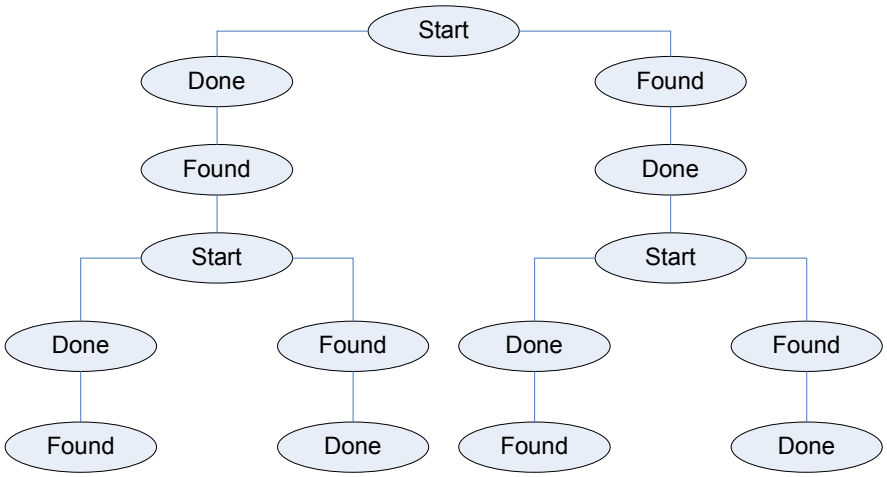

Figure 5.3: All possible orders of actions for two calls to *setFrequency*.

When we want to check whether Rule1 holds for multiple calls to set a frequency, formula LTL1 is not sufficient. If the first call would lead to a synchronous call and the second call to an asynchronous call, the result would still be valid, because the actions of the second call are also in the future from the first "start" action. Therefore the formula LTL1 could return incorrect results for multiple calls. All possible orders of the actions for two calls to *setFrequency* are illustrated in figure 5.3. Here we assumed it is not possible to make a call to *setFrequency* before a previous call is finished and we combined the actions "station_found" and "station_not_found" into Found. The third path in figure 5.3 shows the case where a synchronous call is followed by an asynchronous call. Only the first path should be marked as valid. In order to be able to check Rule1 for multiple calls, we have to introduce a boolean auxiliary variable. This variable indicates that *setFrequency* is being executed. We can now define the desired behaviour as:

> While the function *setFrequency* is being executed, the notification handler must not be executed.

This leads to the following LTL formula:

LTL2 $G(start \rightarrow \neg(p\ U(station\_found \vee station\_not\_found)))$.

Here, the atomic proposition p stands for the added variable having the value *true*.

5.2.2 Second Architectural Rule

In order to check Rule2 we have to add part of the functionality of a program selection component and a program control component. These components

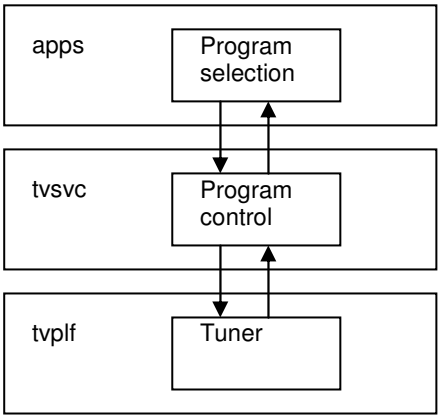

Figure 5.4: Overview components

are part of the Applications (apps) and Services (tvsvc) subsystems of the MG-R architecture. An overview of the location of the components involved in the MG-R architecture is illustrated in figure 5.4. We can then simulate the selecting of a new preset. In practice this happens for example by pressing the P+ button on the remote control. The program control component contains a table that links the presets to frequencies. The specification for the tuner indicates that when a change of frequency has been requested and occurs, a notification should be sent. However, a notification might not be generated if *setFrequency* is called with the current frequency as its parameter. When a change of preset is requested by the program selection component, it expects this notification and waits for it. Because the tuner might not generate a notification, program control should do so when a new request requires no change. However, it is possible that the case in which two presets are mapped to the same frequency is overlooked. In this case switching between these presets will not generate a notification. The correct behaviour is illustrated in figure 5.5. Using the architectural rule we want to check whether the desired behaviour is always obtained, even if the new frequency is the same as the current one. This behaviour is very hard to find without knowledge about the problem.

Again, we have to translate the architectural rule into an LTL formula. The behaviour to check is:

> Every time a new preset is selected, eventually the tuner should generate a notification indicating a station has been found or not.

Checking this for one request to change to another preset is relatively straightforward:

LTL3 $G(preset \rightarrow F(station\_found \vee station\_not\_found))$.

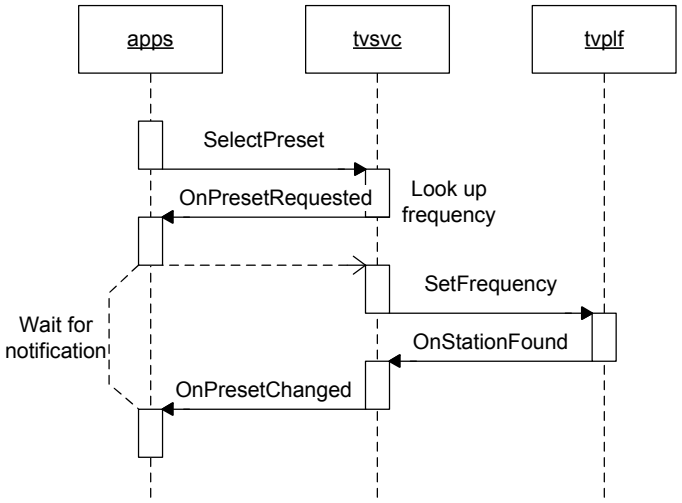

Figure 5.5: Correct behaviour for setting a frequency when a station is found at the selected preset.

The following atomic propositions are used:

preset A new preset is selected.

station_found The notification handler returns that a station has been found.

station_not_found The notification handler returns that no station has been found.

Again, this leads to problems when checking multiple changes in the selected preset. Erroneous behaviour for the first change could be marked as valid if the second change is correct. This is similar to the behaviour explained for multiple calls while checking Rule1 in section 5.2.1. In order to be able to check whether a notification is always generated, we introduce an auxiliary variable, a counter. We increase the counter on every occasion a transaction requires a completion/failure acknowledgement and decrease the counter when the notification is generated. The desired behaviour now becomes:

The counter must always become 0 again.

This results in the following LTL formula:

LTL4 $G(Fc)$.

Here, the atomic proposition c stands for the counter variable having the value 0.

5.3 Experimental Setup

The goal is to check the selected architectural rules on the test configuration using Copper 2.0. We can work around the limitations found in chapter 4 if needed. However, if more problems arise, we continue the experiments using the model checker Spin. First we try to check Rule1 on the test configuration. If this succeeds, we extend the test configuration as described in section 5.2.2 and check Rule2 on the resulting test configuration.

As mentioned in section 5.1, we had to rewrite the tuner and pump engine components to obtain the test configuration for checking Rule1. For both components the following was done:

- Replace all arrays with global structs combined with selection and update functions to prevent problems with arrays. An example of this can be found in listing 5.1.
- Remove the use of pointer operations to prevent problems with pointers.
- Comment out all unused functions to reduce the size of the model.
- Remove coupling with the OS.

For the tuner component we also changed the following:

- Comment out a recursive call, which was not needed to make Copper build the model.
- Comment out debug functions to reduce the size of the model.
- Comment out the unused pumps and pump functions for this test to reduce the size of the model.
- Insert modified pump engine component (see below) so that Copper can handle asynchronous function calls.
- Modify all function calls regarding the pump engine component to match the updated function parameters.

And for the pump engine component:

- Limit the number of pump engines, pumps and messages to be stored by reducing the sizes of the buffers to a minimum in order to reduce the size of the model and reduce the time needed to check LTL formulas.
- Update all functions to replace the array operations with operations on structs and calls to the coupled functions.
- Hardcode a list of pump functions used by the tuner component and couple them to pumps to work around function calls through pointers.

Listing 5.1: Example of equivalent behaviour of an array and a struct with update and select functions.

```
1  typedef struct {
2      int    field00;
3      int    field01;
4      int    field02;
5  } int_array3;
6
7  static int_array3 struct_list;
8  static int array_list[3];
9
10 static int getFieldInt(int_array3 s, int i) {
11     switch(i)
12     {
13         case 0:
14             return s.field00;
15         case 1:
16             return s.field01;
17         case 2:
18             return s.field02;
19         default:
20             assert(0);
21     }
22 }
23
24 static void setStructList(int i, int v) {
25     switch(i)
26     {
27         case 0:
28             struct_list.field00= v;
29             break;
30         case 1:
31             struct_list.field01= v;
32             break;
33         case 2:
34             struct_list.field02= v;
35             break;
36         default:
37             assert(0);
38     }
39 }
40
41
```

```
42 int main(void) {
43     int a = 0;
44     int b = 0;
45
46     array_list[1] = 2;
47     a = array_list[1];
48
49     setStructList(1, 2);
50     b = getFieldInt(struct_list, 1);
51
52     assert(a == b);
53
54     exit(0);
55 }
```

During this phase, the experiments with language constructs, which can be found in section 4.5, were done. Part of the code for the tuner used bitwise operations and large while loops. Checking a claim on the resulting program, only containing the relevant functionality, would still take hours with these language constructs in it. Therefore also modifications regarding these language constructs were done for both components to significantly reduce the time needed to check LTL formulas.

- While loops were replaced by lists of statements where possible.
- Branches in if statements or case statements were limited to as few branches as possible or replaced by hardcoded code.
- Bitwise operations were removed completely. Functions were stubbed at a level just above where these operations were used.

Some unexpected problems were encountered during this preparation phase. Copper returned an error message about illegal types during parsing and then exit. After an extensive search we found out that Copper could not handle the division and modulo operations in the code. Even though these operations have been present in the code since the start of the experiments, the error message did not show up until this point. In section 3.1.2 we already mentioned that not all operators are handled correctly because integers are treated as unbounded quantities when the default theorem prover Simplify is used, but we did not expect this would mean it cannot handle these operations. To be able to start the real experiments, all expressions containing division and modulo operations were replaced by hardcoded precalculated values. The resulting code consists of 5 files and contains 441 lines of code.

5.4 Experiments with Copper

At this point we had a program containing the tuner and pump engine components that could both be executed on the command line and handled by Copper. Several basic LTL formulas could be checked within a few seconds on these heavily modified versions of the tuner and pump engine components.

In our stripped down version of the test configuration it was in fact possible that the pump engine handled the notification before the function finished after an asynchronous call. In the real software this is prevented by using semaphores and locking of the pump engine. Checking claim LTL1 on our code, modified to generate one synchronous call, results in a counter example, as it should. When checked on our code with an asynchronous call, the claim is marked valid. As mentioned, we expected a counter example that illustrates that the pump engine thread finishes before *setFrequency* is finished.

In order to investigate why we did not get the expected counter example, a small experiment was set up. We used two concurrent components, which are shown in listing 5.2. Note the use of handshakes again. The "init" and "synch" handshakes are added for the global synchronization. If we look at the structure of the program, the following interleaving is possible:

- $x = 0$
- $x = 1$
- *read*
- *set*

The specification file for this experiment is given in listing 5.3. We want to check that whenever the action *set* is done, it will be followed by the action *read*. This should give the counter example listed above. Surprisingly, Copper does not give a counter example, but marks the LTL formula valid. This means Copper indeed does not go through all possible interleavings, even with the option –noParAssign. However, we can force Copper to find this counter example by uncommenting the "wait" handshakes. Using this extra synchronization we force Copper to take the interleaving we expected. When we check the same LTL formula, indeed the correct counter example is now returned. This means there is a serious limitation regarding checking concurrent programs.

We could not check Rule1 using Copper with the LTL formula LTL1. However, we tried checking Rule1 with LTL2 as well. Again, this did not return the correct results. Therefore we continued the experiments using Spin.

Listing 5.2: Test program 2

```
1  int x;
2
3  void set(void)
4  {
5      x = 0;
6      __COPPER_HANDSHAKE__("init");
7      x = 1;
8      //__COPPER_HANDSHAKE__("wait");
9      __COPPER_HANDSHAKE__("set");
10     __COPPER_HANDSHAKE__("synch");
11 }
12
13 void read(void)
14 {
15     __COPPER_HANDSHAKE__("init");
16     while( !x )
17     {
18         //skip!
19     }
20     __COPPER_HANDSHAKE__("read");
21     //__COPPER_HANDSHAKE__("wait");
22     __COPPER_HANDSHAKE__("synch");
23 }
```

Listing 5.3: Test specification 2

```
1  program set,read {
2      specification abs_01,{1,1},LTLSpec01;
3  }
4
5  ltl LTLSpec01 { #G (set => #F (read)); }
```

5.5 Experiments with Spin

Since it was not possible to check the selected architectural rules on a heavily modified version of a Philips software component, we came up with an alternative. We have chosen to use the model checker Spin to find out if it possible at all to check the selected architectural rules using model checking.

In order to be able to check an LTL formula using Spin, a model has to be built. We have built a model for the tuner and pump engine components. This model can be found in appendix B.1. It is a translation of the vital parts of the original code into a Promela model. Two channels are used to simulate semaphores to lock the pump engine during the execution of the function to set the frequency. We cannot use actions in LTL formulas in Spin, but it is possible to use labels in the code of the model to indicate the execution has led to that point. This has the same effect as the actions in LTL formulas in Copper. If we check Rule1 for the model using LTL formula LTL1, the results are as expected for one call to *setFrequency*. LTL1 is found to be valid for the model. If we modify the model by creating the possibility that a synchronous call to the notification handler is made, the expected counter example is returned. The LTL formula LTL2 confirms that Rule1 holds for the model and works for any number of calls to *setFrequency*. Again, modifying the model to create the possibility of undesired behaviour leads to expected counter examples. We now know it is possible to check Rule1 using model checking.

To continue the experiments and check Rule2, we had to extend the model. As mentioned in section 5.2.2, to check Rule2 we need parts of the program selection and program control components. With our knowledge of the possible behaviour that is checked by Rule2, it is relatively easy to create circumstances where this behaviour is possible. However, we want to check that a model that was not biased by this knowledge would still find this problem. Therefore we did not directly feed the correct settings, i.e. two presets pointing to the same frequency, to the model. The Promela model builds a lookup table that randomly couples a set of predefined frequencies to some presets. If we succeed in finding the incorrect behaviour, this would also have been found if it was tested using model checking of the code. The resulting model can be found in appendix B.2. The results of checking Rule2 using LTL3 are equal to checking Rule1 using LTL1. It confirms that for one change of preset, Rule2 can be checked using LTL3. Using formula LTL4 it was possible to find the erroneous behaviour for any number of changes of preset. Now, we have also shown it is possible to express Rule2 in temporal logic and use model checking techniques to verify a model of Philips software components. It is possible to check Rule1 on the extended model as well.

Chapter 6

Conclusion

First we give an evaluation of the experiments in section 6.1. Section 6.2 contains several suggestions for future work regarding model checking in industry.

6.1 Evaluation

The goal of the work done for this thesis was to find out if it was possible to automatically check dynamic architectural rules in Philips software components using model checking techniques. We have shown that it is possible to express the selected architectural rules as temporal logic formulas. Using the model checker Spin, we were able to verify the architectural rules on a very abstract model of a small part of the code of a mid-range TV. However, using model checking techniques to verify the code directly is not yet possible with the available tools. We considered two model checkers that are able to have ANSI C code as input and support concurrency, Copper and Blast. Both have their limitations. Blast supports concurrency in a very limited way and we were not able to install the available version in time without help from the developers. Using Copper, several problems were encountered during the experiments:

- Pointer and array operations are not fully supported
- Recursive function calls or recursive call trees are not supported
- Numerical data types other than integers are only supported when the much slower theorem prover Cprover is used.
- The number of input files is limited to 512.
- Supplied predicates do not seem to have influence when looking at the results and Copper's debug information

- Using threads, the time taken to check LTL claims is increasing exponentially in the number of threads, but embedded software in industry usually uses more than two threads.
- Large or nested while loops, case statements and if statements can slow down the process a lot.
- Bitwise operations are supported, but are handled much too slow.
- Division and modulo operations cause parsing errors.
- Several possible paths were not found until we forced Copper to follow these paths while checking LTL claims that involved concurrency.

Besides these problems, we also had to write several scripts to be able to use Copper with multiple files as input. An advantage of Copper is that it has the built-in support for handshakes. Using these constructs, synchronization can be forced and behaviour can be made visible using the names of the handshakes, the actions. These actions can be used in the state/event LTL. In Copper 2.0, pointer and array operations are supported, which is confirmed in toy examples, but still cause problems, as explained in section 4.1. Looking at the rest of the list of limitations, more work needs to be done in order for Copper to be useable in industry. There are still bugs and undesired behaviour and it is not easy to use the tool with multiple files. Since there are no serious alternatives available yet that support concurrency, it seems it is not possible yet to use model checking techniques for verifying software without building your own simulations or models.

6.2 Future Work

In order for companies like Philips to be able to use model checking techniques to verify their software, either the current model checkers must improve significantly or a different type of tool should be developed. An interesting option for future research is using language translations to convert ANSI C code into the input language of a stable model checker, e.g. Spin. Promela, the language used in Spin, already has a syntax that resembles C, although it does not support, e.g., pointers. If such a translation is possible, the advantages of Spin can be used without the need to build models. Some of these tools are described in [11].

A framework that might be useful for research in these language transformations as well is the ASF+SDF Meta-Environment [6]. It can be used for parsing programming languages for further processing the trees or transformation of source code [6].

More interesting information about current research and progress regarding computer aided verification can be found in [11].

Appendix A

Abbreviations

A list of all abbreviations used throughout the thesis:

ADL	Architectural Description Language
CDL	Component Description Language
CE	Consumer Electronics
CEGAR	Counter Example Guided Abstraction Refinement
CFG	Control Flow Graph
CMU	Carnegie Mellon University
CTL	Computational Tree Logic
DPnP	Design for Plug 'Play
EPG	Electronic Programming Guide
FSP	Finite State Process
IDL	Interface Definition Language
IDS	Interface Data Sheet
LTL	Linear Temporal Logic
LTS	Labelled Transition System
OBDD	Ordered Binary Decision Diagram
PR	Problem Report
Promela	PROcess MEta LAnguage
SEI	Software Engineering Institute
UI	User Interface

Appendix B

Promela Models

The models presented in this appendix were used during the experiments as input for the Spin model checker. Section B.1 contains the Promela model used to check the first architectural rule on the tuner component. The Promela model in section B.2 is an extension of the first model. Both selected architectural rules can be checked using this model.

B.1 Tuner Model

Listing B.1: Tuner Promela Model

```
1  /*
2  The msg queue for the pump engine
3  First param is pump
4  Other params of type byte here
5  */
6  chan msg_queue = [2] of {byte, byte, byte}
7  /*
8  channel to simulate request to change freq from environment
9  */
10 chan set_freq = [0] of {byte}
11 /*
12 channels that simulate semaphores
13 */
14 chan x = [0] of {bit}
15 chan y = [0] of {bit}
16
17 /*
18 Global var used in ltl claim
19 */
20 bit p = 0
```

```
21
22 active proctype pump_engine()
23 {
24   byte pump, param1, param2;
25
26   loop: atomic {
27     if
28       :: msg_queue?pump,param1,param2 ->
29         x?1;
30         if
31           :: (pump == 1) -> goto check_vid
32           :: else -> end: y!1
33         fi
34     fi
35   }
36
37   done: y!1;
38   goto loop;
39
40   check_vid: if
41     :: (param1 == 128) -> found: skip; goto done
42     :: (param1 == 64) -> not_found: skip; goto done
43   fi;
44 }
45
46 active proctype env()
47 {
48   set_freq!128;
49   set_freq!64
50 }
51
52 active proctype tuner()
53 {
54   byte freq;
55
56   wait: if
57     :: set_freq?freq ->
58       x?1;
59       p = 1;
60       start:
61       if
62         :: msg_queue!1,freq,0
63         :: goto check_vid
64       fi;
```

```
65          goto done
66      fi;
67
68      done: p = 1 - p;
69      y!1;
70      goto wait;
71
72      check_vid: if
73        :: (freq == 128) -> found: skip; goto done
74        :: (freq == 64) -> not_found: skip; goto done
75      fi
76  }
77
78  active proctype dummy()
79  {
80    do
81      :: timeout ->
82         x!1;
83         y?1
84    od
85  }
```

B.2 System Model

Listing B.2: System Promela Model

```
1  /*
2  Pumps used
3  */
4  #define SetPreset 1
5  #define OnPresetChangeRequested 2
6  #define OnStationFound 3
7  #define OnStationNotFound 4
8  #define OnPresetChangeComplete 5
9  #define OnPresetNotFound 6
10 #define CheckVideoPresence 7
11
12 /*
13 The msg queue for the pump engine
14 First param is pump
15 Other params of type byte here
16 */
17 chan msg_queue = [16] of {byte,byte,byte}
```

```
18 /*
19 Channels to simulate function calls
20 Param can be frequency
21 */
22 chan set_preset = [0] of {byte}
23 chan set_frequency = [0] of {byte}
24 chan station_found = [0] of {byte}
25 chan station_not_found = [0] of {byte}
26 chan preset_change_request = [0] of {byte}
27 chan preset_change_complete = [0] of {byte}
28 chan preset_change_not_found = [0] of {byte}
29 chan check_video_presence = [0] of {byte}
30
31 /*
32 channels that simulate semaphores
33 */
34 chan x = [0] of {bit}
35 chan y = [0] of {bit}
36
37 /*
38 Global vars used in ltl claims
39 */
40 bit p = 0
41 bit q = 0
42 byte counter = 0
43
44 /*
45 Preset table, freq list and freqs in apps
46 */
47 byte table[4]
48 byte freq_list[6]
49 byte current_freq = 0;
50 byte requested_freq = 0;
51
52 proctype pump_engine()
53 {
54   byte pump, param1, param2;
55
56   /*
57   The pump engine keeps looking for messages in the queue
58   */
59   loop: atomic {
60     if
61       :: msg_queue?pump,param1,param2 ->
```

```
62          /*
63          A critical section is needed for the tuner to test
64          the rule about (a)synchronous calls
65          */
66          x?1;
67          if
68            :: (pump == SetPreset) ->
69              set_preset!param1
70            :: (pump == OnPresetChangeRequested) ->
71              preset_change_request!param1
72            :: (pump == OnStationFound) ->
73              station_found!param1
74            :: (pump == OnStationNotFound) ->
75              station_not_found!param1
76            :: (pump == OnPresetChangeComplete) ->
77              preset_change_complete!param1
78            :: (pump == OnPresetNotFound) ->
79              preset_change_not_found!param1
80            :: (pump == CheckVideoPresence) ->
81              check_video_presence!param1
82          fi;
83          y!1
84      fi;
85      goto loop
86    }
87  }
88
89  proctype apps()
90  {
91    byte freq;
92
93    /*
94    Simulate changing to preset 2
95    The counter is incremented for the LTL claim to check that for
96    this function notifications are required
97    */
98    atomic {
99      msg_queue!SetPreset,2,0;
100     counter++;
101   }
102   /*
103   Wait for notification that indicates there will be
104   a change of frequency
105   */
```

```
106   atomic {
107     preset_change_request?freq;
108     requested_freq = freq;
109   }
110   /*
111   Wait for notification that indicates the frequency
112   at the preset has (not) been found
113   The counter indicates the required notification
114   has been received
115   */
116   atomic {
117     if
118       :: preset_change_complete?freq ->
119         current_freq = freq;
120         counter--
121       :: preset_change_not_found?freq ->
122         counter--
123     fi;
124   }
125   /*
126   Another change of preset/frequency request
127   Symmetric to the above
128   */
129   atomic {
130     msg_queue!SetPreset,3,0;
131     counter++;
132   }
133   atomic {
134     preset_change_request?freq;
135     requested_freq = freq;
136   }
137   atomic {
138     if
139       :: preset_change_complete?freq ->
140         current_freq = freq;
141         counter--
142       :: preset_change_not_found?freq ->
143         counter--
144     fi;
145   }
146   /*
147   The variable q can be used for a simple version of
148   the claim that tests that a notification is required
149   In this simulation this becomes unreachable if a
```

```
150     notification is missing
151     */
152     q = 1
153   }
154
155   proctype services()
156   {
157     byte preset;
158     byte freq;
159
160     /*
161     Keep receiving/sending messages from/to other layers
162     */
163     do
164       :: set_preset?preset ->
165         freq = table[preset];
166         msg_queue!OnPresetChangeRequested,freq,0;
167         set_frequency!freq
168       :: station_found?freq ->
169         msg_queue!OnPresetChangeComplete,freq,0
170       :: station_not_found?freq ->
171         msg_queue!OnPresetNotFound,freq,0;
172     od
173   }
174
175   proctype tuner()
176   {
177     byte freq;
178
179     do
180       :: set_frequency?freq ->
181         /*
182         Critical section is needed here to check asynchronous vs
183         synchronous call
184         Same semaphores are used in pump engine
185         */
186         x?1;
187         /*
188         The variable p is used in the LTL claim for (a)synchronous calls
189         It indicates the tun_SetFrequency function is being executed
190         The label start is used in the LTL claim as starting
191         point of the function
192         */
193         p = 1;
```

```
194         start: if
195           /*
196           The first option is an asynchronous call
197           The second option is a synchronous call
198           */
199           :: msg_queue!CheckVideoPresence,freq,0
200           :: check_video_presence!freq
201         fi;
202         /*
203         End of function tun_SetFrequency and critical section
204         */
205         p = 1 - p;
206         y!1
207     od
208 }
209
210 proctype tvplatform()
211 {
212     byte freq;
213
214     do
215       :: check_video_presence?freq ->
216         if
217           /*
218           Do not generate notification if:
219           current frequency == requested frequency
220           */
221           :: (current_freq == freq) -> skip
222           :: else ->
223             if
224               /*
225               Generate notification
226               Possible frequencies are hardcoded between 1 and 6
227               1 - 4 will be found, 5 - 6 not
228               */
229               :: (freq < 5) ->
230                   found: msg_queue!OnStationFound,freq,0
231               :: else ->
232                   not_found: msg_queue!OnStationNotFound,freq,0
233             fi
234         fi
235     od
236 }
237
```

```
238 proctype dummy()
239 {
240   /*
241   Keep handling the semaphores for the critical sections
242   */
243   do
244     ::
245       x!1;
246       y?1
247   od
248 }
249
250 init
251 {
252   byte index = 0;
253
254   /*
255   List possible frequencies
256   */
257   freq_list[0] = 1;
258   freq_list[1] = 2;
259   freq_list[2] = 3;
260   freq_list[3] = 4;
261   freq_list[4] = 5;
262   freq_list[5] = 6;
263
264
265   /*
266   Fill the preset/frequency table in a random way
267   */
268   do
269     :: (index == 4) -> goto done
270     :: else ->
271       if
272         :: table[index] = freq_list[0]
273         :: table[index] = freq_list[1]
274         :: table[index] = freq_list[2]
275         :: table[index] = freq_list[3]
276         :: table[index] = freq_list[4]
277         :: table[index] = freq_list[5]
278       fi;
279       index++
280   od;
281
```

```
282     /*
283     Start all processes
284     */
285     done:
286     atomic {
287       run dummy();
288       run apps();
289       run services();
290       run tvplatform();
291       run tuner();
292       run pump_engine()
293     }
294 }
```

Bibliography

[1] Blast project. http://mtc.epfl.ch/software-tools/blast/.

[2] Copper manual. http://www.sei.cmu.edu/pacc/manual.html.

[3] Copper model checker. http://www.sei.cmu.edu/pacc/copper.html.

[4] Copper tutorial. http://www.sei.cmu.edu/pacc/tutorial.html.

[5] Magic. http://www.cs.cmu.edu/ chaki/magic/.

[6] The meta-environment. http://www.cwi.nl/htbin/sen1/twiki/bin/view/Meta-Environment.

[7] MG-R documentation. Philips Company Restricted.

[8] NuSMV. http://nusmv.irst.itc.it/.

[9] Simplify theorem-prover. http://research.compaq.com/SRC/esc/Simplify.html.

[10] Spin - formal verification. http://spinroot.com/spin/whatispin.html.

[11] *Computer Aided Verification : 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*. Springer, 2007.

[12] Sagar Chaki, Edmund Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in c. In *International Conference on Software Engineering*, pages 385–395, May 2003.

[13] Sagar Chaki, Edmund Clarke, Alex Groce, Joël Ouaknine, Ofer Strichman, and Karen Yorav. Efficient verification of sequential and concurrent C programs. *Formal Methods in System Design (FMSD)*, 25(2–3):129–166, September–November 2004.

[14] Sagar Chaki, Edmund Clarke, Alex Groce, and Ofer Strichman. Predicate abstraction with minimum predicates. In *Proceedings of 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, 2003.

[15] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[16] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[17] Orna Grumberg Edmund M. Clarke, Jr. and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[18] C.A.R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[19] Gerard J. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, May 1997.

[20] J. Ivers and N. Sharygina. Overview of comfort: A model checking reasoning framework. Technical Report TN-018, CMU/SEI, 2004.

[21] Jeff Magee and Jeff Kramer. *Concurrency: state models & Java programs*. John Wiley & Sons, Inc., 1999.

[22] George C. Necula, Scott McPeak, Shree P. Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Compiler Construction: 11th International Conference, CC 2002*, pages 209–265, 2002.

[23] J. Kramer R. van Ommering, F. van der Linden and J. Magee. The Koala component model for consumer electronics software. *Computer*, 33(3):78–85, 2000.

[24] N. Koppalkar T. Trew and M.R. Narasimhamurthy. Architectural rules for design for plug 'n play for MG-R sub-systems. Philips Company Restricted, 2004.

[25] T. Trew. Integration and evolution in product families - making it work for high-end tvs, 2006.

[26] Jaco van de Pol. Lecture 1: The temporal logics ctl*, ctl, ltl - syntax, semantics and fairness -. http://www.cwi.nl/ vdpol/amc.html.

[27] R. van Ommering. On pumps and pump engines, 1999. Philips Company Restricted.

[28] R. van Ommering. Building product populations with software components. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 255–265. ACM Press, 2002.