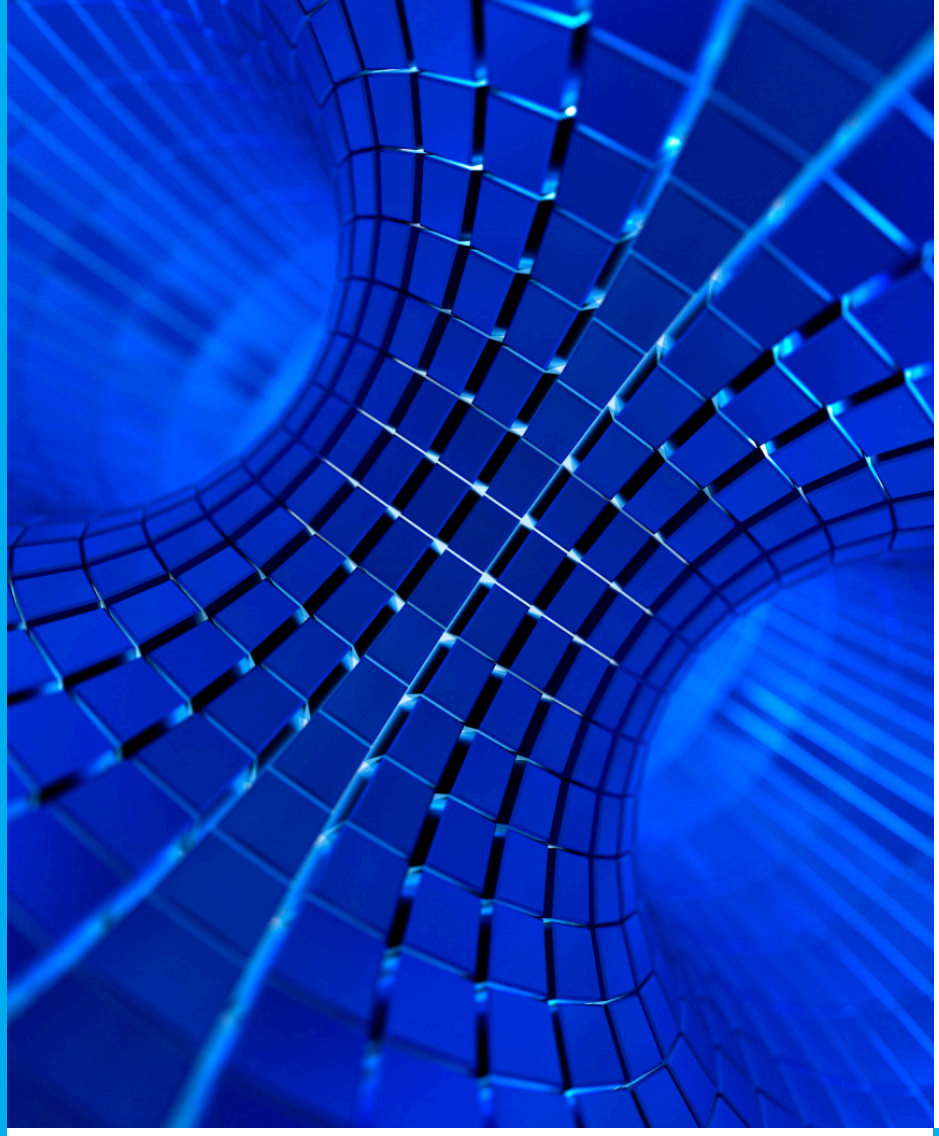


TECHNOLOGY • RESEARCH HIGHLIGHTS • VISIONS • SCIENCE •
12

Model checking methodology for large systems, faults and asynchronous behaviour

SARANA 2011 work report

Jussi Lahtinen | Tuomas Launiainen | Keijo Heljanko | Jonatan Ropponen

VTT TECHNOLOGY 12

Model checking methodology for large systems, faults and asynchronous behaviour

SARANA 2011 work report

Jussi Lahtinen

VTT Technical Research Centre of Finland

Tuomas Launiainen, Keijo Heljanko, Jonatan Ropponen

Aalto University, Department of Information and Computer Science

ISBN 978-951-38-7625-8 (URL: http://www.vtt.fi/publications/index.jsp)
ISSN 2242-122X (URL: http://www.vtt.fi/publications/index.jsp)

JULKAISIJA – UTGIVARE – PUBLISHER

VTT
PL 1000 (Vuorimiehentie 5, Espoo)
02044 VTT
Puh. 020 722 111, faksi 020 722 4374

VTT
PB 1000 (Bergsmansvägen 5, Esbo)
FI-2044 VTT
Tfn +358 20 722 111, telefax +358 20 722 4374

VTT Technical Research Centre of Finland
P.O. Box 1000 (Vuorimiehentie 5, Espoo)
FI-02044 VTT, Finland
Tel. +358 20 722 111, fax + 358 20 722 4374

Model checking methodology for large systems, faults and asynchronous behaviour
SARANA 2011 work report

Jussi Lahtinen, Tuomas Launiainen, Keijo Heljanko & Jonatan Ropponen. Espoo 2012. VTT Technology 12. 84 p.

Abstract

Digital instrumentation and control (I&C) systems are challenging to verify. They enable complicated control functions, and the state spaces of the models easily become too large for comprehensive verification through traditional methods. Model checking is a formal method that can be used for system verification. A number of efficient model checking systems are available that provide analysis tools to determine automatically whether a given state machine model satisfies the desired safety properties.

This report reviews the work performed in the Safety Evaluation and Reliability Analysis of Nuclear Automation (SARANA) project in 2011 regarding model checking. We have developed new, more exact modelling methods that are able to capture the behaviour of a system more realistically. In particular, we have developed more detailed fault models depicting the hardware configuration of a system, and methodology to model function-block-based systems asynchronously. In order to improve the usability of our model checking methods, we have developed an algorithm for model checking large modular systems. The algorithm can be used to verify properties of a model that could otherwise not be verified in a straightforward manner.

Keywords model checking, verification, I&C, NuSMV, UPPAAL, SARANA, SAFIR

Preface

This report has been prepared as part of the research project Safety Evaluation and Reliability Analysis of Nuclear Automation (SARANA), which is part of the Finnish Research Programme on Nuclear Power Plant Safety 2011–2014 (SAFIR2014). This report describes the development of fault modelling methodology, methodology for modelling asynchronous systems using timed automata and a method for analysing large system designs.

We wish to express our gratitude to the representatives of the organizations that provided us with the case examples and all those who have given their valuable input in the meetings and discussions during the project.

Espoo, January 2012

The Authors

Contents

1. Introduction

The verification of digital instrumentation and control (I&C) systems is challenging because programmable logic controllers enable complicated control functions, and the state spaces (number of distinct values of inputs, outputs and internal memory) of the designs easily become too large for comprehensive manual inspection. Design verification is a key task in the design flow because it can eliminate tricky design errors that are hard to detect later in the development process and very expensive to repair, often leading to a major redesign and reimplementation cycle. Typically, verification and validation (V&V) activities rely heavily on subjective evaluation, which only covers a limited part of the possible behaviours of the system, and more rigorous formal methods are therefore required. Such formal methods have been studied (see, for example, [Valkonen 2008] for an overview) but they are not yet widely used.

Model checking [Clarke et al. 1999] is a formal method that can be used to verify the correctness of system designs. Internationally, it has been used in the verification of, e.g., hardware and microprocessor designs, data communications protocols and operating system device drivers. Several model checking systems and tools exist. In our work, we have focused on two model checking tools: NuSMV and UPPAAL. The tools are able to determine automatically whether a given state machine model satisfies given specifications. Model checking can also handle delays and other time-related operations that are crucial in safety I&C systems and challenging to design and verify.

This report reviews the work performed in the Safety Evaluation and Reliability Analysis of Nuclear Automation (SARANA) project in 2011 regarding model checking. We have developed new, more exact modelling methods that are able to capture the behaviour of a system more realistically. In particular, we have developed more detailed fault models depicting the hardware configuration of a system as well as methodology to model function-block-based systems asynchronously. In order to improve the usability of our model checking methods, we have developed an algorithm for model checking large modular systems. The algorithm can be used to verify properties of a model that could otherwise not be verified in a straightforward manner.

The MODSAFE project previously experimented with a technique based on the modular structure of the model, in which the model could be over-approximated by leaving the behaviour of some of the modules out of consideration. Using such a

technique, any composition of the modules can be formed and analysed with little effort. In this work, these modular abstractions are used to create an algorithm that is able to verify automatically a large modular system. This work is reported in Section 3.

We have also developed methodology to model system faults so that the failure behaviour of systems can easily be integrated into traditional models depicting the software logic of a system. The work regarding fault modelling is represented in Section 4.

Finally, we have created asynchronous techniques for modelling function-block-based designs using timed automata. This work is covered in Section 5.

2. Model Checking

Model checking [Clarke et al. 1999], [Clarke & Emerson 1981], [Quielle & Sifakis 1981] is a computer-aided verification method developed to formally verify the correct functioning of a system design model by examining all of its possible behaviours. The models used in model checking are quite similar to those used in simulation as, basically, the model must describe the behaviour of the system design for all sequences of inputs. However, unlike simulation, model checkers examine the behaviour of the system design with all input sequences and compare it with the system specification. In model checking, at least in principle, the analysis can be fully automated with computer-aided tools. The specification is expressed in a suitable language, temporal logics being a prime example, describing the permitted behaviours of a system. Given a model and a specification as inputs, a model checking algorithm determines whether the system has violated its specification. If none of the behaviours of the system violates the given specification, the (model of the) system is correct. Otherwise, the model checker will automatically give a counter-example execution of the system demonstrating how the specification has been violated.

In the SARANA project we have used two model checkers: NuSMV [Cavada et al. 2010], [NuSMV 2011], which was originally designed for hardware model checking, and UPPAAL [Uppaal 2009], which supports model checking of timed automata. NuSMV is a state-of-the-art symbolic model checker that supports synchronous state machine models in which the real-time behaviour must be modelled with discrete time steps using explicit counter variables that are incremented at a common clock frequency. NuSMV supports model checking using both Linear Temporal Logic (LTL) and Computation Tree Logic (CTL) [Clarke et al. 1999], making it quite flexible in expressing design specifications. Several model checking algorithms are employed in this work. The standard algorithm is based on symbolically representing and exploring the state space of the system using Binary Decision Diagrams (BDDs) [Bryant 1986] [McMillan 1993]. SAT-based (Propositional Satisfiability) bounded model checking [Biere et al. 1999] is also supported by NuSMV [Biere et al. 2006] for finding bugs in larger designs. The sophisticated model checking techniques used by NuSMV can handle non-determinism induced by free input variables well, but modelling real-time aspects can be more challeng-

ing due to the inherently discrete time nature of the synchronous state machine model employed by NuSMV.

UPPAAL is a model checking tool for timed systems based on modelling the system as a network of timed automata that communicate through message channels and shared variables. The timed automata have a finite control structure and real-valued clocks [Alur & Dill 1994], making the modelling of timers fairly straightforward. Networks of timed automata can express the real-time behaviour of the system in continuous time and still be automatically analysed. This is feasible because all the possible behaviours of the system can be captured using a finite graph on which different clock valuations with the same behaviour are intuitively grouped into a finite number of equivalence classes called regions [Alur & Dill 1994]. The model checking algorithms use symbolic methods to represent compactly the clock valuations associated with each state of the system quite efficiently in terms of memory. The model checking algorithms employed inside UPPAAL [Alur & Dill 1994], [Larsen et al. 1997] are able to check a subset of the temporal logic TCTL (Timed Computation Tree Logic) [Alur et a. 1990] by explicit state model checking that explicitly traverses the finite graph induced by the behaviour of the system. The main strength of UPPAAL is in analysing the complex timing behaviour of a system. However, it is not well suited to systems with a very high amount of non-determinism as induced by, e.g., reading a large number of input variables (sensor readings) provided by the environment because each combination of inputs is explicitly explored by the employed model checking algorithms.

2.1 Model checking large systems

Model checking has been successfully used to analyse individual functions of safety-critical automation systems. However, it is often necessary to examine several functions simultaneously because the functions may influence the same system parameters. A system may also have several redundant implementations whose behaviour should also be covered. Applying the current model checking methods in a straightforward manner is not always possible in these large and complex systems because the behaviour of the models becomes too rich (i.e. the state explosion problem).

A normal approach to avoid state explosion would be to create an abstraction of the system manually. Based on the verified specification, some system functionality can be irrelevant and left out of the model. Unfortunately, creating such an abstracted model manually for each specification requires a great amount of work. The motivation for our work is: 1) to reduce the amount of work by creating these abstractions automatically, 2) to infer system correctness based on verifications performed automatically on these abstractions, and 3) to reduce the computational effort (avoiding the state explosion of the model).

2.2 Fault models for model checking

Single-fault tolerance has been analysed using model checking in the MODSAFE project in the SAFIR2010 research programme. Our model checking methodology has traditionally included quite non-detailed fault models. Typically, only the status/fault bits of an automation system have been implemented in the model. Extended fault models that allow the model to include physical faults such as faults in telecommunication links, microprocessor faults and electrical faults influencing all equipment in a cabinet were created in SARANA in 2011.

3. Model checking large systems

This work focuses on the analysis of large function-block-based systems. These systems can be modelled as a collection of interacting modules each of which encompasses the functionality of a few function blocks. The methodology is also applicable to other kinds of systems that can be modelled in a similar way.

The analysis of large systems is based on two separate aspects. The first is that the system should be modelled in a way that allows different abstraction-level versions of the model to be created in a practical way. The second part of the verification approach is finding a suitable abstraction level using that framework. The abstraction level should have enough detail to allow verification but not too much, so that the system is still computationally verifiable. In what follows, we first describe how abstractions are created from the models and then an algorithm that can be used to verify large modular systems automatically. The algorithm is implemented for use together with NuSMV model checking software.

3.1 Abstracting the model

Typically, only a small part of the model is needed to verify a specification. The model must be able to be divided easily into such parts, and it should be possible to easily leave some parts of the model outside examination. We have modelled function-block-based systems as a collection of interacting modules (see Figure 1). This section presents the over-approximation already introduced in [Lahtinen et al. 2010] that can be used to create abstractions of the model by replacing some modules with interface modules.

We should be able to select a set of the modules whose functionality we want to analyse. We do not want to set any limitations for the modules that are left out of the analysis. The creation of an abstract model based on a selection of modules should be done automatically and the resulting abstraction should be unambiguous.

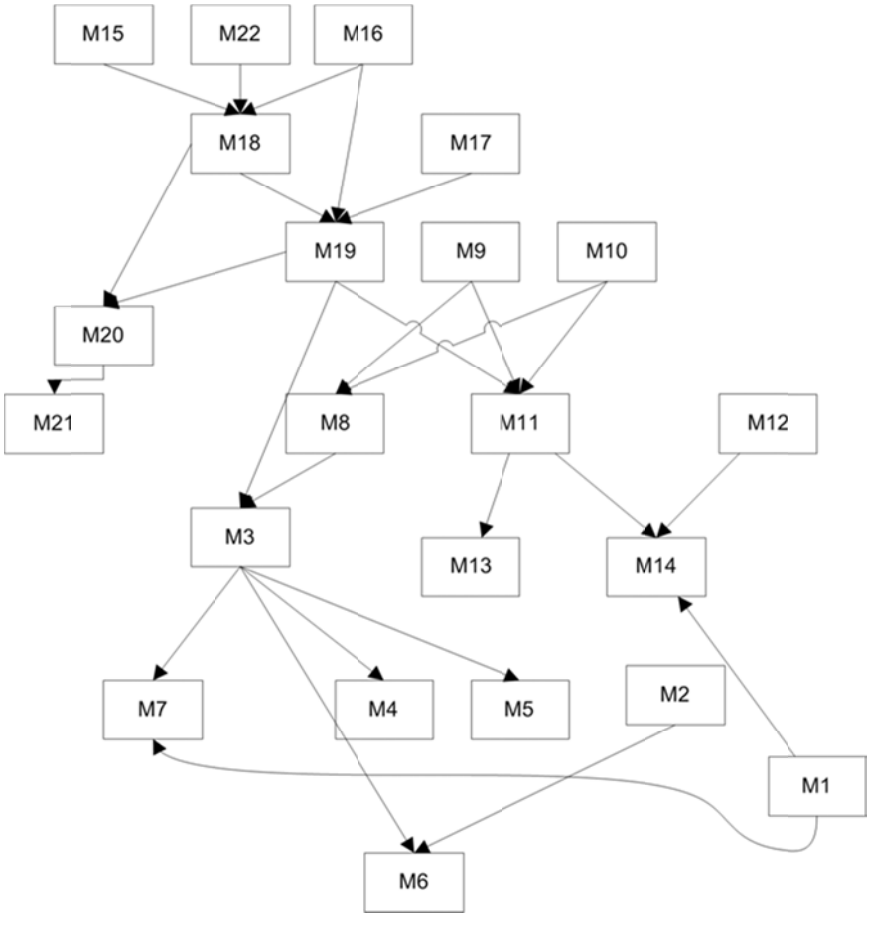


Figure 1. The dependency graph of a system with 22 modules.

The abstraction is implemented so that an abstract version of every module is first created. The abstract version, called the interface module, is such that it has no inner functionality. Only the interfaces of the module are implemented. All outputs of the module are changed into non-deterministic variables. The modules and their interface versions are located in separate files. An abstraction of the system is then created by selecting either a normal version or the interface version of each module and creating a model file based on the selections (see Figure 2). The selection of modules is also referred to as the configuration of the model.

In addition to the different modules, the model has the main module in which the interconnections between the modules are defined. Our implementation of the algorithm also uses a model data file in which all relevant information regarding the model (including dependencies between modules) are listed and can be conveniently read by the algorithm.

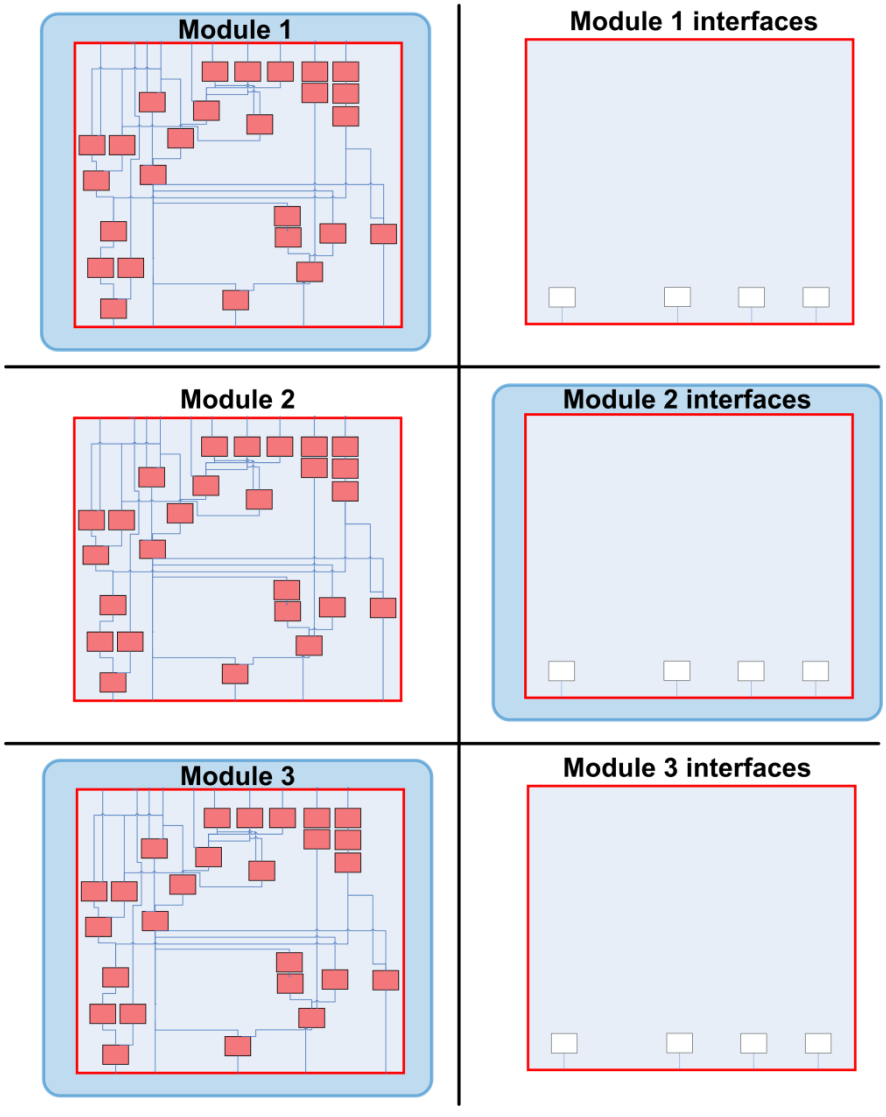


Figure 2. Creating an abstraction of the model by replacing some modules with interface modules

3.2 Property verification using the abstractions

The abstractions discussed above are such that the abstract model always has more behaviour than the non-abstracted model (the interface module is an over-approximation). This is because the interface modules that are used in abstract models are not restricted in any way. The interface modules can output any se-

quence of outputs (unlike the non-abstracted modules). This feature of the abstraction allows verification of safety properties.

A safety property asserts that nothing bad happens. A typical safety property would assert that a defined error state is not reachable in the model. If a safety property is true on some configuration of modules, of which some are interface modules, the property is also true on the original non-abstracted configuration. This is because the interface module abstraction adds behaviours to the model. If a safety property is true in a configuration that has more behaviours than the non-abstracted model, the property is also true in the non-abstracted model.

Our algorithm is designed to verify only safety properties. In particular, the algorithm currently verifies only invariant properties. Invariant properties state that a condition holds for all reachable states. Invariant properties are safety properties but not all safety properties are invariants.

3.3 Automatizing abstraction-level selection

As it is possible to create abstractions of the model by selecting the modules whose functionality we want to analyse, it is possible to find a suitable abstraction level automatically. By suitable, we mean that the abstraction is sufficiently detailed to verify the analysed property, and the abstraction level is coarse enough to be model checked in reasonable time. The whole verification process should also require less time than model checking the non-abstracted model as such.

The technique of finding a suitable abstraction level is largely based on the idea of the Counterexample-Guided Abstraction Refinement technique (CEGAR) by Clarke et al. [Clarke et al. 2004]. The general idea of the CEGAR technique is to model check an abstraction of the system that preserves all behaviours of the concrete system. If the property is true on the abstraction, it is also true in the concrete system. However, a property may be false in the abstraction and still be true in the concrete system (a spurious counterexample is found). In this case, the abstraction is refined based on the counterexample. The refined abstraction is such that it eliminates the spurious behaviour. The process is repeated until the abstract system satisfies the property or a true counterexample is found.

Our algorithm follows the general CEGAR loop, but the adaptation of the process to our modular framework is novel. In addition, we attempt to increase the performance of the algorithm through counterexample minimization. The abstraction refinement step is also different from the ideas in [Clarke et al. 2004].

Our algorithm to model check large modular systems is as follows:

1. Choose the initial configuration of modules based on the invariant property.
2. Model check the current configuration of the modules.
3. If the property is true, return ‘true’. Otherwise, a counterexample is given.
4. Minimize the counterexample.

5. Check the feasibility of the counterexample on the non-abstracted model. If the scenario is feasible, a real error has been found. Return the counterexample.
6. Refine the abstraction based on the counterexample.
7. Go to step 2.

The general intention of the algorithm is to begin with as much abstraction as possible, and then iteratively add modules to the configuration until the abstraction satisfies the property or a counterexample is found. As input the algorithm receives:

- An invariant property
- The main module describing the system as modules
- The modules that consist of a collection of function blocks
- A function block library
- Interface module descriptions for each module
- Dependency information regarding the model: inputs/outputs of each module and the dependencies between modules. It is also possible to extract this information automatically by parsing the main module.

In order to clarify the operation of the algorithm, we present an example run of the algorithm. Suppose that we want to verify the system in Figure 1. The arrows depict dependencies between the modules (outputs that are used as inputs to another module). Each module in Figure 1 consists of a set of function blocks. We want to check if the system satisfies the property that the outputs of modules 19 and 13 (M19 and M13) are never true at the same time (both have only one output 'out1'). We write an invariant:

```
INVARSPEC ! (M19.out1 & M13.out1);
```

Next, we choose the initial configuration of modules based on the invariant property (Step 1 of the algorithm). This is done simply by extracting the variables from the invariant (M19.out1, M13.out1) and determining the modules that have these variables as outputs (Modules M19 and M13). These modules are selected as the initial configuration. Other modules of the system are replaced by their respected interface modules. The initial configuration is illustrated in Figure 3.

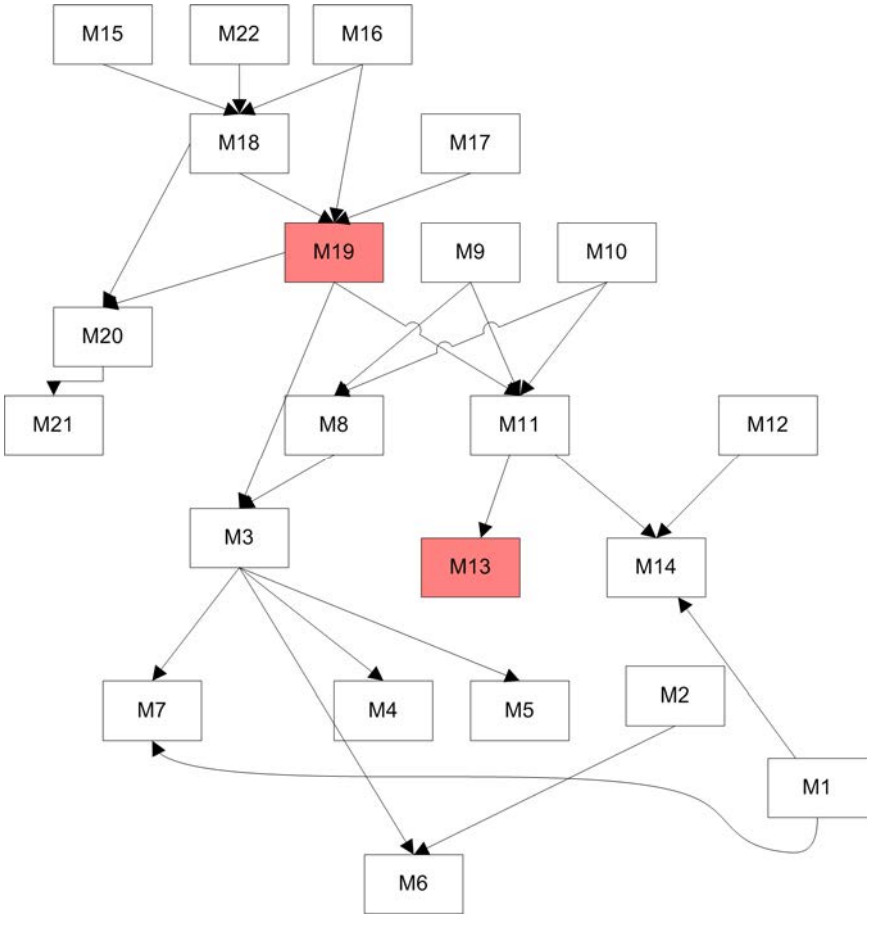


Figure 3. The initial configuration of the example run of the algorithm.

Next, based on the selection of modules, we generate the model file by appending the main module with appropriate module files and apply model checking to this file. We use the invariant checking algorithm of the NuSMV tool together with the cone of influence reduction (described later).

In our running example, NuSMV outputs a counterexample in which the outputs of M13 and M19 are both true at the same time. Next we attempt to minimize the counterexample. Based on the dependency graph we can see that our current abstraction (M13 and M19) is dependent on modules M11, M16, M17 and M18. Since the modules are deterministic, we know that the outputs of M11, M16, M17 and M18 cause the error in the counterexample. In the counterexample minimization step we check whether there is some subset of these outputs in the counterexample that would be the cause of the error. Counterexample minimization is discussed in the next section in more detail. In our example, let us assume that we

deduce that the outputs of module M18 are irrelevant with respect to the counterexample. We remove the variable assignments referring to module M18 from the counterexample.

The fifth step of the algorithm is to check whether the minimized counterexample is feasible in the non-abstracted model. We generate a configuration of the model in which all modules are non-abstracted and add clauses that force the model towards behaviour described by the counterexample. Then we check whether the last state of the counterexample can be reached in the model. If it can be reached, the counterexample is feasible in the non-abstracted model and depicts a true error in the system. If the last state cannot be reached, the counterexample is spurious, and the abstraction level should be refined.

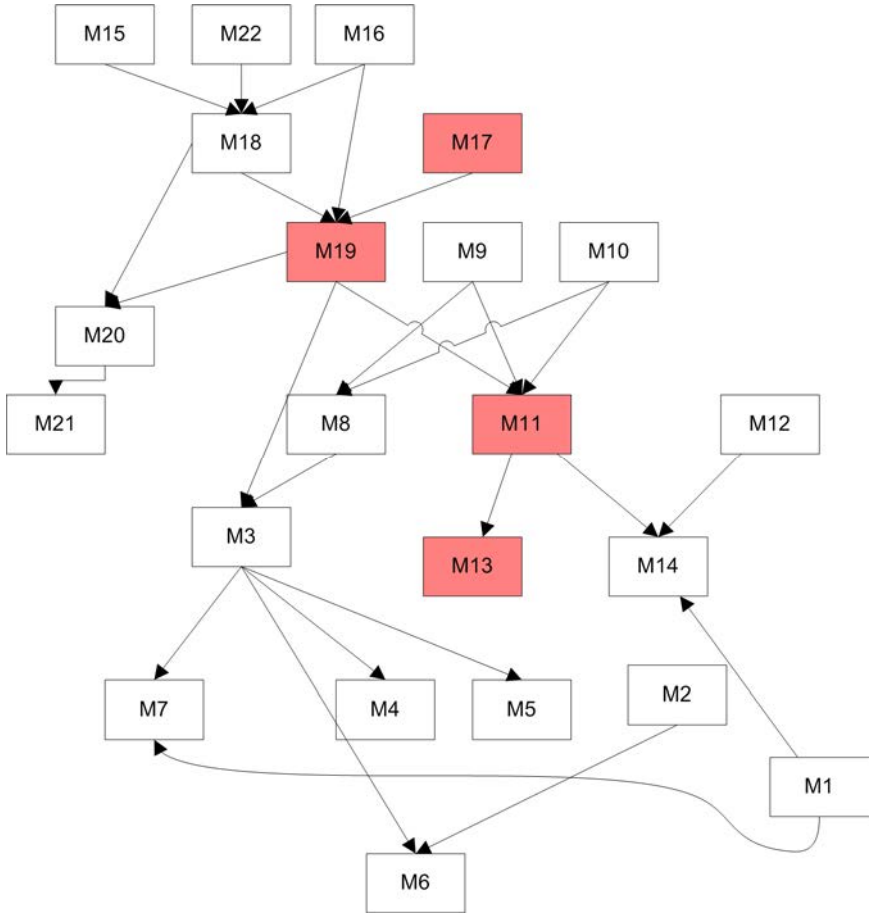


Figure 4. The refined abstraction level after one iteration.

Suppose that in our running example, the counterexample is not feasible in the non-abstracted model. We have to refine the abstraction in such a way that it eliminates the spurious counterexample (but still adds as few modules as possible). The refinement step starts by looking at the dependency graph. Earlier we concluded that the error is caused by modules M11, M16 and M17 (M18 was deduced to be irrelevant). We first check whether adding these three modules eliminates the counterexample. We generate the module configuration (M11, M13, M16, M17, M19) and perform another feasibility check. In our example, the configuration is not feasible, which is exactly what we were looking for (the spurious counterexample is eliminated). We can still try to improve the refinement by checking if adding some subset of the new modules M11, M16 and M17 also eliminates the counterexample. We find out that the smallest such subset is (M11, M17). The resulting refined abstraction is then (M11, M13, M17, M19); see Figure 4. The abstraction refinement is also discussed in more detail later.

Finally, the refined abstraction is model checked again. In our example, the result is true, which implies that the non-abstracted model satisfies the specification. The algorithm finishes and returns the value ‘true’.

In what follows, some steps of the algorithm (invariant model checking, counterexample minimization, feasibility checking of the counterexample, and abstraction refinement) are discussed in more detail.

3.4 Invariant model checking

The second step of the algorithm is model checking the current configuration of modules. In this step, we use two model checking algorithms in parallel: BDD-based invariant checking and bounded model checking (BMC) -based invariant checking. The BDD-based model checking algorithm may require a large amount of time when the size of the model increases. This is why the bounded model checking is run in parallel. The BMC algorithm can find counterexamples faster (if they exist). This can reduce the run-time of the algorithm because the BDD algorithm can be interrupted when a BMC counterexample is found. On the other hand, we need both algorithms because the used BMC algorithm may not be able to prove a property within a reasonable bound.

3.5 Counterexample minimization

If model checking an abstract configuration leads to a counterexample, the feasibility of the counterexample should be checked on the non-abstracted full model. The counterexample is minimized before the feasibility check.

The abstract counterexample consists of a set of variable assignments on different time steps. The counterexample can be minimized so that only the relevant variable assignments remain that actually cause the counterexample. The minimized counterexample is such that the variable assignments in it always lead to

the original error despite the variable assignments that exist outside the minimized counterexample.

Counterexample minimization is performed for two reasons:

- Free non-deterministic variables of the abstract model can be restricted in the full non-abstracted model in a way that makes the counterexample infeasible. However, these variables may not be causing the error described by the counterexample (values are irrelevant). Leaving these variables outside the counterexample reduces false negative feasibility answers.
- The abstract model includes the behaviour of a set of modules A. These modules are dependent on signals received from a set of modules B that are not in the abstract model. In the abstraction refinement step of the algorithm, the set A is increased by adding new modules from the set B based on the dependency relation. If it can be demonstrated that a module M in B is irrelevant with respect to the examined counterexample, that module M can be left out in abstraction refinement, and thus the size of the abstract model after refinement is decreased. The size of the model has major influence on the performance of the algorithm. It may be possible to simplify the dependency relation of the modules based on counterexample minimization (if all signals from one module M in B are minimized).

In what follows, we describe techniques for counterexample minimization that can be used together with the algorithm. The techniques based on random walk and delta debugging are search methods that do not produce the smallest possible counterexample. The CTL query-based technique does produce the smallest possible counterexample but the technique requires more computation.

Some issues are related to the minimization regardless of the minimization technique. The counterexample minimization techniques require:

- The model file
- The model checker (NuSMV)
- The abstraction level of the model (which modules are abstracted as interface modules)
- Module dependency information (A list of modules, their inputs and the modules from which these inputs are received)
- The counterexample file in XML format
- The examined invariant property (only invariant properties and their counterexamples are supported).

The counterexample given as input can be simplified by itself. Typically, the model checker can perform some form of trace simplification. NuSMV allows the use of the cone-of-influence (COI) reduction, which can be used to reduce the number of variables in the counterexample before further simplification is applied. The COI technique is introduced later on.

The counterexample can also easily be simplified by only taking into account the input variables of each module. The model behaviour is fully determined by exactly these variables. In the modelling approach used, all the other variables (output variables, memories of the function blocks) receive their values from the input variable sequences in a deterministic way. Thus, the counterexample can also be described in terms of the input variables only. A list of 'free variables' can be extracted from the module dependencies and the abstraction levels of the modules. Free variables determine the behaviour of the system in an unambiguous way. All **free variables** of an abstracted model are either:

- **Interface variables**: input variables of the non-abstracted modules that receive their value from an interface module, or
- **Non-deterministic variables**: input variables whose value is determined non-deterministically in the model (they can have any value at all times).

Thus, an input variable whose value is received from a non-abstracted module M (a module that is included in the abstraction) is not a free variable, since its value is determined by the input variables of module M. This leads to two realizations:

- Only variable assignments for the free variables are preserved in the counterexample. Other variable assignments are redundant and can be left out.
- The non-deterministic variables of an abstract model are also non-deterministic in the full non-abstracted model. The non-deterministic variables cannot cause false feasibility answers in the algorithm nor can their removal lead to reductions in the abstraction refinement step. Therefore, the non-deterministic variables should not be the target of minimization. The non-deterministic variables are always included in the minimized counterexample.

Thus, our counterexample minimization techniques focus on further minimizing the variable assignments of the counterexample (already minimized using the cone-of-influence reduction of NuSMV) that are the interface variables of the current abstraction.

The minimization techniques are based on the idea of creating modified copies of the model in which some variables of the model are forced to follow the behaviour described by the counterexample.

The modified copy is created by adding a clock variable and clauses for the forced variables. The clock variable *clock* is simply an integer variable that is added to the model. The clock variable has the values from 0 to the length of the counterexample. The clock variable's initial value is 0. After that, the value is increased by one at each time step. When the length of the counterexample is reached, the value of the clock is permanently set to the highest value. For example, if the counterexample consists of two time points, the NuSMV code for the clock variable would be:

```
init(clock):= 0;
next(clock) := case
clock < 2 : clock +1 ;
TRUE : 2 ;
esac;
```

For each forced variable, the *init* and *next* clauses of NuSMV are created so that the variable follows the counterexample values until the end of the counterexample. For example, a variable that has the value FALSE at time point 0 and TRUE at time points 1 and 2 in the counterexample would translate into the clauses:

```
init( variable1 ) :- FALSE;
next( variable1 ) := case
         clock = 0  : TRUE ;
         clock = 1  : TRUE ;
         TRUE :  {TRUE, FALSE} ;
esac;
TRUE : {TRUE, FALSE} ;
esac;
```

Using the modified copies of the model, it can then be verified by model checking whether the copies are such that the error always manifests itself. If it does, the set of forced variables can be used to create a new minimized counterexample. Other variables are irrelevant. In order to find the smallest possible counterexample, the naive approach would be to create a modified model for every subset of variable assignments in the counterexample and check each one separately. The smallest subset leading to a true minimized counterexample would then be the smallest possible counterexample. However, checking all subsets of the variable assignments is too laborious, and some heuristics are needed. We use, for example, local search techniques, and apply the techniques on several granularity levels.

Each minimization technique can be applied on at least three different granularity levels:

- **Module level minimization**: Since one of the objectives of the minimization is to break dependencies between modules, it makes sense to find out whether some module as a whole is irrelevant with respect to the counterexample. On the module level, the variable assignments are grouped according to the module that outputs them. The subsets of these groups are then examined to find the smallest subset, such that any execution still always leads to the error in the counterexample. The variable assignments related to irrelevant modules are then removed from the counterexample. For example, if the module M_1 is removed as a result, assignments of variables output by M_1 are removed from the counterexample at all time points.

- **Variable level minimization**: At the variable level, we look for the smallest subset of variables whose variable assignments in the counterexample always lead to the error. For example, if variable *var1* is removed by minimization, variable assignments of *var1* are removed from the counterexample at all time points.
- **Variable/time point level minimization**: At this level, each *(variable, time-point)* pair of the counterexample can be minimized. We look for the smallest subset of assignments that always leads to the error. For example, the variable assignments of *var1* could be removed at time point 0 but assignments at other time points could still be relevant.

The granularity levels are independent of each other. It is possible to apply the minimization only on some granularity level or first use the module level to quickly reduce the size of the counterexample and then further refine the minimization using minimization on the variable/time point level.

3.5.1 Random walk-based minimization

Random walk-based minimization can be applied on all granularity levels. Below, we describe the functioning on the variable level only. Other granularity levels are applied similarly.

The idea of random walk-based minimization is to start with a modified model in which all variables that are targets of the minimization are forced to the values in the counterexample. Then, we follow the algorithm:

1. All variables that are to be minimized are in the set *Forced;* the set *Result* is empty.
2. If *Forced=Result,* return *Result*.
3. Select some random variable *V* from the set *Forced* such that it is not in *Result.*
4. Remove *V* from *Forced.*
5. Create a new modified model *M* in which the variables in *Forced* are forced to the values in the counterexample
6. Check whether *M* is such that the error described by the counterexample exists no matter what the values of the variables not in *Forced*. This can be done using the bounded model checking algorithm in NuSMV. If so, go to step 2.
7. Put *V* back into the set *Forced,* add *V* to *Result* and go to step 2.

The algorithm returns a set of variables that always leads to the error, and the set is a local minimum. In order to decrease the size of the resulting minimized counterexample, the random walk can be run a few times, and the smallest counterexample is then selected.

3.5.2 Minimization using delta debugging techniques

The minimization based on delta debugging techniques can be applied on all granularity levels. Below, we describe the functioning on the variable level only. Other granularity levels are applied similarly.

The reasoning in this approach is somewhat similar to the random walk-based approach, but the heuristics of selecting candidates for a smaller counterexample differ. In delta debugging, the minimal set of forced variables that always leads to a counter-example is looked for by dividing the forced variables into subsets and checking the subsets and their complements one by one. The search initially divides the variables into two sets and then increases the granularity if none of the subsets explains the counterexample. We follow the algorithm below:

1. Set granularity *n*=2. The set of variables that are to be minimized is *S*.
2. Divide *S* into *n* subsets (*Subsets*).
3. Calculate the complements of the *Subsets* (*Complements*).
4. For *x* in *Subsets:*
   4.1. Create a modified model M in which the variables in *x* are forced to the values of the counterexample.
   4.2. Check if *M* is such that the error described by the counterexample exists no matter what the values of the variables not in *x*. This can be done using the bounded model checking algorithm in NuSMV. If so, set n=2 and *S*=*x*, and go to step 2.
5. For *x* in *Complements:*
   5.1. Create a modified model M in which the variables in *x* are forced to the values of the counterexample.
   5.2. Check if *M* is such that the error described by the counterexample exists no matter what the values of variables not in *x*. This can be done using the bounded model checking algorithm in NuSMV. If so, set n=2 and *S*=*x*, and go to step 2.
6. If granularity n < |S|, set *n*=min(|*S*|, 2**n*) and go to step 2.
7. Else (Granularity is greater than or equal to |S|) return *S*.

3.5.3 CTL query-based minimization technique

The minimization based on CTL queries can be applied on all granularity levels. Below, we describe the functioning on the variable level only. Other granularity levels are applied similarly.

In the minimization by CTL queries, the idea is again to create modified copies of the abstract model in which variables are forced to values in the counterexample. The difference from the previous minimization techniques is that the forced variables are not selected one by one by the algorithm. Instead, a new Boolean variable is introduced for each variable under minimization. The Boolean variable called the **lock variable** determines whether the variable it locks follows the value

of the counterexample. The modified model is then model checked against a special CTL specification in order to determine the minimum number of TRUE assignments in these lock variables that are required to force the system to violate the original property. The ideas are explained in detail in what follows.

Variables under minimization are forced to the counterexample values through lock variables and a clock variable keeping track of time. The clock variable *clock* is simply an integer variable that is added to the model (similarly to that in random walk minimization).

A new variable, *lockX* (that has values 0 and 1), is added for each interface variable (variables that are the target of the minimization). A mapping is created in which the correspondences between the lock variables and the interface variables are determined. The lock variables are such that they non-deterministically choose a value (0/1) at the initial time point and retain the value at all future time points. This is done by omitting the NuSMV init statement and using next statements such as:

```
next(lockX):= lockX;
```

The value 1 of a lock variable means that the variable related to it has the same value as the counterexample at time points less than or equal to the length of the minimized counterexample. If *lockX* has value 0, the value of the variable related to that lock variable is not restricted in any way.

The behaviour of the lock variables is realized through init and next statements written for all variables in the counterexample:

```
init( variable1 ) case
                      lock1 = 1 : FALSE ;
                      TRUE : {TRUE, FALSE} ;
esac;
next( variable1 ) := case
        clock = 0  : case
           lock1 = 1 :  TRUE ;
           TRUE :  {TRUE, FALSE} ;
        esac;
        TRUE : {TRUE, FALSE} ;
esac;
```

In the above example, according to the original counterexample, *variable1* takes the value FALSE at time point 0 and the value TRUE at time point 1. If *lock1* has the value TRUE, *variable1* takes the value FALSE at time point 1 and the value TRUE at time point 2. At all other time points the value of *variable1* is non-restricted. Enumerative variables are also supported. The variable domain is read in the beginning of the counterexample minimization from the module dependency information. The full domain range then replaces *{TRUE, FALSE}* in all instances.

The number of variables that is locked is controlled through another variable:

```
nro_of_locked_variables :=  lock0 +  lock1 +  ... lockN;
```

Finally, the examined property is a CTL property:

```
CTLSPEC ! (nrolockedvars = x & AF(clock = y & error));
```

where:

- x is a variable that determines the number of locked variables
- y+1 is the length of the original counterexample
- error is the negation of the original invariant property.

The CTL formula states that no such initial state exists in which a certain number *x* of the interface variables are locked to the values of the original counterexample so that no matter what the values of the other non-locked interface variables are, the system will eventually lead to the error state manifested in the original counterexample. However, if the formula leads to a counterexample, it means that such a choice of locked variables exists. The actual locked variables can be deduced from the counterexample trace. If the formula is true, a higher value of *x* should be tried out when looking for a minimal counterexample. (Note that the 'function' here is monotonous. If the formula is true for x=5, then the formula is also true for x < 5. If the formula is true for some value of x, then if a smaller value of x were to exist that resulted in a counterexample of the formula, then the same counterexample could be produced by choosing these variables and a number of other variables. If the formula is false for x=10 then a counterexample also exists for all x > 10. If a set of variables and time points exist that are sufficient to produce the counterexample, then adding other variable assignments cannot change this.)

Now, the resulting modified model can be used to check if it is possible to lock a certain number of interface variables in such a way that the original error manifests itself no matter what the other non-locked variable assignments are. The number of interface variables is known, and the minimum number of locks required can be found through binary search:

```
binarySearch(low, high):
x = (high + low) / 2
model check modified model using specification:
'CTLSPEC ! (nrolockedvars = x & AF(clock = y & error));'
if (specification is false):
          return min(minimization, binarySearch(low, x-
        1))
else:
         return binarySearch(x+1, high)
```

The counterexample minimization described here can also be used on the module level and the variable/time point level. On the module level, a lock variable is cre-

ated for each module and these locks affect the interface variables related to that module. On the variable/time point level, a separate lock variable is introduced for each variable-time point pair, e.g. if the counterexample has five time points, five lock variables are created for each interface variable. This way, the counterexample could be further minimized because in some cases only a value at some specific time point is relevant. The value of some variable at other time points does not affect the realization of the error. In large models, however, this more detailed minimization approach can be too complex. The approach can lead to too many lock variables and a state explosion.

The CTL-based minimization approach cannot use the bounded model checking algorithm available in NuSMV. Its result is optimal, but in practice the required CTL checks are too complex. The minimization through delta debugging is faster but may not return the minimal counterexample. In our algorithm, we prefer quick minimization over optimal minimization.

3.5.4 Related work

Counterexample (or bug trace) minimization has been a research topic in both hardware and software verification. Novel verification tools are effective, but a large amount of manual effort is required to analyse the results of these tools. In simulation and model checking (and other formal methods), the analysis of bug traces or counter-examples is time-consuming and laborious. The counterexamples of model checking, for example, can consist of hundreds of different variables and multiple time points. However, only a fraction of the variable assignments are usually important, and most variable values have no influence on the realization of the counterexample.

Techniques have already been developed for counterexample minimization. Some of these techniques have been implemented in the model checking tools themselves. In what follows, some research on counterexample minimization is reviewed.

3.5.4.1 Program slicing and the cone of influence reduction

Program slicing [Weiser 1981] is an abstraction of a program or a specification with respect to a given condition called the slicing criterion. The slicing criterion is such that it holds on the full program if and only if it holds on the reduced program. The ideas of the program slicing techniques have been used in model checking where a temporal logic formula is interpreted as the slicing criterion. In the program analysis of model checking models the temporal logic formula must hold on the reduced model if and only if it holds on the full model. The technique is also known as the cone of influence reduction used in hardware verification.

Cone of influence (COI) [Clarke et al. 1999] is an abstraction technique that decreases the state space of the model by focusing only on the variables of the model that are relevant to the examined specification. The reduction is obtained by

removing variables that cannot influence the variables of the specification. The basic idea is to create a dependency graph of the variables in the model and then traverse the graph starting from the variables of the specification. Since the cone of influence reduction reduces the number of variables in a model, it also reduces the number of variable assignments in the counterexample.

The cone of influence reduction can be improved by taking into account the different time points at which a variable can have influence on the specification. This technique is referred to as Bounded Cone of Influence (BCOI) [Biere et al. 1999].

3.5.4.2 Brute Force Lifting

The technique called Brute Force Lifting (BFL) [Ravi & Somenzi 2004] was introduced in the context of minimizing bounded model checking (BMC) counterexamples. BMC counterexamples are satisfying assignments to a Boolean formula, typically in conjunctive normal form (CNF). The idea is to derive a minimally satisfying counterexample that, together with the Boolean formula describing the model, implies a violation of the checked property.

The examined technique is performed on the level of Boolean formulas solved by a SAT solver. On that level, the paper describes a process of simplification called lifting. Lifting is the process of removing literals or variables from a satisfying assignment such that for each valuation of the lifted variables the formula is still satisfiable. Some variables are clearly irrelevant with respect to the checked property and can be lifted. The relevance of other variables has to be checked by brute force. This means that for each checked literal, the *negation* of the property is checked with a SAT solver. If the result is satisfiable, then the literal cannot be lifted. The BFL technique described in the paper is performed on the inputs of a system. Since a SAT solver run is needed for a single lifting, the technique can be quite expensive computationally. However, the experimental results showed that the average reduction in counterexample variables was 71%.

The BFL technique can be further improved by the elimination of sets of variables simultaneously [Shen et al. 2005]. The technique is based on refutation analysis and incremental SAT. The idea is that after checking the negation of the property with a SAT solver and receiving an UNSAT result, the result is analysed to find out if it implies that other free variables are also irrelevant. If the result of the check is UNSAT, then there must be a conflict clause at decision level 0. The conflict clause is then used to traverse the implication graph in the reverse direction to obtain the set of clauses that leads to that conflict. The irrelevant variables are then the variables that are in that conflict-causing set and whose negation is also in that set. These variables are the reason the problem is UNSAT. Thus, they are the reason the counterexample must always happen. These are also the irrelevant free variables that can be lifted at the same time. The idea is equal to the finding of an unsatisfiable core of the formula and the free variables and their negations that are in the core.

The general idea of checking the satisfiability of the negation of the property is also used in our work. In our work, the check is performed with the model checker, while a set of variables is set free. The variables in our work are the variables of the model, not the low abstraction level variables of the Boolean formula given to the SAT solver. The general idea, however, is similar.

3.5.4.3 Simulation-based bug trace minimization

The minimization of simulation bug traces is examined in [Chang et al. 2007]. In simulation, the system is run against a set of assertions. For example, random simulation can be run on some design, while assertions are monitored. The application of simulation late in the life cycle of the product results in detailed and long traces. The technique and the tool examined in the paper analyse a simulation bug trace and produce an equivalent trace of shorter length. The technique relies on both simulation and formal methods.

The techniques described in [Chang et al. 2007] are two-fold. Some techniques intend to remove redundant time steps from the bug trace. Another group of techniques intends to simplify the trace by identifying essential input values.

Proposed shortening techniques by the paper:

1. Single-cycle elimination: remove cycles completely and re-simulate to see if the bug still exists.
2. Alternative path to bug: simulate with alternative transitions during the trace and detect if a shorter path violating the assertion is found.
3. State-skip: identify non-unique states that represent loops. If the same state is in the trace twice there is a loop.
4. BMC-based refinement: search locally for shorter paths between two trace states.

Proposed simplification techniques:

1. Input event elimination: re-simulate with fewer input events. For example, set c=0 instead of c=1. If the bug still manifests itself, the input event is redundant.
2. Essential variable identification: use three-value simulation to identify non-essential inputs.

In most cases, traces can be reduced to a fraction of their initial size. The average reduction in a trace produced by random simulation was 99% in terms of cycles and input events. For traces that were produced by a semi-formal method, the techniques are also effective (reduced traces ~75-90%).

Our work focuses on simplifying but not shortening the counterexample in time. We also use BMC to produce the shortest possible counterexample.

3.5.4.4 Minimizing automata-based model checking counterexamples

Gastin et al. [Gastin et al. 2004] minimize automata-based model checking counterexamples. Their objective is to find minimal counterexamples in terms of time steps in the counterexample. If the model is represented as a Kripke structure, checking LTL properties is equivalent to testing whether the intersection of the model and a Büchi automaton describing violating executions has no accepting run. The traditional algorithms look for accepting runs with a depth-first algorithm that returns the first accepting run found. The algorithm described in [Gastin et al. 2004] performs a depth-first search to find a minimal bug trace. The idea is that the search does not necessarily stop when a state already visited is reached. Reaching a state with a distance to the initial state smaller than for the previous visit may lead to a shorter counterexample. Therefore, for each state, there is an additional field, storing the smallest length on which that state occurred. The minimal counterexample found so far is also stored.

3.5.4.5 Explaining counter-examples through forced and free segments

The paper [Jin et al. 2002] distinguishes between ‘control’ and ‘data’ signals in the counterexample. The paper discusses the explanation of counterexamples rather than minimization. The explanation is performed through the annotation of the error traces by alternation of fated (forced) and free segments. The fated segments show unavoidable progress towards the error while free segments represent avoidable choices that have led to the error. The annotation helps in the error interpretation. The fated segments are control signal values that lead towards the error. The free segments represent mistakes made in the choice of data values that also lead towards the error.

The paper also interprets counterexample minimization as a two-player concurrent reachability game. The two players are the (hostile) environment and the system. The environment chooses values for the controlling variables and the system simultaneously chooses the values for the rest of the variables (data variables). The environment’s goal is to reach the error state of the counterexample. A (memoryless) strategy for the environment is a function that maps each state to one valuation of the control variables. Likewise, a strategy for the system is a function that maps each state to one valuation of the data variables. A position is a winning position for the environment if there is an environment strategy such that, for all system strategies, the error state is eventually reached. A position is a winning strategy for the system if the error state is never reached.

3.5.4.6 Symbolic Trajectory Evaluation

The ideas of counterexample minimization are also somewhat similar to the techniques used in the abstraction refinement of symbolic trajectory evaluation [Roorda & Claessen 2006].

Symbolic Trajectory Evaluation (STE) [Seger & Bryant 1995] is a formal verification technique that combines three-valued simulation with symbolic simulation. STE is used to verify assertions of the form A → C, where A is called the antecedent and C is called the consequent. The expression A specifies the values used in the simulation, while the expression C depicts the expected result. STE is often used to verify digital circuits, e.g., the technique is extensively used at Intel.

In three-valued simulation, a third value is introduced to the (Boolean) simulator. The third value X represents an unknown value. A state with some variables set to X covers those states obtained by replacing the X values with all combinations of 0 and 1. When three-valued simulation is used, the transition relation of the model is extended to cover also the value X. With three-valued simulation it is possible to verify the STE assertions using fewer simulation runs, since one simulation run corresponds to several of the original Boolean simulations.

In symbolic simulation, Boolean expressions over symbolic variables are used to verify system properties. A Boolean expression over symbolic variables can be written for the model and the consequent of the STE assertion. The expressions should then be compared for equality. One way of doing this is to use the BDD data structure. A BDD is calculated for each input of the model, and for each gate a BDD is calculated that represents the output of the gate. Finally, a BDD is calculated for the whole circuit. Since BDD is a canonical data structure, the comparison with the BDD of the consequent is simple. The disadvantage of symbolic simulation is that the number of symbolic variables needed can be huge, which leads to the BDD blow-up.

The two techniques work well together since three-valued simulation decreases the number of symbolic variables that are needed.

The STE abstraction is typically initially not proven because the antecedent yields X values for nodes that are required to have some particular Boolean value by the consequent. When this happens the abstraction must be refined. The abstraction refinement issue is discussed in [Roorda & Claessen 2006]. Roorda et al. have invented the concept of strengthening, which indicates the input of a circuit that needs to be given a non-X value in order to take non-X values at the relevant outputs. The writers have created a tool that can calculate strengthenings that correspond to counter-examples of the assertion. In this sense, calculating the weakest satisfying strengthening has similarities with counterexample minimization. The weakest satisfying strengthening of a counterexample indicates the variables of the model that have to have some particular Boolean value so that the counterexample manifests itself. The number of such variables is also minimal.

In [Roorda & Claessen 2006], SAT-formulas are generated whose solutions represent the satisfying strengthenings of the assertion. An incremental SAT-solver is used iteratively to find the weakest strengthening. This is done using constraints to block the last found strengthening and allowing only strictly weaker strengthenings.

There are many similarities to ideas used in our minimization method. We also look for the minimal number of variable assignments that are needed to produce the counterexample. Instead of using three-valued simulation, we modify the mod-

el and use a distinct specification that is checked by the model checker. We also find the minimal number of variables iteratively. The difference between our technique and the one in [Roorda & Claessen 2006] is that their three-valued abstraction has some inherent information loss. This means that the technique based on three-valued simulation may come up with a non-minimal result. In other words, some variables of the counterexample may not be necessary to produce it, but the three-valued abstraction requires that they are not removed. Information loss can also occur due to the fact that the STE method performs only forward simulation. If the antecedent of the assertion specifies some output value but not the inputs relevant to it, the inputs are assigned value X, which can cause the assertion to fail. However, the information loss caused by three-valued abstraction and forwards simulation can be avoided by adding extra symbolic variables.

3.5.4.7 Localizing errors in counterexample traces

The paper [Ball et al. 2003] discusses finding the cause of errors in a counterexample trace by comparing the trace against correct traces. They also demonstrate how multiple error traces with independent causes can be generated. The algorithms are implemented in the context of the software model checking tool SLAM.

The counterexample is seen as a symptom of the error. The cause of the error is extracted by comparing these erroneous traces against correct traces and looking for transitions of the error trace that are not in any correct trace of the program. Program statements inducing these transitions are likely to contain the causes of the error. Other possible causes of the same error can be looked for by replacing the detected erroneous transitions with halt statements and re-running the model checker until no more error traces can be found. Thus, a single error trace can be outputted for each possible cause of the error. The approach is problematic in detecting the cause of errors in some cases: all transitions of the counterexample also exist in some correct trace, in which case the cause of the error is empty (coincidental correctness). In general, the algorithm managed to identify the cause of an error directly in 11 out of 15 error traces. In three cases, the cause could be deduced by tweaking the algorithm. In one case, the abstraction level of the model inhibited finding the cause of the error. In many cases, the error causes found were only a small fraction of the error trace. (All were less than 16% of the transitions in the error trace, typically about 1%.)

3.5.4.8 Error cause extraction through variations of the error

Other traces are also used in [Groce & Visser 2003] to extract the error cause. The paper describes how an automated method can be used to find other versions of the error and a set of correct traces and to analyse the executions to extract the cause of the error.

The work focuses on finite executions demonstrating violation of safety properties in Java programs. The algorithms are implemented in the Java Pathfinder

model checker. The paper defines a set of executions called negatives as variations of the counterexample trace that produces the same error. A second set called positives is defined as a set of traces that are variations of the original error trace in which the error does not occur. Negatives are executions that reach the error state from the same control location; not all possible ways to reach the error state are accepted. Similarly, positives are executions that pass through that control location without proceeding to the error state. The method of generating the negatives and positives uses a model checker to explore backwards from the original counterexample.

The paper introduces three analysis methods that can be performed on the negatives and the positives to extract the cause of the error:

1. Analysis of the transitions (similar to the method described in [Ball et al. 2003]): computes sets of projected transitions (pairs of control locations and actions). After this, the transitions that appear in all positive/negative traces are reported. The transitions that only appear in negative/positive traces are also reported. It is also indicated whether these transitions are such that they appear in all negative/positive traces (causal transitions that denote precisely the common behaviour that differentiates the negative and positive sets).
2. Analysis of data invariants over the executions: the same control locations may be present in both negative and positive traces. It may be that the control location does not induce the error, but the choice of data values does. In this analysis, data invariants are calculated over the negatives and these invariants are compared with the invariants of the positive traces. The invariants are calculated using Daikon [Ernst et al. 2007].
3. Analysing the minimal transformations between negatives and positives: here the least number of changes required to make a positive into a negative if looked for.

In experimental tests, the algorithms found 131 variations on one found error. The analysis implied a function call that was present in all negatives, but also in some positives, and a few short transformations indicating that the function call has to be made in certain conditions related to time.

3.5.4.9 Delta debugging

Counterexample minimization has similarities to the test case simplification of the delta debugging method [Zeller 2002], [Zeller & Hildebrandt 2002]. In delta debugging, a test case that produces a failure is simplified to a minimal test case that still produces the failure. Every part of the resulting minimal test case is significant in reproducing the failure. The delta debugging algorithm works by successively running test cases that contain only a subset of inputs of the original test case. It also runs test cases in which the complement of the set of inputs is always chosen. If some of the input sets can produce the failure they are chosen as the new

failure inducing test case in the algorithm. If none of the subsets causes the failure, the granularity of the subsets is increased until a failure-inducing subset (or its complement) is found. For example, the algorithm starts by dividing the test case into two halves. If these input sets do not produce the failure, the test case is split into three mutually exclusive subsets. The complements of these subsets are also checked. The algorithm stops when removing any single input causes the failure to disappear.

The delta debugging algorithm is also used in our work to generate sets of variables that are used to create models that are then model checked. In a way, we have adapted the delta debugging method (originally used with test cases) to model checking. We also use delta debugging style minimization in the abstraction phase of our algorithm.

3.6 Checking the feasibility of the counterexample

The idea of feasibility checking is to find out whether it is possible to obtain the same error that was discovered in the abstracted model using the non-abstracted version of the model. If a trace of the full model can be produced that includes all relevant free variable assignments (the minimized counterexample) then the counterexample is a true counterexample and describes a true error in the model/system. If the trace is not feasible, this is because some modules' functionalities prohibit the variables from obtaining the values of the counterexample. Non-feasibility means that the specification has to be checked on a more refined abstraction of the model.

In order to check counterexample feasibility the following inputs are required:

- The full non-abstracted model.
- A counterexample discovered in the abstract version of the model. The counterexample should be minimized with respect to the number of free variable assignments.

The feasibility of the counterexample is checked through the use of invariant states. The full model is modified by adding a clock variable and invariant states (NuSMV INVAR clauses) that restrict the behaviour of the model so that only behaviours that follow the values of the counterexample are allowed. The clock variable is initialized at 0, and the value is incremented by 1 at each time step until the length of the counterexample is exceeded. After this, the clock value remains at the highest value (counterexample length +1). For example, for a counterexample that consists of seven states the added clock statements would be:

```
init(clock):= 0;
next(clock) := case
            clock < 7 : clock + 1 ;
            TRUE : 7;
esac;
```

Now, in order to restrict the behaviour of the full model, invariant clauses are added for each variable of the counterexample. The invariant clauses are such that the clock value implies the value of a certain variable at given times. For example, if the counterexample states that *variable1* has value TRUE at time points 1 and 6, and FALSE at other time points (time points 0, 2, 3, 4, 5) then the following INVAR statements would be added:

```
INVAR (clock = 0) -> (variable1 = FALSE)
INVAR (clock = 1) -> (variable1 = TRUE)
INVAR (clock = 2) -> (variable1 = FALSE)
INVAR (clock = 3) -> (variable1 = FALSE)
INVAR (clock = 4) -> (variable1 = FALSE)
INVAR (clock = 5) -> (variable1 = FALSE)
INVAR (clock = 6) -> (variable1 = TRUE)
```

In order to see whether the counterexample is realizable, we can find out whether the last state (in which clock is 6) is reachable from the initial state. In this running example, this can be done by checking the invariant specification:

```
INVARSPEC (clock != 6);
```

If the specification is true, it means that the end of the counterexample cannot be reached in the full model and thus the counter-example is not feasible in the full model. If the specification is false, a new trace is given as output that describes how the counterexample (the error) is realized in the full model.

Counterexample feasibility can also be checked using the model checker's own command line options. In NuSMV, it is possible to check feasibility of partial traces by executing them in the full model. This can be done through the command line option *execute_partial_traces*. This approach was not used here due to the procedure sometimes terminating and making the result hard to read.

3.7 Abstraction refinement

The idea of abstraction refinement is to find a new abstraction (i.e. a configuration of non-abstracted modules and interface modules) that is more detailed than the current configuration of the model and makes the current counterexample infeasible. The purpose is to find an abstraction level that is between the full model and the current configuration that could be model checked more efficiently but for which the refined abstraction could not result in the same counter-example that has already been extracted from the earlier abstract model. The refined abstraction can then be used to check the original invariant specification again.

As input, the abstraction refinement step requires:

- The current abstraction level (configuration of modules)
- Module/variable dependency information
- A counterexample trace
- The full non-abstracted model.

The set of modules before the abstraction refinement is denoted by *Current*. The set of modules that is added to this set is denoted by *Refinement*. This set is initially empty. The general abstraction refinement process is as follows:

1. Find out the shortest prefix *P* of the minimized counterexample that is not feasible in the full non-abstracted model.
2. Examine the last state of the prefix P. Extract variable assignments on this state. Using the dependency graph, deduce the modules that output these variables. Add these modules to the set *Refinement*.
3. Check the feasibility of the counterexample prefix *P* in the model in which the modules in *Current* or *Refinement* are non-abstracted and other modules are interface modules.
4. If the counterexample is not feasible, go to step 6.
5. Examine the dependency graph of the model. Extract modules that precede modules in *Current* or *Refinement*. Add these modules to the set *Refinement*. Go to step 3.
6. Minimize the set *Refinement* using delta debugging.

In the first step, the shortest infeasible prefix of the counterexample is looked for. All the feasibility checks used in abstraction refinement are performed as described in Section 3.6. The last state of that trace includes some variable assignments that make the trace infeasible in the full non-abstracted model (but are possible in the current abstraction). Since we want a refined abstraction that makes the counterexample infeasible, it seems logical to remove the abstractions that cause the values of these variables. We attempt this by replacing interface modules that directly influence the values of these critical variables with their non-abstracted versions. If this does not make the counterexample infeasible, we expand the set of new non-abstracted modules based on the dependency graph until the counterexample becomes infeasible. This will ultimately happen, since we know that the counterexample is not feasible in the full model.

When a successful refinement is first found, it is not necessarily a minimum refinement. It is worthwhile to keep the size of the model as small as possible. Thus, minimizing the refinement is necessary, especially since the feasibility checks required in the minimization are quite fast to perform. The objective of the minimization is to find a subset of the modules in *Refinement* that is sufficient to make the counterexample infeasible. For this minimization, we use delta debugging (the algorithm already described in Section 3.5.2) to generate subsets of *Refinement* that are checked for feasibility. The approach leads to a local minimum subset of modules.

3.8 Preliminary results

A prototype implementation of the algorithm was created in the Python programming language. In order to analyse the effectiveness of the algorithm, a model was created based on the case study in [Lahtinen et al. 2010]. The system is a function-block-based control system. The detailed implementation is not presented due to confidentiality issues. We only used a small, simplified portion of the full model and divided that resulting model into 22 modules (see Figure 1). The tested model was kept small so that the running times would also remain reasonable.

Testing the algorithm involves many aspects that should be taken into account. The model checking times vary greatly based on the selected counterexample minimization, technique and abstraction refinement technique. Other issues that affect the verification time are the checked property and our implementation that uses parallel execution. Due to the required diversity of the tests to analyse adequately the algorithm, we only give some preliminary results. A more thorough analysis is left to future research.

In the preliminary tests, we have used the delta debugging technique in counter-example minimization and in abstraction refinement. The counter-example minimization was used on the module level. The running times of the algorithm were compared with algorithms available in NuSMV:

- the standard NuSMV invariant checking algorithm: NuSMV command 'check_invar' with the command line option '–coi' (cone-of-influence reduction)
- the NuSMV invariant-checking algorithm: NuSMV command 'check_invar' with the command line options '-coi' (the cone-of-influence reduction) and '-dynamic' (dynamic variable ordering)
- the NuSMV bounded model checking algorithm for invariants: NuSMV command 'check_invar_bmc_inc'.

The model used was the same full non-abstracted model as that used in our algorithm.

The verified properties 1, 2 and 3 are random invariants that are false in the model. The properties 4, 5, 6 and 7 are derived from the requirement specification of the original system. Properties 4, 5 and 7 are true in the model. Property 6 is not true because some parts of the system have not been included in the model.

Table 1. Model checking times of the compared algorithms.

	NuSMV invariant checking	NuSMV invariant checking with dynamic variable ordering	NuSMV BMC algorithm for invariants	Our CEGAR loop-based algorithm
Property 1	0.2 s	0.7 s	0.3 s	0.9 s
Property 2	5 min 50 s	8.6 s	3.6 s	4.9 s
Property 3	0.3 s	1.1 s	0.4 s	1.2 s
Property 4	5 min 50 s	8.4 s	3.3 s	53 s
Property 5	>1h	6.7 s	5.0 s	0.6 s
Property 6	11 min 15 s	8.3 s	8.2 s	14 s
Property 7	>1h	2 min 14 s	6.5 s	2 min 25 s

The performance of the BMC algorithm is very good in all cases. The BMC algorithm also manages to prove the invariants that are true. Most BMC algorithms can only find counter-examples but not prove properties. The BMC algorithm employed in Table 1 can also prove properties but the needed bounds can be too high to do so in practice.

In the case of properties 1, 2 and 3, the algorithm discovers an abstract counterexample that is minimized and checked for feasibility on the full model. A single iteration of the algorithm is required. For property 2, our algorithm is faster that the NuSMV invariant-checking algorithms.

In the case of property 5, our algorithm discovers that the initial abstract model is true. The verification is faster than all the NuSMV algorithms.

While verifying property 4, our algorithm performs two iterations. It is still quite fast: the verification is faster than the standard NuSMV invariant-checking algorithm.

Property 7 is the most difficult to verify. Our algorithm uses three iterations to solve it. Compared with the NuSMV invariant-checking algorithms, the verification time is still quite competent.

3.9 Shortcomings of the current approach and further development

The efficiency of the algorithm depends largely on the examined formal property. In particular, if the property is such that it requires multiple iterations of the algorithm it is probable that the algorithm will not outperform traditional model checking methods. In some cases, all modules of the model may have to be analysed in order to verify a particular property. In these cases the algorithm is of no use.

In cases where the verification leads to a counterexample, a simple BMC check on the full non-abstracted model is likely to be faster than using the algorithm. This is because BMC is usually quite fast even in large models. However, traditional BMC cannot prove that a property is true. The algorithm becomes valuable when a counterexample cannot be found by BMC in reasonable time and traditional BDD-

based model checking cannot prove the property with reasonable resources (time or memory). In these cases, it is possible that the algorithm finds a sufficient subset of the modules that is computationally feasible. The algorithm can sometimes be used to prove properties of a system that cannot be otherwise proven.

Another shortcoming of the algorithm is that only safety properties can be verified. The implementation of the algorithm is currently for the invariant properties of NuSMV. Verifying liveness properties, for example, is left to future research.

The algorithm spends significant effort minimizing the counterexample traces. This is a trade-off situation. Minimization tends to support the counterexample feasibility checks and keeping the size of the abstraction small. On the other hand, if too much effort is put into minimization, the verification takes a long time (possibly more than just applying traditional model checking methods). It may be reasonable to perform minimization only on a broad level. Simple module-level minimization may be enough.

Improving the counterexample minimization step is one potential future research subject. Using a QBF (quantified Boolean formula) solver in counterexample minimization may make the minimization step faster. The counterexample minimization problem can be solved by writing it as a quantified Boolean formula (Boolean logic with quantifiers). The solutions to this formula describe a minimized counterexample.

New approaches could also be found for the abstraction refinement step. Using a MUS solver in abstraction refinement is a possible improvement. A MUS solver finds a minimal unsatisfiable core of clauses in a set of clauses (a SAT problem). Since the model checking problem can be described as a SAT problem, a MUS solver could be used to find these clauses. The set of clauses would then be used in abstraction refinement to select a minimal set of modules that makes the spurious counterexample infeasible.

Further improvements to the algorithm could be related to using assume-guarantee reasoning. Assumptions related to different modules can be used to facilitate the verification. The assumptions could then be verified separately. However, some more systematic methodology is needed.

Finally, we plan to extend the current methodology to more detailed models. We envision that fault models (as described in Section 4) and asynchronous properties (as seen in the UPPAAL models in Section 5) could be integrated into our methodology in a modular manner. This kind of modular extension of the model together with the use of the algorithm could allow the verification of very large and detailed models of a system.

4. Architecture-level model checking

This section discusses model checking of I&C safety systems at architecture level. ‘Architecture level’ in our context means that in addition to modelling the intended (software function of a) safety I&C system, we also take into account the hardware architecture of the system. In particular, hardware is modelled as a set of individual components and container elements through which the information flows. Hardware failures (possibly including a defined set of common cause failures) are included in the model to induce alterations to the information flow. The intention is to examine the effects of a set of hardware failures on the overall operation of the safety system.

The YVL guides state that a safety system (typically implemented in several subsystems) shall accomplish the safety function in the case of a single failure and simultaneous inoperability of any other component due to maintenance. Using the methodology described in this section, the realization of the safety function implemented in software can be verified using a model that also examines the behaviour of the system in all possible hardware failure cases. The methodology also allows, e.g., the analysis of hypothetical common cause failures and their effect on the safety function.

The term ‘architecture level’ in this section is only discussed in the context defined above. For instance, issues related to control room architecture, software architecture and system security are not addressed.

4.1 Model checking systems with detailed fault models

Our model checking methodology has focused primarily on the verification of logic designs. We have also analysed single-fault tolerance of these designs, but the fault models have been quite non-detailed. Typically, the exact functional behaviour of the system is abstracted to a bare minimum to focus on the fault tolerance issues and thus only the status/fault bits of an automation system have been implemented in the model. However, the behaviour of a system can be examined in more detail by creating more detailed fault models. The fault models can include physical faults such as faults in telecommunication links, microprocessor faults,

cable failures and electrical faults influencing all equipment in a cabinet. Common-cause failures (CCFs) could also be postulated in such a fault model.

Using detailed fault models, model checking could be used to analyse the fault tolerance of hardware architecture designs. If it were possible to analyse the logical design together with a fault model based on the hardware architecture, the overall system behaviour could be analysed under various assumptions. However, in this work the logical design is abstracted to a bare minimum to focus on the faults themselves. In addition to single-fault tolerance, all kinds of failure assumptions can be made on the model.

This paper presents how an I&C system can be modelled so that various hardware failures are taken into account. The model checking tool used in this work is NuSMV. The technique is intended to be an extension of our current techniques of modelling logic designs, so that these two aspects of the system could be examined in the future using a single combined model. Modelling logic designs requires as input only low-level design diagrams such as function block diagrams and a set of requirements. In addition to this, detailed fault models require a hardware architecture description of the system and a document covering the postulated failing components and their failure modes. For example, a failure mode and effects analysis (FMEA) report typically provides this information.

4.2 An example system

The fault model methodology was developed using a simple imaginary system. The example is essentially realistic though it does not encompass all the relevant details of a real I&C system. Most importantly, the example does not have any redundancy that would allow more sensible analysis of the system under various failure assumptions. The purpose of the example is to demonstrate the fault modelling methodology.

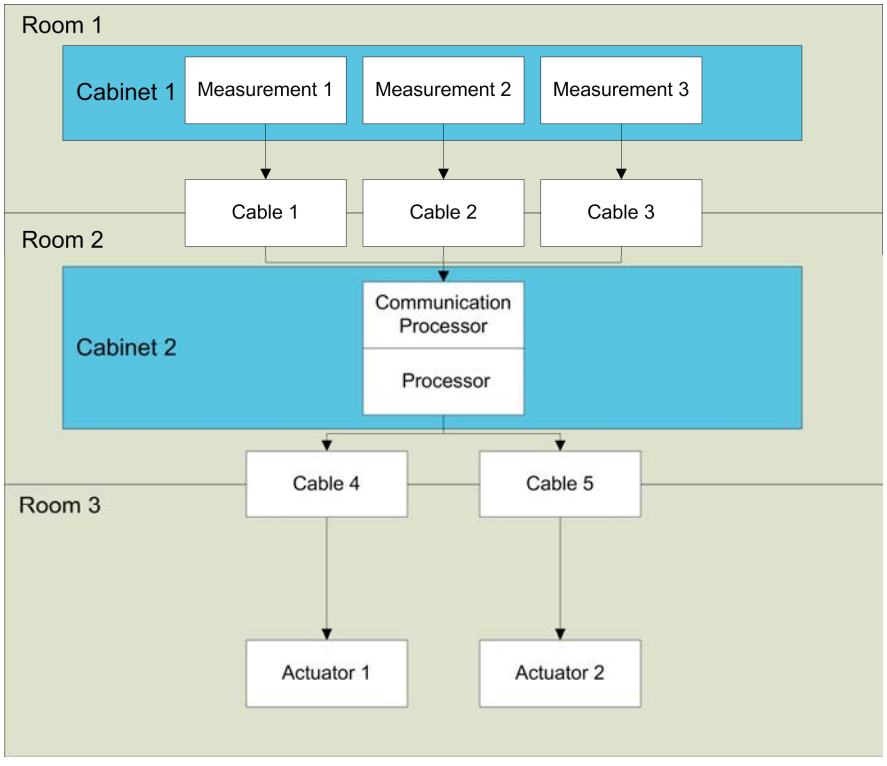


Figure 5. Hardware architecture of the example system.

The hardware configuration of the imaginary system is described in Figure 5. The system produces actuator signals based on three measurements located in Cabinet 1. Measurement 1 is of analogue type and has values in the range of 0 to 20. Measurements 2 and 3 have binary values. Each measurement is delivered through a cable to a processing unit that decides when the actuator signals are set. The processing unit located in Cabinet 2 has two parts: a communication processor that collects the input signals and a processor that does the calculations. The output signals of the processor are sent to the two actuators via two cables. Finally, all the hardware components are located inside buildings: the measurement devices inside Cabinet 1 are located in Room 1, the processing unit inside Cabinet 2 is located in Room 2, and the actuators are located in Room 3. Cables 1, 2 and 3 are located in Rooms 1 and 2. Cables 4 and 5 are located in Rooms 2 and 3.

The logical function realized by the processor of the example system is illustrated in Figure 6. The logic consists of a comparator function block, an AND function block, a set-reset flip-flop and a TON timer. The flip-flop is set whenever Input 1 has a high value (over 10) and Input 2 is true. If Inputs 1 and 2 have low values,

the flip-flop can be reset by Input 3. Output 1 is set whenever the flip-flop is set. Output 2 is set whenever the flip-flop has been set for 3 seconds.

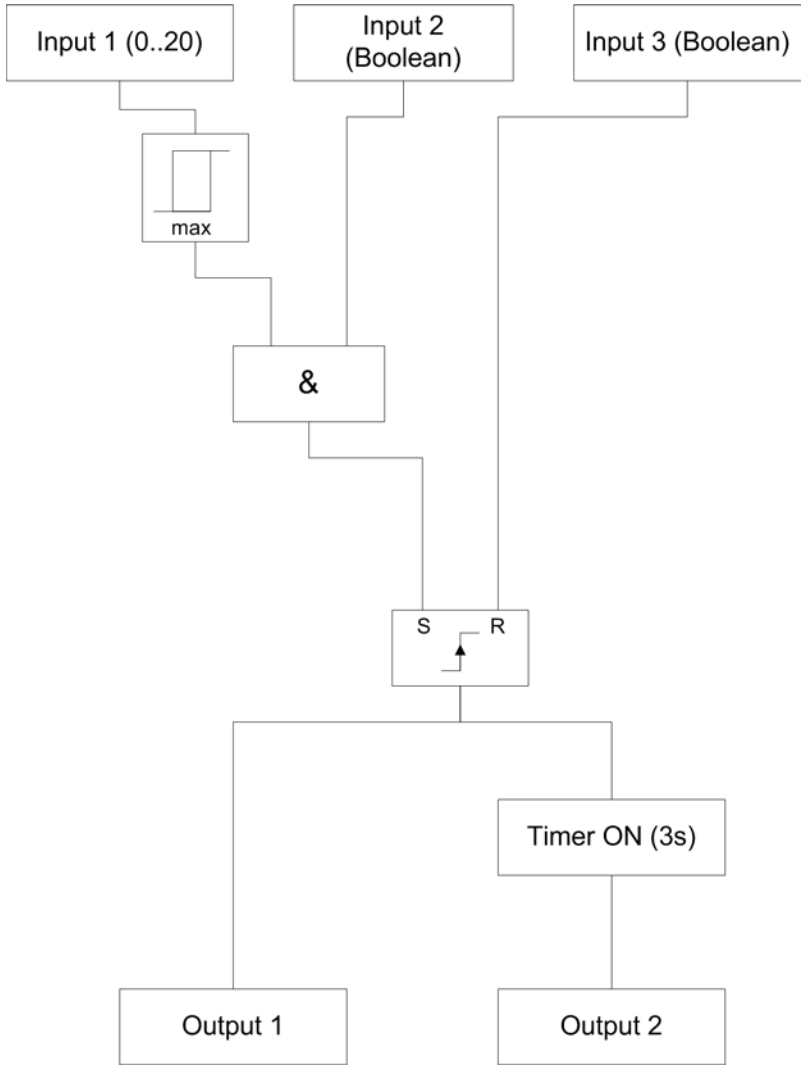


Figure 6. The logical function of the example system.

In the example system we want to be able to model both the logical function of the system and a set of failures related to the hardware structure. In the example, we assume the following failing components and failure modes (the failures can occur at any time point):

- Component 1: Analogue measurement device
  - Failure mode 1: Value stuck at minimum value
  - Failure mode 2: Value stuck at maximum value
  - Failure mode 3: Non-deterministic value (changes at every time point)
- Components 2–3: Digital measurement devices
  - Failure mode 1: Value stuck at ‘0’
  - Failure mode 2: Value stuck at ‘1’
  - Failure mode 3: Random value
- Components 4–8: Cables
  - Failure mode 1: Cable broken
  - Failure mode 2: Disturbance causing overcurrent
- Component 9: Communication processor
  - Failure mode 1: Loss of operation
  - Failure mode 2: The two Boolean signals are erroneously swapped
- Component 10: Processor
  - Failure mode 1: Loss of operation
- Components 11–12: Actuators
  - Failure mode 1: Loss of operation
  - Failure mode 2: Spurious actuation.

We also assume a set of common cause failures that lead to the failure of several components simultaneously:

- CCF 1: Cabinet 1 electrical failure leading to loss of functions in Cabinet 1. This is represented by values of the measurement devices being stuck at the minimum value.
- CCF 2: Cabinet 2 electrical failure leading to loss of functions in Cabinet 2. This is represented by the loss of function in the processors.
- CCF 3: Fire in Room 1 damages Cables 1, 2 and 3 and causes the measurement devices to fail.
- CCF 4: Fire in Room 2 causes loss of function in the processors and damages all cables.
- CCF 5: Fire in Room 3 destroys Actuators and Cables 4 and 5.
- CCF 6: Due to electromagnetic disturbance all cables experience disturbance.

In addition to the failure mode effects in the components, the consequential effects of the failures have to be identified. Special attention is needed in cases in which the failure causes the output of the component to become outside range, such as overcurrent/overvoltage/loss of signal. In our example, the following are taken into consideration:

- A disturbance in the cable is transferred to the communication processor or actuator. The communication processor identifies the failure (if it operates), and changes the status bit of the signal to TRUE. If the ac-

tuator receives this signal disturbance, and the actuator is operable, it produces a spurious actuation.

- The processor can only perform calculations using proper signals. If the received input signal has disturbances this is detected and the input used for software calculations is set to a minimum value.

4.3 Modelling methodology

The general idea is to have a model that consists of separate modules for depicting hardware and the effects of failures, and modules for realizing the software functionality. In our example system, the corresponding model has two hardware modules and one software module. The first hardware module represents the hardware, failures and information flows before the software is executed. The software module implements the logical function of the system. The second hardware module represents hardware, failures and information flow after software execution. The module composition is illustrated in Figure 7.

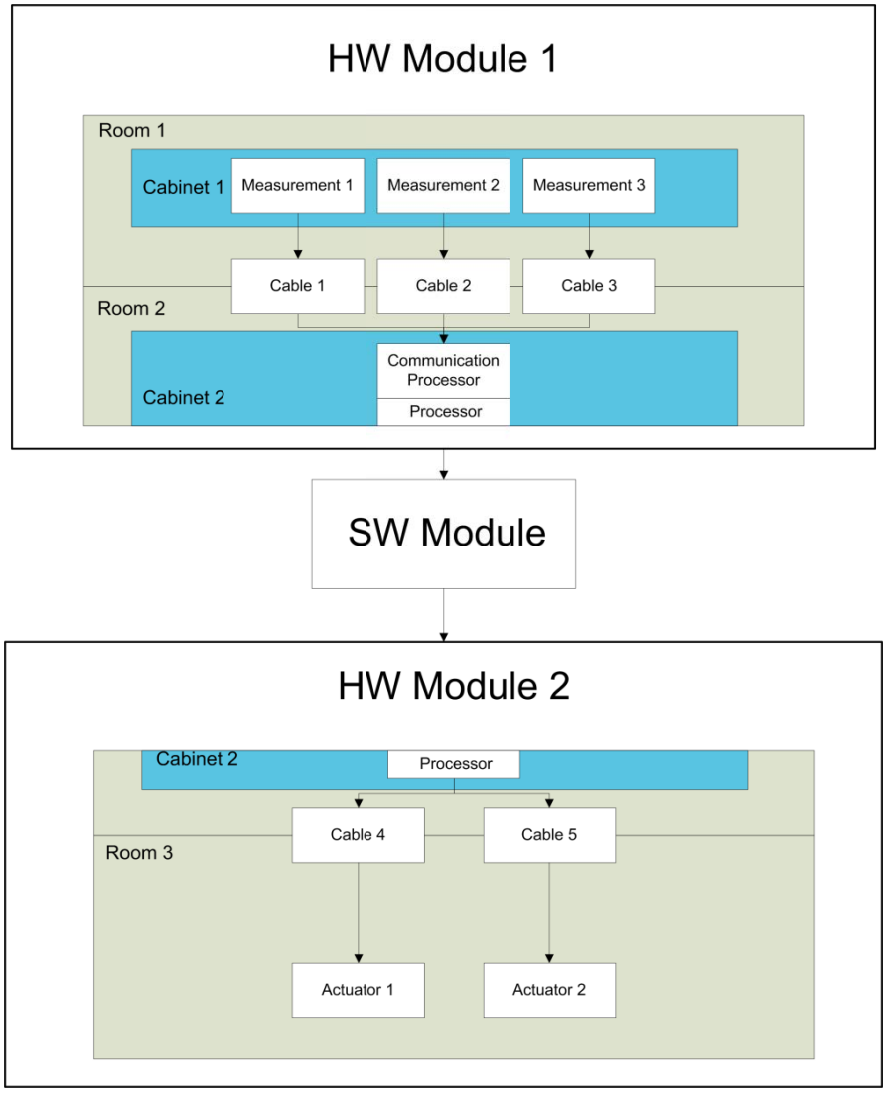


Figure 7. Module composition of the model.

4.3.1 Software modelling

The software was modelled using the traditional methods of I&C system model checking. The model is based on a small function block library that is used to implement the software function in Figure 6.

4.3.2 Hardware modelling

The hardware modules describe the behaviour of a group of hardware components and the information flows that exist between the components. In the example, there are two hardware modules (see Figure 7. HW module 1 has as input an analogue variable and two Boolean variables. The module takes into account the failures of the components it encapsulates and gives as output the values witnessed by the logic program (software module). HW module 2 transmits the outputs of the software to the appropriate actuators (again taking into account possible failure effects). The resulting actuator behaviour produced by the system is given as output of HW module 2.

Each of the hardware components (e.g. cable, processor) inside a hardware module is modelled as a sub-module. A sub-module has as input:

- The value of the signal transferred via the component (e.g. '0'/'1')
- A fault status signal (inherent actual status bit used by the software)
- The specified range of the signal including minimum and maximum value
- Signal disturbance information (if, for example, an overvoltage signal is carried instead of a signal value in the proper range)
- A variable indicating the components that will fail at a given time point (an array)
- A variable indicating the failure modes of these components at a given time point (an array)
- An identification tag of the particular hardware component.

Based on the failure information given as input, the sub-module determines the value of the signal, the fault status of the signal and whether any disturbances have arisen. These three values are then given as output and further as inputs to the next hardware component sub-module.

In our example, the cabinets and rooms were not modelled as sub-modules of the hardware module, since it was interpreted that these aspects are not actual components that directly influence the information flow. Failures of cabinets and rooms were modelled as common cause failures that cause a defined set of failures in the actual components.

The software is executed by the processor, and the processor's fault behaviour can thus be modelled in both of the hardware modules. This is done by creating a processor sub-module in both hardware modules. The first part can be used to describe failures that can be seen to occur before the calculations are performed, for example, in input reception. The second part can be used to describe failures that envelop the software outputs. (For example, it does not matter what the outcome of the software is if the processor has no power. These kinds of failures override the results of the software calculations.) In our example model, a processor sub-module was created in both hardware modules but all failures of the processor were modelled in the second part (in HW module 2).

4.3.3 Considerations on fault modelling

Our modelling methodology allows multiple simultaneous failures. Fault modelling and the issues that arise due to the multiple failure assumption are discussed in this section.

4.3.3.1 Component failures and common cause failures

Every single component can choose to fail in a way that is manifested by one of the possible failure modes related to that component. A component cannot be in two failure modes at the same time. This achieved by:

- A Boolean array *component_failure [1..12]*: For each individual component there is a Boolean variable that states whether that component will fail. The array is such that the initial Boolean values are chosen non-deterministically, and the value will not change after that.
- An array of type [1..3] *component_failuretype [1..12]*: For each individual component, the failure mode (1, 2 or 3) is selected. The initial value is selected according to the component, i.e. a cable cannot have failure mode number 3, since only two failure modes are specified. Other than that, the value is chosen non-deterministically at the initial time step, and the value will not change after that.

In addition to failures of single components, common cause failures can also be modelled. A common cause failure will affect a number of the individual components, causing them to reach one of the failure modes of that component. This is achieved by:

- A Boolean array *ccf_failure [1..6]*: A Boolean variable exists for each possible CCF scenario. The values are chosen non-deterministically and do not change after the initial time step.

4.3.3.2 Failure time dependency

The software behaviour is dependent on the time instance at which the failures manifest themselves. In order to cover all possible scenarios, our model should at least allow components to fail after a non-specified time. This is achieved by:

- A Boolean array *component_failure_realizes [1..12]*: a Boolean variable for each individual component indicating whether the failure is experienced by the system. The values of the variables are such that at each time point the Boolean value can choose to have value 1, and if 1 is chosen the value will not change anymore.
- A Boolean array *ccf_realizes [1..6]*: a Boolean variable for each CCF scenario indicating whether the CCF failure is experienced by the system. The values of the variables are such that at each time point the

Boolean value can choose to have value 1, and if 1 is chosen the value will not change anymore.

In this example, we have chosen to model failures that are permanent after the first failing time step. Any other desired temporal behaviour of failures can be created by modifying these two model variables.

4.3.3.3 Failure prioritization

We assume multiple failures. This is why there must be some prioritization of the CCF failures and failures of individual components, i.e. if an individual component fails and there is a simultaneous CCF affecting that component, how will the component behave? Which failure is dominating? The end results (which failure modes actually take place in the components) when all CCFs are taken into account are represented by another set of variables:

- A Boolean array *failure_manifestation [1..12]*: a Boolean variable for each individual component indicating whether the component failures and CCFs lead to an end effect that is experienced as a failure at a given time point.
- An array of type (1/2/3) *failure_type [1..12]*: for each individual component, the value indicates the failure mode at a given time point that is experienced when component failures and CCFs are taken into account.

Below is some actual model code of Component 1 (the analogue measurement device). The component is affected by CCFs: Cabinet 1 failure and Room 1 fire. The CCFs are presented before the component failure variable in the case structure, so that in case of a CCF occurring simultaneously with a failure in the analogue measurement device, the loss of failure caused by the CCF overrides any other component failure. The time points at which the failures occur are also taken into account (variables *ccf_realizes*, *component_failure_realizes*).

```
init(failure_manifestation[1]) := case
            ccf_failure[1] & ccf_realizes[1] : TRUE;
            ccf_failure[3] & ccf_realizes[3] : TRUE;
            TRUE : component_failure[1] & compo-
nent_failure_realizes[1];
esac;
next(failure_manifestation[1]) := case
            next(ccf_failure[1]) & next(ccf_realizes[1]) :
TRUE;
            next(ccf_failure[3]) & next(ccf_realizes[3]) :
TRUE;
TRUE : next(component_failure[1])&
next(component_failure_realizes[1]);
esac;
init(failure_type[1]) := case
            ccf_failure[1] & ccf_realizes[1] : 1;
            ccf_failure[3] & ccf_realizes[3] : 1;
```

```
            TRUE : component_failuretype[1];
esac;
next(failure_type[1]) := case
            next(ccf_failure[1]) & next(ccf_realizes[1]) :
1;
            next(ccf_failure[3]) & next(ccf_realizes[3]) :
1;
            TRUE : next(component_failuretype[1]);
esac;
```

The prioritization of the component failures and CCFs can be difficult because the issue is not typically addressed in an FMEA that primarily focuses only on a single failure occurring at a given time.

4.3.3.4 Single-fault tolerance examination

Limiting the model so that only a single failure is examined makes the verification task simpler. This could be done, e.g., by using a variable that non-deterministically chooses one of the failure cases (instead of the arrays that are used in the example). Examining only single failures also simplifies the issue of failure prioritization.

Our modelling methodology, however, also allows the analysis of single failure tolerance. The code below restricts the model to behaviour in which a single component failure or a CCF is occurring. The code creates a variable *nro_of_faults* that calculates the number of occurring component failures and CCFs (non-deterministic variables). The last line is an invariant clause that states that *nro_of_faults* should not be greater than 1. The invariant could easily be changed to any number of failures.

```
DEFINE
nro_of_faults := toint(component_failure[1]) +
            toint(component_failure[2]) +
            toint(component_failure[3]) +
            toint(component_failure[4]) +
            toint(component_failure[5]) +
            toint(component_failure[6]) +
            toint(component_failure[7]) +
            toint(component_failure[8]) +
            toint(component_failure[9]) +
            toint(component_failure[10])+
            toint(component_failure[11]) +
            toint(component_failure[12])+
            toint(ccf_failure[1])+
            toint(ccf_failure[2])+
            toint(ccf_failure[3])+
            toint(ccf_failure[4])+
            toint(ccf_failure[5])+
            toint(ccf_failure[6]);
ASSIGN
INVAR nro_of_faults <= 1;
```

4.3.3.5 Consequential failures

The modelling methodology created here allows for the examination of some consequential failures. By this we mean that, for example, a voltage spike in a cable could cause a consequential failure also in the device receiving the signal. Such cases are modelled using a parameter that carries information about the signal quality. Based on the case, this could include at least overvoltage/overcurrent, low voltage/current, loss of signal or a drift in the signal. The behaviour of the component receiving the bad signal should then be modelled in that component's sub-module. This may require prioritization: which failure dominates if there is disturbance in the input and a simultaneous component failure? If the component cannot detect the input disturbance and retransmits the value as such, this signal quality information can also be given as output of the sub-module.

Below, the definitions for two component sub-modules are given as an example of how consequential failure effects can be analysed. In hardware module 2, cables transmit the signal to the actuators. If the cable is broken (failure mode 1), the transmitted signal (output) takes a logical '0' (min) value. In addition, the signal quality output (output_errortype) also takes the value '0' indicating that the signal is lost, and it is not an actual logical '0' that is transmitted. In the case of the disturbance (failure mode 2) in the cable, the logical output is set to '1' (max), and a '2' is given as signal quality output indicating an overcurrent. In the actuator sub-module, the signal quality is received as an input (signalerror) and the value '2' (overcurrent) causes the actuator to reach the spurious failure mode. As a consequence, the logical output of the actuator is set to '1' (max). The final result is a spurious actuation caused by an overcurrent in the cable.

```
MODULE cable(var, var_FAULT, min, max, range, signalerror,
failure, failuretype,  id)
DEFINE
     broken := failure[id] & (failuretype[id]=1);
            disturbance := failure[id] & (failure-
type[id]=2) ;
            output := case
                        broken : min;
                        disturbance : max;
                        TRUE : var;
            esac;
            output_FAULT := var_FAULT;
            output_errortype := case
                        disturbance : 2;        --# overcur-
rent to the next hw component
                        broken : 0;
                        TRUE : signalerror; --# the possible
existing disturbance in the signal

            transfers through the cable
            esac;
ASSIGN
```

```
MODULE actuator(var, var_FAULT, min, max, range, signal-
error, failure, failuretype,  id)
DEFINE
            lossofoperation := failure[id] & (failure-
type[id]=1) ;
            spurious := case
                        failure[id] & (failuretype[id]=2) :
TRUE;
                        signalerror = 2 : TRUE;
                        TRUE : FALSE;
            esac;
            output := case
                        lossofoperation : min;
                        spurious : max;
                        TRUE : var;
            esac;
            output_FAULT := var_FAULT;
            output_errortype := 1;
ASSIGN
```

4.4 Application of compositional verification

The software and hardware in the example are separated. This suggests that some verification tasks could be divided into smaller subtasks that together imply correct behaviour. In fact, assume-guarantee reasoning can be applied to the verification of our example system.

In assume-guarantee reasoning, the system M is verified against a specification P by dividing the system into two parts, M1 and M2, that are verified in isolation. The system is typically expected to satisfy its requirements only in a specific context. For example, it can be assumed that M1 satisfies another specification A. Now, we can verify P on M compositionally:

1. First we verify that M1 satisfies A.
2. Next we verify that if A is assumed then M2 satisfies P. In other words the specification A→P is checked on M2.
3. These two independent verifications imply that the whole system M satisfies P.

In systems such as our example, given a specification P, it can be possible to separately verify the software functionality and after that verify the functionality of the hardware system (in specified failure conditions) assuming that the software functions as specified. In other words, the model M is divided into software (M1) and hardware (M2). The assumption A that software works as specified is first derived from P and verified on M1. Then it is verified that the hardware part (M2) satisfies the specification P if A is assumed. In the analysis of the hardware, the software module can be replaced with an interface module that has no internal functionality. The checked specification is changed into the form: ‘if the software

inputs and outputs behave as specified, then the overall system behaves as specified'.

4.5 Results

The resulting model describing the behaviour of the running example is quite large (~1000 lines of code). A major part of the model consists of the *init* and *next* clauses of the variables determining the failing components and failure modes at a given time point. This is because case structures have to be written separately for each component, each CCF and each failure mode variable.

Even though the example is quite simple, the resulting model becomes complex. This is mainly because our methodology allows multiple failures that complicate the model. The model would be more efficient without the assumption of multiple simultaneous failures. To see how the assumption of the number of simultaneous errors affects the running time of the model, two temporal logic specifications were checked on three versions of the model. The first version has an additional invariant that states that no failures are allowed. The second model allows one failure. The third model makes no limitations on the number of possible failures. The examined temporal specifications were:

- **Specification 1**: A value 20 of the analogue measurement 1 and a true value of the digital measurement 2; always cause the first actuator to actuate. In LTL this can be written as:

  ```
  G ((measurement1 = 20 & measurement2 = TRUE) → actuator1_operates)
  ```

- **Specification 2**: A value 20 of the analogue measurement 1 and a true value of the digital measurement 2; will eventually lead to the actuation of the second actuator. In LTL this is written as:

  ```
  (G (measurement1 = 20 & measurement2 = TRUE)) → F actuator2_operates
  ```

If no failures are allowed, both specifications are true. In case of failures, both specifications are false. For example, if a single failure is assumed, the first specification results in a counterexample that describes the behaviour in which measurement device 2 experiences a random failure that masks the true value. The model checking times for all model versions are shown in Table 2. We can see that when the number of assumed simultaneous failures increases, the model checking times also increase. For such a small system, the model checking time of the multiple failure model is quite long.

Table 2. Model checking times of two specifications.

	No failures	**Single failures**	**Multiple failures**
Specification 1	5s	10s	163s
Specification 2	4s	9s	199s

If the compositional assume-guarantee approach described in Section 4.4 is used, two additional specifications are first written:

- **Specification 1a**: A software input1 value 20 of the analogue input and a true value of software input2; always cause software output1 to be set. In LTL this can be written as:

  ```
  G ((input1 = 20 & input2 = TRUE) → output1)
  ```

- **Specification 2a**: A software input1 value 20 of the analogue input and a true value of software input2; will eventually lead to software output2 set. In LTL this is written as:

  ```
  ((G (input1 = 20 & input2 = TRUE)) → F output2)
  ```

Specifications 1a and 2a are separately checked on models that consist only of the software module. Both specifications are true. The model checking time is << 1s in both cases. After this, the software module in the model of the overall system is replaced with an interface module in which the internal behaviour is removed, and the two software outputs are changed into non-deterministic Boolean variables. This modified model is then checked against specifications:

- **Assume-guarantee specification 1**: Whenever specification 1a is true, specification 1 is also true. In LTL this can be written as:

  ```
  G ((input1 = 20 & input2 = TRUE) → output1)
  →
  G ((measurement1 = 20 & measurement2 = TRUE) →
  actuator1_operates)
  ```

- **Assume-guarantee specification 2**: Whenever specification 2a is true, specification 2 is also true. In LTL this can be written as:

  ```
  ((G (input1 = 20 & input2 = TRUE)) → F output2)
  →
  ((G (measurement1 = 20 & measurement2 = TRUE)) →
  F actuator2_operates))
  ```

Table 3. Model checking times using the assume-guarantee approach.

	No failures	**Single failures**	**Multiple failures**
Assume-guarantee 1	3s	5s	7s
Assume-guarantee 2	3s	5s	9s

The model checking times using the assume-guarantee approach are shown in Table 3. We can see that the assume-guarantee-based verification approach is much more effective.

4.6 Remaining problems

The most important problem in our example seems to be that even a model of a simple system quickly becomes quite complex. The application of assume-guarantee reasoning has a significant effect on the verification time of the system. Limiting the scope of the analysis by, e.g. focusing only on single failures, simplifies the verification.

The single-failure analysis of safety systems could be made more efficient by integrating the approach to the algorithm used for model checking large systems described in Section 3. The integration would require the system (software, hardware and fault modelling) to be modelled in a compatible manner. Using separate modules for software and hardware is a good starting point. In the running example discussed in this report, the software module is already compatible with the algorithm for large systems, as it can be replaced with an interface module. Similar modelling techniques for abstracting the hardware modules are probably needed. Furthermore, a major part of the failure model is currently part of the main module of the model. This behaviour needs to be encapsulated in a separate module or integrated with the hardware modules. The role of assume-guarantee reasoning together with the algorithm is also an open matter. Creating more systematic methodology for large systems and detailed fault models is left to future research.

Another practical problem is the modelling of communication architectures. In our example, all connections were point-to-point, which made modelling the information flow easy. Safety systems may, however, implement all kinds of network topologies (e.g. serial bus) to transmit signals. The modelling of these issues is left to future research.

Failures in hardware components are frequently discussed in the context of probabilistic reliability analysis. The methodology developed here should be made consistent with these already existing concepts and methods. Differences between the two approaches have not yet been identified. Merging this method with the reliability analysis environment is left to future research.

Finally, our fault models could possibly be used in the identification of new common-cause failures. For example, if a new consequential failure effect in the system is identified or postulated, it may not be clear how it affects the overall system. The methodology used here could be used to analyse the overall effects of hypothetical consequential failures in the system.

5. Asynchronous techniques for modelling timed automata

5.1 Introduction

5.1.1 Work description

In this work, we look into techniques for modelling safety-critical systems represented using function block diagrams. Three alternative techniques are presented for modelling, each with a different approach to the problem. The techniques are based on the methodologies designed in the MODSAFE project, as described in [Ropponen 2010]. Furthermore, the techniques are evaluated based on the simplicity of the modelling and the efficiency of the analysis according to UPPAAL.

We form a component library for UPPAAL to construct systems consisting of the components. The component library should be as modular as possible to offer flexibility when similar function blocks are found in other systems. This enables easy modelling by combining individual components into the entire system by parameterization. However, some of the techniques sacrifice modularity for verification speed, having the modeller make changes to the components themselves depending on the system. In this case, some components of the function block chart are only represented by functions of the templates of the UPPAAL model.

Three case studies are used to evaluate the techniques: two have been completed and one is currently being worked on. The case studies are models of electrical circuits from safety-critical automation components. The automation circuits use fault signalling in addition to normal signals: if a fault is detected in the inputs to the circuit, the fault signal is propagated to areas that are affected by those inputs. Models without fault signal processing were also made for comparing the impact to performance.

5.1.2 The UPPAAL model checker

UPPAAL is a real-time model checker, which can model, validate and verify real-time systems. The systems are modelled with networks of timed automata. The state of the system in UPPAAL includes the locations of the automata as well as

the values of the clocks and integer variables. UPPAAL is available at http://www.uppaal.com. [Behrmann et al. 2004]

When the time scales of the delays of the system vary greatly, it is difficult to verify properties of the system with conventional testing. In these cases, UPPAAL is at its most useful as a model checking tool. On the other hand, having a large number of input signals makes UPPAAL models particularly slow to verify.

5.2 Modelling Techniques

In the following section, we present three techniques for asynchronous modelling of function block diagrams. The first technique uses standard methods for modelling with UPPAAL, with a UPPAAL component for each component in the system. The second technique replaces certain components with functions and the third technique expands the second using an external program to determine possible output combinations of the initial time-independent section of the system.

5.2.1 Standard asynchronous modelling technique

This technique is the asynchronous modelling methodology described in the MODSAFE project, as seen in [Ropponen 2010]. Each input and component in the system has its own component in the UPPAAL model as well as its own synchronization channels for input and output. Since the technique uses asynchronous modelling, the system does not have system-wide clock cycles for synchronizing the components. Instead, the model does not make any assumptions concerning the order in which the components are updated, thus making it possible to examine all possible orders of events.

This approach makes modelling relatively straightforward to implement with UPPAAL. When modelling the system, it suffices to add new components to the system declarations page and new variables to the declarations page of the project. However, every component of the system with its own component in UPPAAL increases the size of the model and slows down the verification compared with a model in which some components are replaced with functions. The time-independent components can be modelled with small automata consisting of only a few locations. On the other hand, the time-dependent function blocks have larger automata and require the use of clocks.

5.2.2 Function-based asynchronous modelling technique

With the function-based modelling technique, the time-independent components of the system are not always represented by their own components in the UPPAAL model. Instead, the output of a time-independent section of the system may be calculated in the preceding input component or time-dependent component with the help of functions. Before an edge that produces a synchronization signal is

taken, the output of the following group of time-independent components is calculated. The result is relayed as input to the next time-dependent component. For example, the calculation of the output of all time-independent components before the first time-dependent components is included in the input components. Furthermore, now all the input signals originate from a single component, which changes the value of one input at a time.

Time-independent components in later parts of the system can also be replaced with functions. The technique is similar to the one used for the input components, but synchronization channels are added from the time-dependent components with the input values. It should be noted that every synchronization channel for an input needs its own edge, making modelling more complicated than for input components.

This approach decreases the space required by the UPPAAL model because the model has fewer time-independent components and input components. This may increase verification speed since calculating only with functions is significantly faster than taking transitions in separate components. As a result, the function-based technique can be especially helpful when the system has many inputs and time-independent components. With this technique, UPPAAL can focus on its strength in modelling time-dependent components with real-time clock variables, while time-independent functions are easier to model with functions.

However, the input component calculates all the outputs of the time-independent section regardless of the change in input value. This is not always the case with the standard modelling technique, since calculation may stop quickly if the change in input does not change the output of the following components.

Furthermore, the technique makes modelling more complicated. With the standard technique, it suffices to add components to the system declarations page and variables to the declarations page. With the function-based technique, the user also has to make changes to the functions and parameters of the components, although the modifications of the components resemble the contents of the system declarations page to some extent. This decreases the modularity of the components, thus making the components system-dependent.

5.2.3 Function-based asynchronous modelling technique with input reductions

This technique is based on the previous one, with the addition that an external program is used to determine the possible combinations of outputs of the initial time-independent section of the system. The input component of the UPPAAL model is then modified so that it only produces possible combinations of these output values. Many of these outputs are given as inputs to the first time-dependent components of the system. However, there are also outputs of the initial time-independent section that are outputs of the entire system and do not affect the time-dependent components. If the modelling of these outputs is consid-

ered unnecessary, they may be removed from the model to increase verification speed and decrease the effort of modelling.

This technique decreases the verification time by reducing the number of possible input combinations and removing the need to calculate these values separately. The technique tends to be useful for systems with a large time-independent section preceding the first time-dependent components and many input signals compared with the number of outputs of the time-independent section. More precisely, the more combinations of outputs that can be reduced by the program, the better this technique is for the system in question. On the other hand, for some systems the technique does not provide any impossible output combinations. Even then, removing the initial time-independent section of the system increases verification speed.

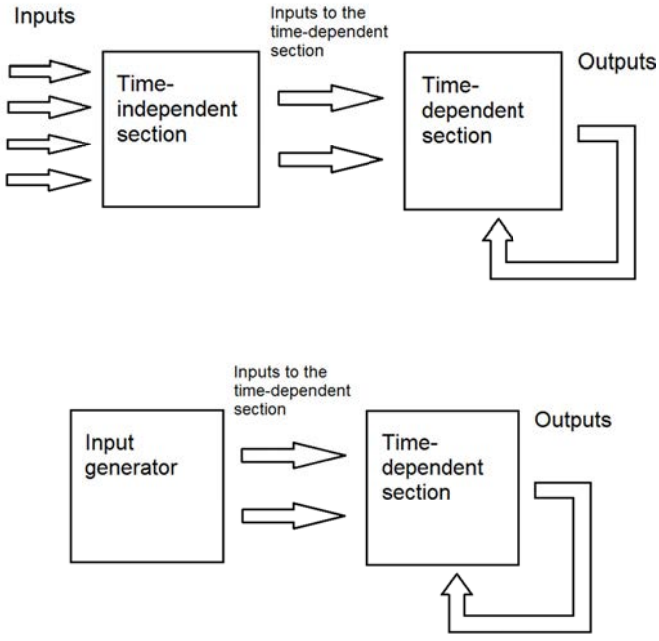
However, it is not always best to explicitly list all possible combinations if the list is overly long. It is most beneficial to list values for the inputs whose combinations decrease most, while inputs that are relatively independent from others may be modelled with functions, similarly to the previous technique. The user of this technique has to analyse the data given by the external program carefully to determine the best course of action.

This technique makes modelling significantly more complicated, as you must first construct the time-independent section of the system with the external program, have it calculate outputs and then modify the input component of the UPPAAL model. However, for simple systems, the possible output combinations can be deduced without the help of the program. It is particularly easy to determine the possible combinations of fault inputs because the value 1 of a fault variable generally spreads to all following components.

If there are large clusters of time-independent components between time-dependent components, it may be worthwhile to use this technique even when the first time-dependent components are early in the system. Namely, it is possible to examine the outputs of the time-dependent components and then use the program to determine which combinations of inputs for the next time-dependent components are possible.

The external program used was written in Java with the help of the JavaBDD library. The program takes as input the BDDs, binary decision trees, describing the time-independent section of the system. For each output of the time-independent section, a BDD is given as input. The program then constructs a BDD that describes whether there is a combination of inputs that results in a specific combination of outputs of the time-independent section. Finally, the program prints the truth values satisfying the BDD, in other words the possible combinations of output values for the time-independent section.

The following pictures demonstrate the functionality of this technique. The first picture describes modelling with regular inputs, while the second picture describes modelling with input reductions.



5.3 Modelled Components

5.3.1 Standard asynchronous modelling technique

The components of the system and their corresponding UPPAAL templates are as follows: IN1, AND, OR, NEG and FF_STAT_R are time-independent components, while ONDELAY, OFFDELAY, PULSE and LIMIT are time-dependent.

The basic function of fault variables is similar in all components. If at least one of the input signals has 1 as the value of its fault variable, the output signals of the component also have 1 as the value of their fault variables. If the fault variable becomes active, the internal memory of FF_STAT_R and the time-dependent components is not changed until the fault variable has receded. In most cases, the value of the output will not change until the value of the fault variable of the input has returned to 0.

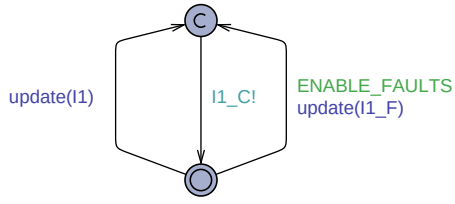


Figure 10. An input automaton.

Component IN1 models a non-deterministic input signal. The produced signal is binary, as required by UPPAAL, so 0 and 1 are the only possible input values. Each transition in the IN1 component is equivalent to the value of the input signal changing. When the value of the input signal changes, a synchronization signal is sent to the affected components. There are separate transitions for changes in the fault variables, which can be deactivated with the ENABLE_FAULTS guards.

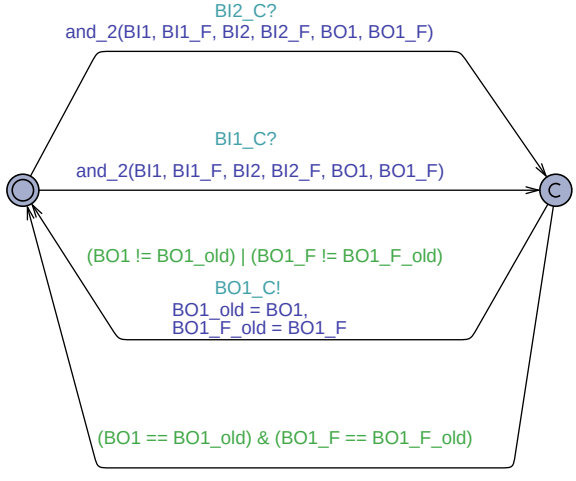


Figure 11. An automaton for the AND function.

Components AND, OR and NEG function like their counterparts in Boolean logic. AND gives 1 as its output signal only if both input signals are 1, OR gives 1 if at least one of the input signals is 1 and NEG inverts the input signal. These components are particularly simple and only require a few locations and transitions. AND, OR and NEG are similar in structure, the main difference being the change in functions. Thus, only the figure of AND is displayed.

FF_STAT_R is a static RS flip-flop, with the preferred state on the reset side (the R side) and priority on the set side (the S side). The UPPAAL model of FF_STAT_R is similar to AND and OR in structure but with a different function handling the change in output. The behaviour of FF_STAT_R is best described by its truth table. The symbol X describes a situation in which the value of the input variable is irrelevant. After each cycle, the value of Output 1 is stored in the internal memory of the component.

Table 4. Truth table for a static RS flip-flop.

Input 1 (S)	Input 2 (R)	Output 1, previous cycle	Output 1 (S)	Output 2 (R)
0	X	0	0	1
0	0	1	1	1
0	1	1	0	1
1	X	X	1	0

It can be seen that output 2 is the complement of output 1. Furthermore, if input 1 has the value 1, output 1 will be 1 regardless of the other input signals. This is called the set command of the flip-flop. If input 1 is 0 and input 2 is 1, output 1 will be 0. This is called the reset command. If the fault variable becomes active, the internal memory will not be changed until the fault variable has receded. In addition, the output signals will retain the value of the last faultless cycle.

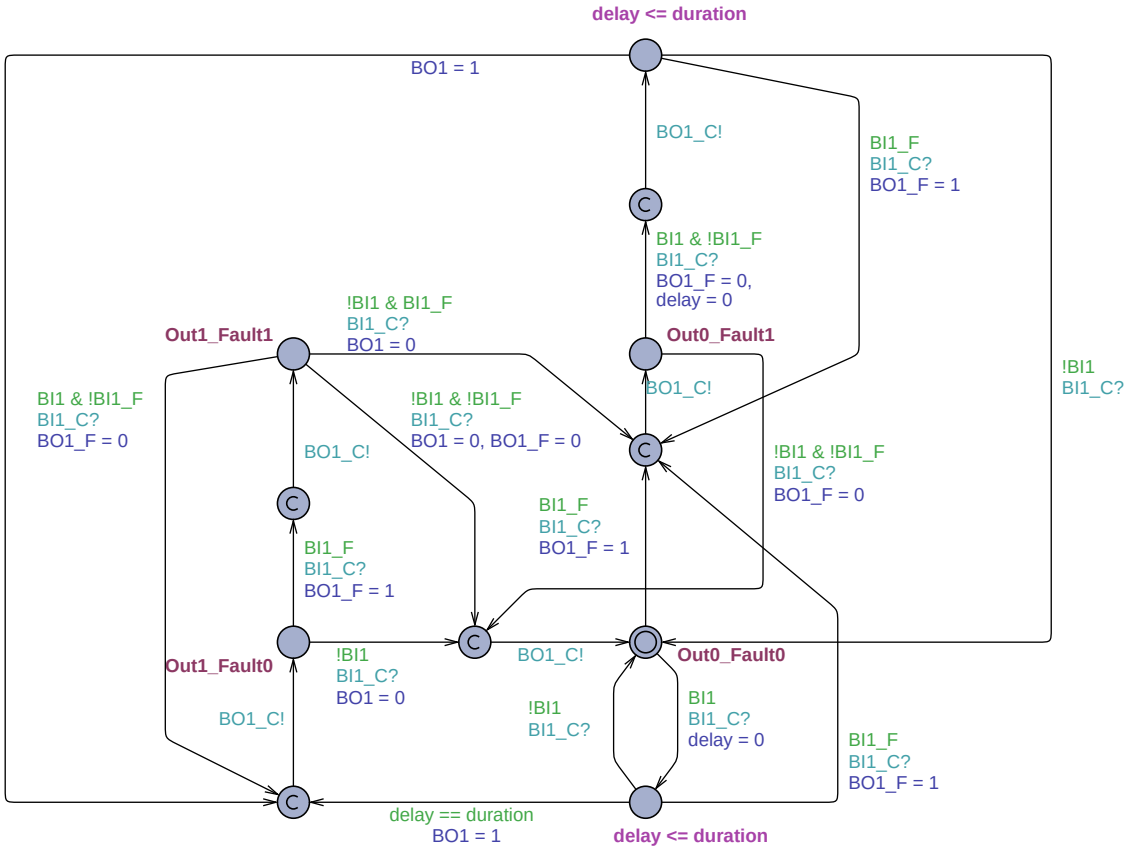


Figure 12. An automaton for the ONDELAY function block.

ONDELAY and OFFDELAY have a predetermined delay before the value of the output is changed according to the input. In ONDELAY, when the input signal changes its value from 0 to 1, a timer is set. After a sufficient amount of time has passed, if the input is still 1, the output of the component is also set to 1. However, if the input changes to 0, the output is immediately set to 0. OFFDELAY behaves similarly: if the input changes from 1 to 0 and remains at 0 after the specified time, the output is set to 0. If the input changes to 1, the output is immediately set to 1. In both ONDELAY and OFFDELAY, time counting is stopped when the fault variable has the value 1. The UPPAAL template of OFFDELAY is also similar to the model of ONDELAY, so only the figure of ONDELAY is shown.

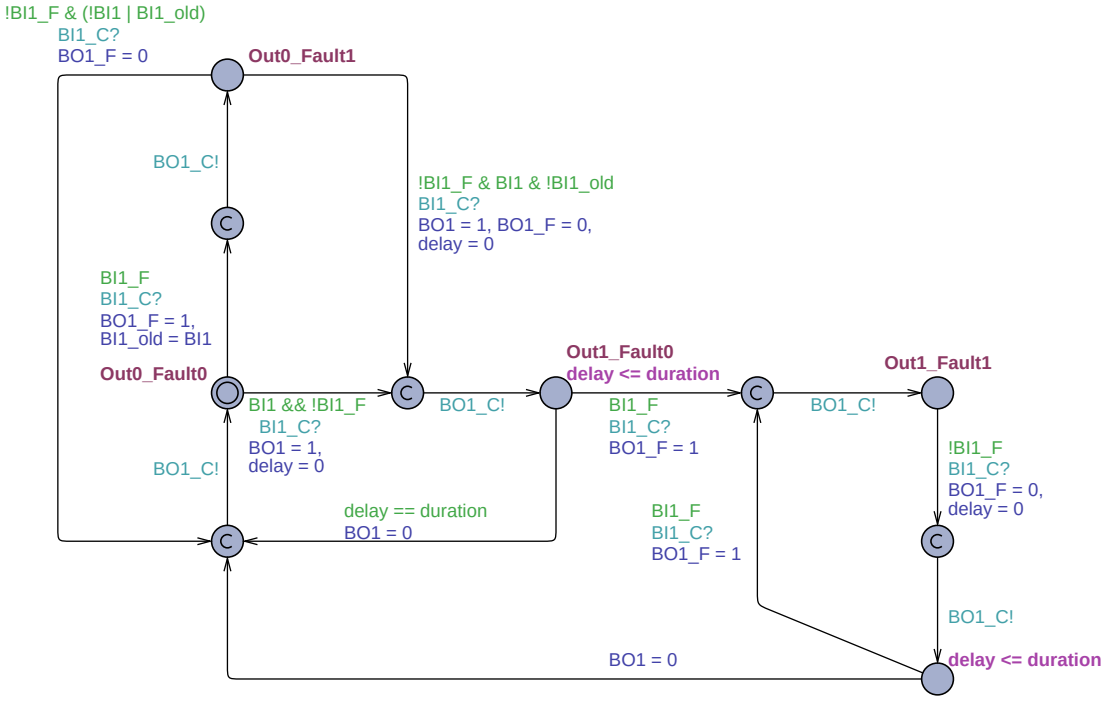
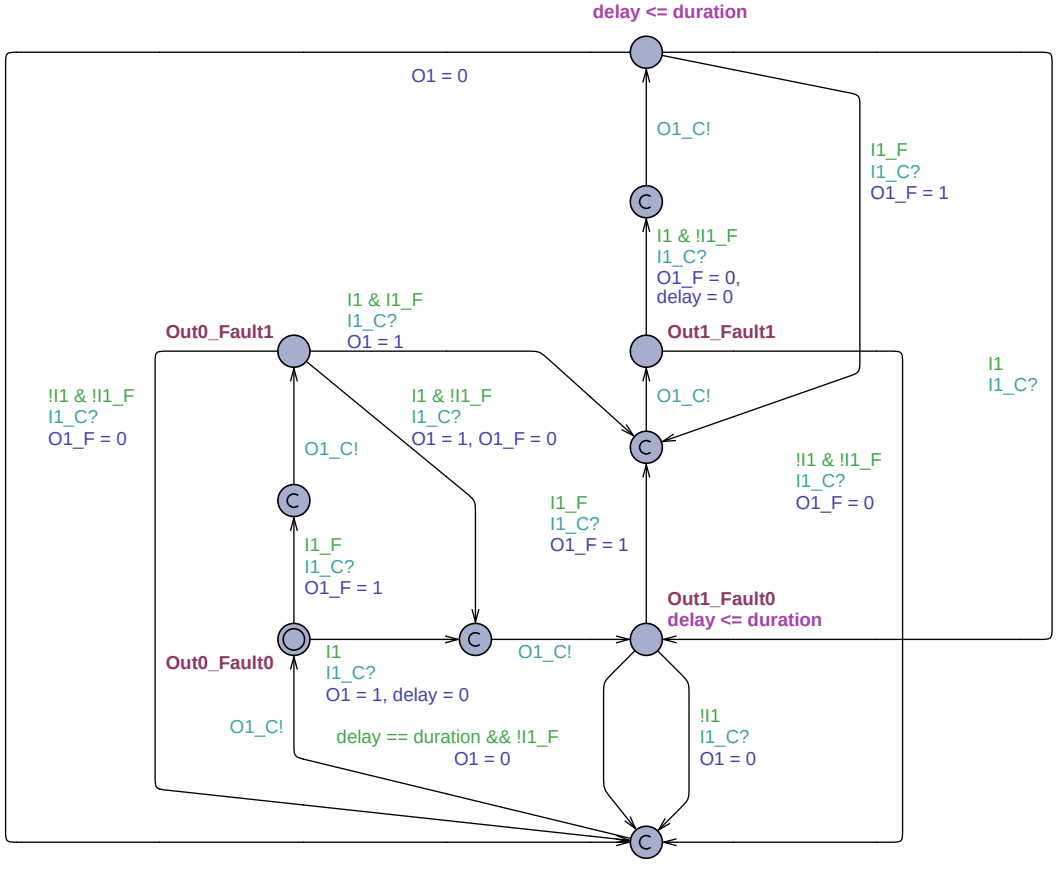


Figure 13. An automaton for the PULSE function block.

PULSE produces a pulse of the output signal for a specified time. When the input signal changes from 0 to 1, the output is set to 1 and time begins to elapse. The output remains at 1 until the time has passed, at which point it changes back to 0. When the fault variable is 1, time counting is stopped and the output remains at the value of the last faultless cycle.



LIMIT produces an output signal of 1 for an input of value 1 until the input changes to 0 or a specified amount of time has passed. Otherwise, the output of LIMIT is 0. If the fault variable is 1, time counting in LIMIT is assumed to stop, similarly to ONDELAY and OFFDELAY.

5.3.2 Function-based asynchronous modelling technique

Most of the components are essentially unchanged compared with the standard asynchronous modelling technique, with the exception of the input signals. However, certain conditions regarding the fault variable have been added to the guards in case several of the input signals can change simultaneously. This has also resulted in an additional edge in the components ONDELAY, OFFDELAY and LIMIT from the location Out0_Fault0. These additions improve compatibility with input components that allow several input values to change simultaneously.

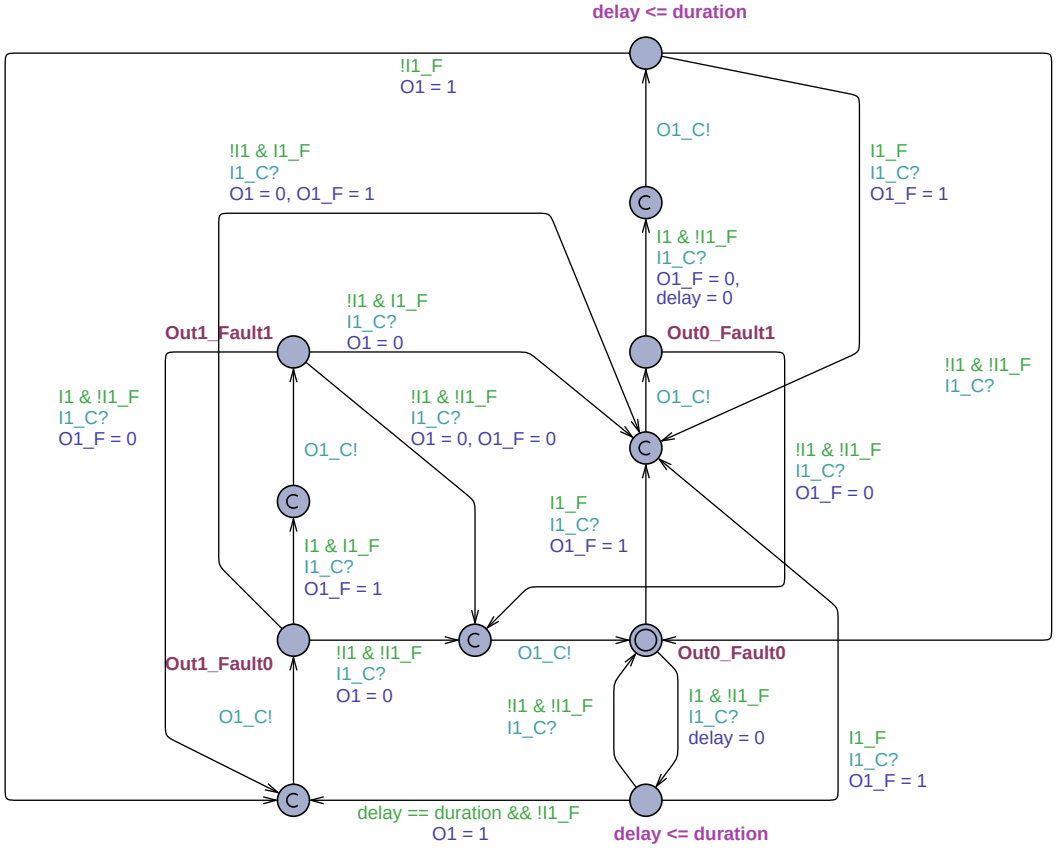


Figure 15. Function-based ONDELAY automaton.

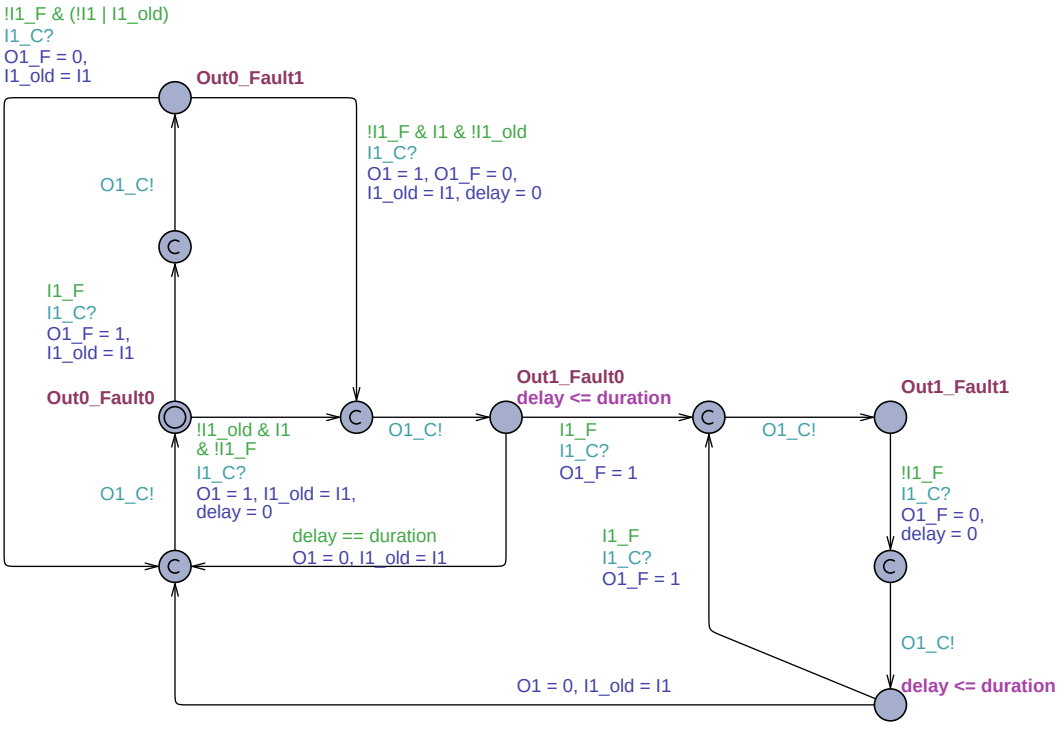


Figure 16. Function-based PULSE automaton.

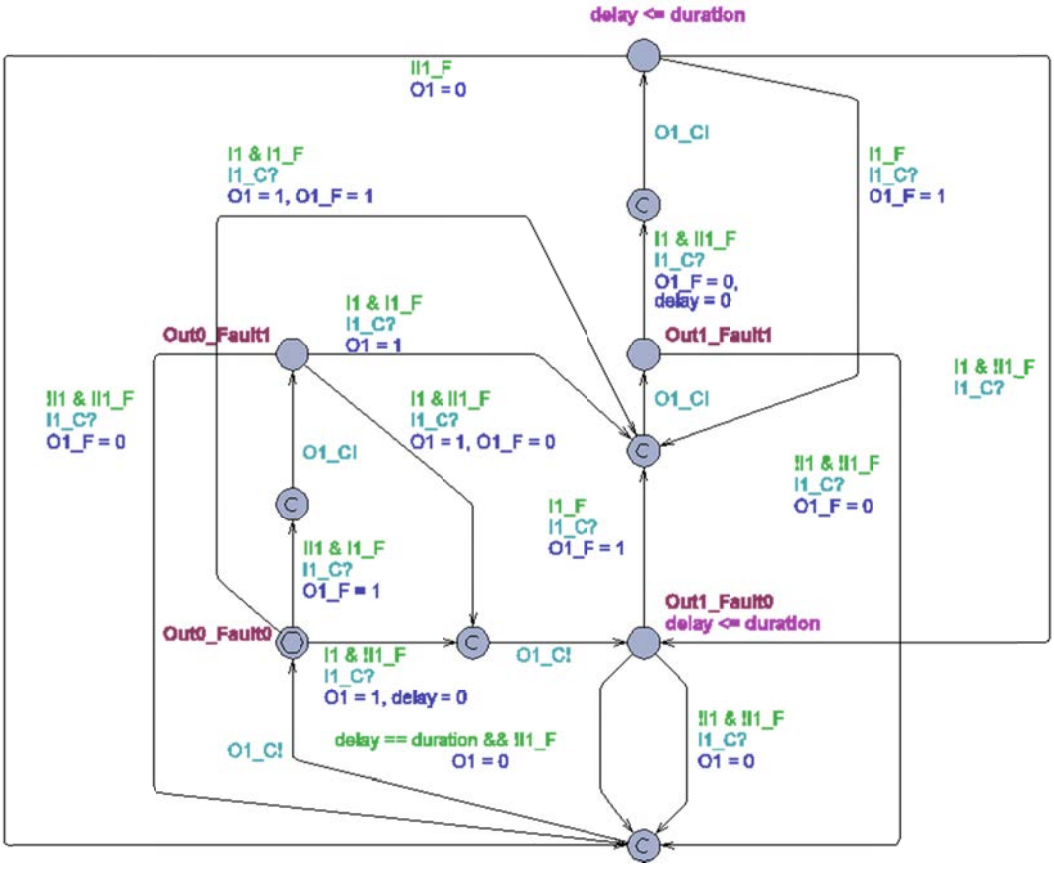


Figure 17. Function-based LIMIT automaton.

Instead of each input signal having its own component, all the values of the inputs are now generated in a single component, IN_ALL. It also contains the time-independent section of the system before the first time-dependent components.

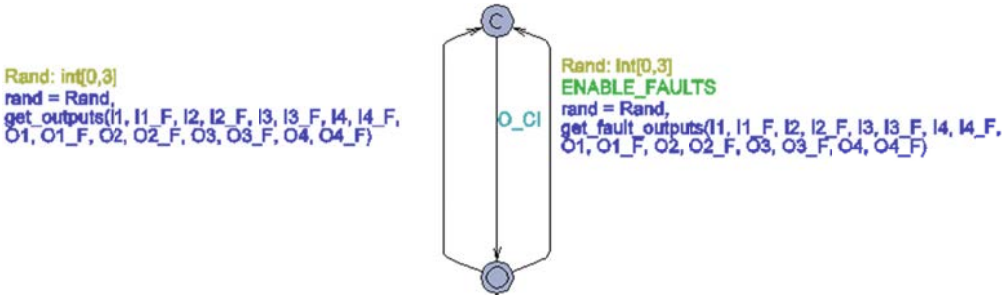


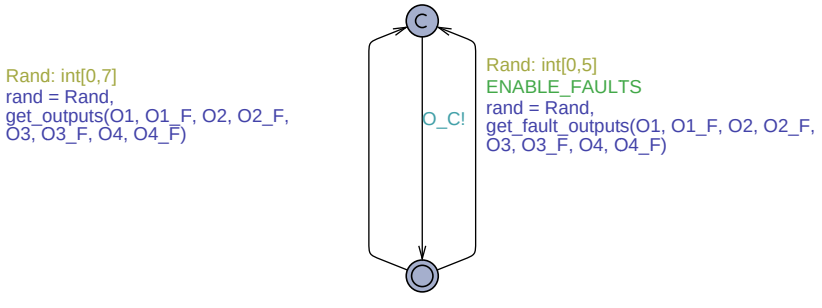
Figure 18. IN_ALL automaton.

IN_ALL changes the value of one input signal, determined by the value of the integer Rand. The affected input signals are listed on the declarations page of IN_ALL, one input for each value of Rand. Values of the fault signals can only be changed if the fault signals are enabled. IN_ALL also computes the outputs of the initial time-independent section and gives them as inputs to the time-dependent components.

The declarations and the maximum value of Rand depend on the system in question. The picture above is from one of the case studies.

5.3.3 Function-based asynchronous modelling technique with input reductions

Compared with the previous technique, the only UPPAAL component changed is IN_ALL. Instead of changing the value of one input signal, it now chooses one combination of inputs out of the possible combinations determined by the external program. Similarly to the previous technique, the input combinations are listed on the declarations page of IN_ALL, one for each value of the integer Rand. As before, IN_ALL uses these inputs to compute the outputs of the initial time-independent section.



5.4 Java program

5.4.1 Description of the program

The function-based asynchronous technique with input reductions uses an external program to determine which combinations of outputs of the initial time-independent section are possible. The program was written in Java with the help of the JavaBDD library, which the program uses to construct and modify BDDs, binary decision diagrams.

In the context of the function-based asynchronous technique with input reductions, the BDDs describe the time-independent section of the system. The program constructs a BDD that describes whether there is a combination of inputs that results in a specific combination of outputs of the time-independent section. Then, the program prints the truth values satisfying the BDD, in other words the possible combinations of output values of the time-independent section. The program can either construct the BDDs directly with the JavaBDD library or read input BDDs from files.

The outputs are often given as inputs to the first time-dependent components of the system but there are also outputs of the time-independent section that are outputs of the entire system and do not affect time-dependent components. If it is not necessary to model these outputs, they may be removed from the model to increase verification speed and decrease the effort of modelling. For each output dropped, the corresponding BDD will be left out of the execution of the Java program.

5.4.2 Example reduction

We now examine a small, hypothetical example of a time-independent section of a system and describe the resulting input reductions when the Java program is run for it. The time-independent section consists of two inputs and two outputs, with output 1 being a conjunction of the inputs and output 2 being a disjunction of the inputs. The following chart demonstrates the possible output combinations.

Table 5. Possible output combinations in the example.

Input 1	Input 2	Output 1 (AND)	Output 2 (OR)
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

It can be seen that there are only three possible combinations of outputs: (0, 0), (0, 1) and (1, 1). The combination (1, 1) is impossible and thus, it does not have to

be modelled. Naturally, the presented example is so simple that the possible output combinations can be determined without using the Java program.

5.5 Modelled Systems

5.5.1 Case study: emergency tank system

In the case study, we model a part of an emergency tank system. First, we examine the function of the modelled system and the individual components in it. Then we model the problem with different modelling methodologies, evaluate them and compare them with each other.

Other parts of the system of the power plant have been abstracted away, as well as the behaviour of the environment of the system. As a result, we make no assumptions on the behaviour of the environment or the parts of the system left outside the system we are examining. The necessary inputs from other parts of the system are modelled non-deterministically.

The emergency tank system consists of four identical subsystems, each corresponding to an emergency tank. The purpose of the system is to monitor the surface levels of the tanks, with each subsystem sending information on its tank to two of the other subsystems. Every subsystem monitors three surface levels and if at least two of them are too low, a signal is sent to the other components of the subsystem.

In addition to testing the system as a whole, we examine a smaller version of it by removing several inputs, outputs and time-independent components. This approach may reveal how the techniques scale for systems of different sizes.

5.5.2 Case study: emergency diesel system

We also look into systems modified from the emergency diesel system examined in the MODSAFE project, as described in [Ropponen 2010]. The system in question manages the function of a diesel generator of a power plant in case of emergency. The techniques evaluated in this work partially differ from those used in MODSAFE, so certain changes have been made to the systems. In particular, some sections of the systems have been cut to make modelling easier and some time-dependent components have been removed to better test the function-based techniques.

5.5.3 Case study: power reduction unit

Another case study that is being worked on is the modelling of a power reduction unit. This case is done to further the task 3.2 in the SARANA project, the objective of which is to develop modelling methods for asynchronous and semi-synchronous

systems. Current modelling methods either assume that the system is fully synchronous or that there are no constraints on the timings of signal changes.

In reality, there are systems for which neither assumption holds: either there are multiple subsystems with different clock signals, resulting in a semi-synchronous system, or there is no explicit clock synchronisation, but some delay elements are present. In the latter case, it is reasonable to assume that the timing of fast components (e.g. logic gates) can happen in any order, but the delay elements function much more slowly. It is therefore not reasonable to assume that the output signal of a delay element changes before the logic gates have had time to update theirs. Accurately modelling and verifying such systems is a problem for which no definitive solution exists.

The power reduction unit monitors the output rate of a process and an array of pumps that produce a critical resource for the process. If one or more of the pumps stop working, the output rate of the process must be decreased and the reduction unit output a signal that tells the process to slow down.

The reduction unit is implemented as redundant asynchronous circuits that include timing components. The circuits can monitor the output rate via two measurements: one that is accurate and one that updates quickly. To use both of these measurements, a correction parameter is applied to the quick but less accurate measurement. When the corrected value appears to differ from the accurate one, the correction is adjusted until it matches the accurate value again.

Modelling and verifying this design has unique challenges related to the timing and synchronisation issues, and solving them will hopefully lead to advances in the modelling and verification techniques. The models are built and verified both in the NuSMV and UPPAAL model checking tools, as both have strengths and weaknesses with systems like these: NuSMV can handle very large state spaces with a lot of non-deterministic inputs, but UPPAAL can model timing more accurately. Ultimately the strengths of both the tools need to be combined to verify complicated asynchronous and semi-synchronous systems. This work started in late 2011 and will continue in 2012.

5.5.4 Properties of the systems

The number of input signals and the number of time-dependent and time-independent components greatly influence the time spent on verifying the properties, so they are considered critical variables for the testing process. The number of inputs given to the time-dependent components and the number of outputs of the initial time-independent section are also important because of the way the function-based modelling techniques operate.

- System 1: Emergency Tank System, small version
- System 2: Emergency Tank System, full version
- System 3: Emergency Diesel System, Subsystem 1, small version
- System 4: Emergency Diesel System, Subsystem 1, full version

Table 6. Properties of the modelled systems 1.

System	Inputs	Inputs for the time-dependent components	Outputs of the initial time-independent section	Outputs
1	4	4	4	8
2	8	4	8	12
3	8	1	2	3
4	13	1	2	3

Table 7. Properties of the modelled systems 2.

System	Time-dependent components	Time-independent components (with the standard technique excluding inputs)
1	4	14
2	4	22
3	2	11
4	2	18

5.5.5 Verified properties

The modelled systems differed in structure, so comparing them by verifying system-specific properties would have yielded inconsistent results. Therefore, the same property was verified for all systems: whether the system had a path that led to a deadlock. Assuming the systems work properly, this property should be satisfied for all of them.

5.6 Results

5.6.1 Verification results

The number of input signals and the number of time-dependent and time-independent components, as well as the number of input signals given to the time-dependent components and the number of output signals of the initial time-independent section, are important variables for the testing process. After all, these variables greatly influence the time spent on verifying the properties. See 5.5.4 for the values of these variables for the examined systems. However, the structure of the system is also important, including where the time-dependent and time-independent components are located in the system.

The model checking was performed on a standard PC with 8 GB of RAM and an Intel Core i5-2500 processor running at 3.30 GHz.

The following results were obtained with UPPAAL version 4.0.11 using default settings.

Table 8. Verification times for the deadlock property without fault signals.

System	Standard	Function-based	Function-based with input reductions
1	0 min 2.821 s	0 min 0.349 s	0 min 0.057 s
2	1 min 34.153 s	0 min 0.563 s	0 min 0.083 s
3	0 min 0.441 s	0 min 0.753 s	0 min 0.063 s
4	1 min 1.262 s	2 min 9.264 s	0 min 0.418 s

Table 9. Verification times for the deadlock property with fault signals.

System	Standard	Function-based	Function-based with input reductions
1	> 27 min 10.779 s *	0 min 5.498 s	0 min 0.442 s
2	> 41 min 35.992 s *	0 min 7.316 s	0 min 0.708 s
3	1 min 19.402 s	14 min 36.620 s	0 min 0.357 s
4	> 15 min 15.274 s *	> 19 min 59.385 s *	0 min 17.224 s

* Out of memory

The following results were obtained with UPPAAL version 4.1.4 using default settings.

Table 10. Verification times for the deadlock property without fault signals.

System	Standard	Function-based	Function-based with input reductions
1	0 min 2.070 s	0 min 0.522 s	0 min 0.048 s
2	1 min 12.756 s	0 min 0.266 s	0 min 0.063 s
3	0 min 0.333 s	0 min 0.525 s	0 min 0.036 s
4	0 min 51.111 s	1 min 29.701 s	0 min 0.218 s

Table 11. Verification times for the deadlock property with fault signals.

System	Standard	Function-based	Function-based with input reductions
1	30 min 43.590 s	0 min 3.539 s	0 min 0.344 s
2	> 32 min 54.421 s *	0 min 4.686 s	0 min 0.513 s
3	1 min 8.028 s	10 min 1.501 s	0 min 0.308 s
4	12 min 30.890 s	> 14 min 40.493 s *	0 min 14.298 s

* Out of memory

It would appear that version 4.1.4 of UPPAAL provides faster verification times than version 4.0.11. Version 4.1.4 even finished certain verification tasks for which version 4.0.11 ran out of memory. This behaviour may be caused by the 64-bit version of UPPAAL 4.1.4 using more memory than UPPAAL 4.0.11 and being able to use all of the 8 GB of RAM available.

5.6.2 Achieved input reductions

We now summarize by how much the number of input combinations for the first time-dependent components was reduced with the Java program for each system.

System 1: The number of combinations of regular inputs was reduced from 16 to 8, while the number of combinations of fault inputs was reduced from 16 to 6. In total, the number of input combinations was reduced from 256 to 48.

System 2: The number of combinations of regular inputs was reduced from 256 to 41, while the number of combinations of fault inputs was reduced from 256 to 13. In total, the number of input combinations was reduced from 65536 to 533.

Systems 3 and 4: All combinations of inputs are possible, for both regular and fault inputs.

Input reductions were clearly much more effective for systems 1 and 2 than systems 3 and 4 because of structural differences between these systems. In fact, determining input reductions did not yield any impossible input combinations.

However, there are only 2 inputs for the first time-dependent components in system 3 and 4 such inputs in system 4, while the entire system 3 has 18 inputs and the entire system 4 has 13 inputs. Therefore, in practice the number of inputs for systems 3 and 4 is still reduced with the technique and a large part of the system is cut in the process, so the technique results in faster verification times. Even so, this could have been achieved without actually determining input reductions by removing the initial time-independent section.

System 1 is a smaller version of system 2, and system 3 is a smaller version of system 4. However, the basic structure of the time-independent section is relative-

ly similar in these systems of different scales. The proportional decrease in the number of input combinations is significantly greater for system 2 than system 1.

5.7 Summary

5.7.1 Efficiency of the analysis

According to the results, verification with UPPAAL is fastest with the examined systems using the function-based asynchronous modelling technique with input reductions. This approach reduces the number of possible output combinations of the initial time-independent section and as a result, UPPAAL has fewer combinations to go through. This technique also removes the time-independent section, reducing the model size. The function-based technique with input reductions is at its best when the initial time-independent section of the system is large or a large portion of the combinations of outputs can be reduced by the program as infeasible.

Depending on the system, verification may be faster with the function-based asynchronous modelling technique than the standard asynchronous modelling technique or vice versa. If changes in the values of the inputs affect the outputs of the initial time-independent section most of the time, the function-based technique is more efficient since calculating with functions is faster than with separate components. This is the result of the model size being smaller with the function-based technique than with the standard technique. Nevertheless, the standard technique is faster if a sufficient number of the possible changes in inputs do not influence the outputs of the initial time-independent section, resulting in unnecessary calculations.

UPPAAL is not an efficient tool for modelling large systems, particularly with a large number of input signals. Therefore, having fault signals also makes modelling much more demanding. However, with the function-based modelling techniques, it is more plausible to verify properties of large systems, even if fault variables are used. Using input reductions can be very beneficial namely because the efficiency of verifying UPPAAL models is highly dependent on the number of inputs.

It should also be noted that this study only examined verification speed with a limited number of systems. In the future, examining a wider array of systems can reveal more information about which modelling technique is most suitable for each system.

5.7.2 Simplicity of the modelling

Modelling with the function-based asynchronous modelling technique is more difficult than with the standard asynchronous technique. With the standard technique, the user only needs to add UPPAAL components to the system declara-

tions page and variables to the declarations page. However, with the function-based technique, changes are also required in the functions and parameters of the components themselves, particularly the input component. Even if the modifications of the components mostly contain the same logical functions as the modifications of the system declarations page, it still makes the modelling process more difficult. This approach decreases the modularity of the components and makes them system-dependent.

It is also possible to replace time-independent components in later parts of the system with functions. This can be achieved similarly to that of input components, with the addition of synchronization channels from the time-dependent components that give the input values. However, each synchronization channel for an input requires its own edge in the UPPAAL model, which may make modelling tedious.

Modelling with the function-based asynchronous technique with input reductions is even more demanding. The user must first construct the time-independent section of the system with the external Java program, have it calculate possible combinations of outputs and then modify the input component of the UPPAAL model. In particular, changes in the Java code of the program require the construction of the BDDs describing the time-independent section.

6. Conclusions

This report presents the model checking results of the SARANA project in 2011. The report covers an algorithm for the model checking of large systems, methodology for fault models and methods for model checking function-block-based designs in UPPAAL.

A model checking algorithm for large models was introduced. The algorithm can be used with modular models in which an abstraction of the model can be created by replacing some of the modules with interface modules. So far, the algorithm has been developed and tested using a model of a function-block-based design. The algorithm is largely based on counterexample-guided abstraction refinement, in which an abstraction of the system is examined and the abstraction iteratively refined based on the responses of the model checking tool. The algorithm puts significant effort into counterexample minimization. We present three counterexample minimization techniques that can be used on several granularity levels. The algorithm has not yet been extensively compared with any standard model checking methods. In 2011, the performance of the algorithm was tested on a model based on work in [Lahtinen et al. 2010]. These preliminary results suggest that in some cases the algorithm can be more effective than using traditional model checking methods. A more thorough analysis of the performance of the algorithm is left to future research. We plan to analyse the effectiveness of the algorithm using various models and many temporal properties. Some improvements and extensions of the methodology are also planned (see Section 3.9).

In Section 4 we presented new methodology to model faults in a system. The fault models take into account the hardware configuration of a system and the various failure modes of the different hardware components. In addition, common-cause failure modes can be included in the fault models. We created a way to integrate fault models into models depicting the software logic of a system. However, when a detailed fault model is used together with the model of the logic of a system, the model checking task becomes quite complex. This suggests that it could be possible to use fault models modularly together with our traditional methods, so that hardware faults of a complex system could also be analysed using the algorithm for large systems. This work is left to future research. To test the fault

modelling methodology, an imaginary system was modelled as a case study. We also managed to apply assume-guarantee reasoning to decrease the verification time in that model.

Section 5 describes methodology for modelling function-block-based designs asynchronously using timed automata of the UPPAAL model checking tool. Three modelling techniques are presented. In the standard technique, a timed automaton is created for each function block and input of the system. The function-based technique uses functions to replace the time-independent parts of the model. In the third technique, the possible inputs of the time-independent part of the model are calculated separately using a Java program. The inputs are then used to create more efficient functions in the UPPAAL model. The modelling methods have been tested using three separate case studies. While the work is still partly underway, the results thus far show that the function-based modelling technique and, especially, the input reductions can make the model checking of function-block-based systems more feasible. UPPAAL is known to behave badly when there are a large number of inputs in the system because UPPAAL explicitly checks each input combination. The input reductions counter this weakness of the UPPAAL tool and the use of input reductions can lead to major improvements in verification time. However, using the input reductions is not straightforward and requires more modeller expertise.

References

Alur, R. & Dill, D. L. 1994. ‘A theory of timed automata’. Theoretical Computer Science, 126(2), 1994, pp. 183–235.

Alur, R., Courcoubetis, C. & Dill, D. 1990. ‘Model-checking for real-time systems’. In: Proceedings, Fifth Annual IEEE Symposium on Logic in Computer Science, 1990, pp. 414–425.

Ball, T., Mayur, N. & Rajamani, S.K. 2003. ‘From Symptom to Cause: Localizing Errors in Counterexample Traces’, POPL’03, January 15–17, 2003, New Orleans, Louisiana, USA.

Behrmann, G., David, A. & Larsen, K. G. 2004. A Tutorial on Uppaal. Formal Methods for the Design of Real-Time Systems. Springer Berlin / Heidelberg, 2004.

Biere, A., Cimatti, A., Clarke, E. M. & Zhu, Y. 1999. ‘Symbolic model checking without BDDs’. In: Proc. of the Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS’99).

Biere, A., Heljanko, K., Junttila, T., Latvala, T. & Schuppan, V. 2006. ‘Linear Encodings of Bounded LTL Model Checking’. Logical Methods in Computer Science 2(5:5), pp. 1–64.

Bryant, R. E. 1986. ‘Graph-Based Algorithms for Boolean Function Manipulation’. IEEE Trans. Computers 35(8), pp. 677–691.

Cavada, R., Cimatti, A., Jochim, C. A., Keighren, G., Olivetti, E., Pistore, M., Roveri, M. & Tchaltsev, A. 2010. ‘NuSMV 2.5 User Manual’. FBK-irst.

Chang, K-H., Bertacco, V. & Markov, I. L. 2007. ‘Simulation-Based Bug Trace Minimization with BMC-Based Refinement’. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 26, No. 1, 2007.

Clarke, E. M., Grumberg, O. & Peled, D. A. 1999. Model checking. Cambridge MA: MIT Press, 1999. 314. ISBN 0-262-03270-8.

Clarke, E. M. & Emerson, E. A. 1981. ‘Design and synthesis of synchronization of skeletons using branching time temporal logic’. In: Proceedings of the

IBM Workshop on Logics of Programs, Vol. 131 of LNCS, Springer, pp. 52–71.

Clarke, E. M., Gupta, A. & Strichman, O. 2004. 'SAT-Based Counterexample-Guided Abstraction Refinement'. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 23, No. 7, July 2004.

Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S. & Xiao, C. 2007. The Daikon system for dynamic detection of likely invariants Science of Computer Programming 2007, Vol. 69, No. 1–3, pp. 35-45. ISSN 01676423. Doi: 10.1016/j.scico.2007.01.015

Gastin, P., Moro, P. & Zeitoun, M. 2004. 'Minimization of counterexamples in SPIN'. In: SPIN Workshop on Model Checking of Software, pp. 92–108, 2004.

Groce, A. & Visser, W. 2003. 'What went wrong: Explaining Counterexamples'. In: SPIN Workshop on Model Checking of Software, pp. 121–135, 2003.

Jin, H., Ravi, K. & Somenzi, F. 2002. 'Fate and Free Will in Error Traces', Katoen J. P. & Stevens, P. (Eds.): TACAS 2002, LNCS 2280, pp. 445–459, 2002.

Lahtinen, J., Björkman, K., Valkonen, J., Frits, J. & Niemelä, I. 2010. 'Analysis of an emergency diesel generator control system by compositional model checking'. MODSAFE 2010 work report. VTT Working Papers 156. 2010. http://www.vtt.fi/inf/pdf/workingpapers/2010/W156.pdf

Larsen, K. G., Pettersson, P. & Yi, W. 1997. 'UPPAAL in a nutshell'. International Journal on Software Tools for Technology Transfer, 1(1–2), 1997, pp. 134–152.

McMillan, K. L. 1993. 'Symbolic Model Checking', Kluwer Academic Publ.

NuSMV 2011. NuSMV Model Checker v.2.5.2, 2011. http://nusmv.irst.itc.it/

Quielle, J. & Sifakis, J. 1981. 'Specification and verification of concurrent systems in CESAR'. In: Proceedings of the 5th International Symposium on Programming, pp. 337–350.

Ravi, K. & Somenzi, F. 2004. 'Minimal Assignments for Bounded Model Checking', Jensen, K. & Podelski, A. (Eds.): TACAS 2004, LNCS 2988, pp. 31–45, 2004.

Roorda, J-W. & Claessen, K. 2006. 'SAT-Based Assistance in Abstraction Refinement for Symbolic Trajectory Evaluation', Ball, T. & Jones, R. B. (Eds.): CAV 2006, LNCS 4144, pp. 175–189, 2006.

Ropponen, J. 2010. Modular modelling with timed automata. Aalto University.

Shen, S., Qin, Y. & Li, S. 2005. 'A Fast Counterexample Minimization Approach with Refutation Analysis and Incremental SAT'. In: Proc. ASP-DAC 2005, pp. 451–454.

Seger, C. H. & Bryant, R. E. 1995. Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories, Formal Methods in System Design 1995, Vol. 6, pp. 147–190.

Uppaal. 2009. UPPAAL integrated tool environment v. 4.0.6, 2009. http://www.uppaal.com/

Valkonen, J., Karanta, I., Koskimies, M., Heljanko, K., Niemelä, I., Sheridan, D. & Bloomfield, R. E. 2008. 'NPP Safety Automation Systems Analysis – State of the Art'. VTT Working Papers 94, VTT, Espoo. 62 p. http://www.vtt.fi/inf/pdf/workingpapers/2008/W94.pdf

Weiser, M. 1981. Program slicing. ICSE '81 Proceedings of the 5th International Conference on Software Engineering. NJ, USA: IEEE Press Piscataway, 1981, pp. 439–449. ISBN 0-89791-146-6.

Zeller, A. 2002. 'Isolating Cause-Effect Chains from Computer Programs'. In: SIGSOFT 2002/FSE-10, November 18–22, 2002, Charleston, SC, USA.

Zeller, A. & Hildebrandt, R. 2002. 'Simplifying and Isolating Failure-Inducing Input'. In: IEEE Transactions on Software Engineering, Vol. 28, No. 2, February 2002.

Series title and number
VTT Technology 12

Title	**Model checking methodology for large systems, faults and asyn-chronous behaviour** **SARANA 2011 work report**
Author(s)	Jussi Lahtinen, Tuomas Launiainen, Keijo Heljanko & Jonatan Ropponen
Abstract	Digital instrumentation and control (I&C) systems are challenging to verify. They enable complicated control functions, and the state spaces of the models easily become too large for comprehensive verification through traditional methods. Model checking is a formal method that can be used for system verification. A number of efficient model checking systems are available that provide analysis tools to determine automatically whether a given state machine model satisfies the desired safety properties. This report reviews the work performed in the Safety Evaluation and Reliability Analysis of Nuclear Automation (SARANA) project in 2011 regarding model checking. We have developed new, more exact modelling methods that are able to capture the behaviour of a system more realistically. In particular, we have developed more detailed fault models depicting the hardware configuration of a system, and methodology to model function-block-based systems asynchronously. In order to improve the usability of our model checking methods, we have developed an algorithm for model checking large modular systems. The algorithm can be used to verify properties of a model that could otherwise not be verified in a straightforward manner.
ISBN, ISSN	ISBN 978-951-38-7625-8 (URL: http://www.vtt.fi/publications/index.jsp) ISSN 2242-122X (URL: http://www.vtt.fi/publications/index.jsp)
Date	March 2012
Language	English
Pages	84 p.
Name of the project	Safety Evaluation and Reliability Analysis of Nuclear Automation
Commissioned by	
Keywords	Model checking, verification, I&C, NuSMV, UPPAAL, SARANA, SAFIR
Publisher	VTT Technical Research Centre of Finland P.O. Box 1000, FI-02044 VTT, Finland, Tel. 020 722 111

VTT TECHNOLOGY 12

Model checking methodology for large systems, faults and...

ISBN 978-951-38-7625-8 (URL: http://www.vtt.fi/publications/index.jsp)
ISSN 2242-122X (URL: http://www.vtt.fi/publications/index.jsp)