

# Model Driven Development of Data Sensitive Systems

Olsen, Petur

*Publication date:*
2014

*Document Version*
Peer reviewed version

# Model Driven Development of Data Sensitive Systems

Petur Olsen

Ph.D. Dissertation

May 12, 2014

*Aalborg University*
*Department of Computer Science*

**Title:**
Model Driven Development of Data Sensitive Systems

**PhD Student:**
Petur Olsen

**Supervisors:**
Associate Professor Arne Skou
Professor Kim G. Larsen

**Paper A:**

| | |
|---|---|
| **Authors:** | Petur Olsen, Kim G. Larsen, and Arne Skou |
| **Title:** | Present and Absent Sets: Abstraction for Testing of Reactive Systems with Databases |
| **Outlet:** | MBT 2010. ENTCS, Volume 264 Issue 3, December 2010, pages 53-68 |

**Paper B:**

| | |
|---|---|
| **Authors:** | Andreas Engelbredt Dalsgaard, René Rydhof Hansen, Kim Gulstrand Larsen, Mads Chr. Olesen, Petur Olsen, and Jiří Srba |
| **Title:** | Model Checking with Lattices |
| **Outlet:** | Unpublished |

**Paper C:**

| | |
|---|---|
| **Authors:** | Andreas Engelbredt Dalsgaard, René Rydhof Hansen, Kenneth Yrke Jørgensen, Kim Gulstrand Larsen, Mads Chr. Olesen, Petur Olsen, and Jiří Srba |
| **Title:** | opaal: A Lattice Model Checker |
| **Outlet:** | NFM 2011, Pasadena, CA, USA, April 18-20, 2011. LLNCS, Volume 6617, pages 487-493 |

**Paper D:**

| | |
|---|---|
| **Authors:** | Fides Aarts, Faranak Heidarian, Petur Olsen, and Frits Vaandrager |
| **Title:** | Automata Learning Through Counterexample-Guided Abstraction Refinement |
| **Outlet:** | Unpublished |

**Paper E:**

| | |
|---|---|
| **Authors:** | Petur Olsen, Johan Foederer, and Jan Tretmans |
| **Title:** | Model-Based Testing of Industrial Transformational Systems |
| **Outlet:** | ICTSS 2011, Paris, France, November 7-9, 2011, Volume 7019, pages 131-145 |

# Abstract

Model-driven development strives to use formal artifacts during the development process. Formal artifacts enables automatic analyses of some aspects of the system under development. This serves to increase the understanding of the (intended) behavior of the system as well as increasing error detection and pushing error detection to earlier stages of development.

The complexity of modeling and the size of systems which can be analyzed is severely limited when introducing data variables. The state space grows exponentially in the number of variable and the domain size of the variables. This quickly leads to state-space explosion problems and usually results in data being abstracted away in the models. This works great for systems where the particular values of the variables do not significantly alter the execution of the system. Examples of this type of system are transport protocols or pure storage systems, where the actual values of the data is not relevant for the behavior of the system. For many systems the values are important. For instance the control flow of the system can be dependent on the input values. We call this type of system *data sensitive*, as the execution is sensitive to the values of variables.

This theses strives to improve model-driven development of such data-sensitive systems. This is done by addressing three research questions. In the first we combine state-based modeling and abstract interpretation, in order to ease modeling of data-sensitive systems, while allowing efficient model-checking and model-based testing. In the second we develop automatic abstraction learning used together with model learning, in order to allow fully automatic learning of data-sensitive systems to allow learning of larger systems. In the third we develop an approach for modeling and model-based testing of stateless systems with very large input and output domains.

**Keywords:** Model-based development, model-based testing, data-sensitive systems, model learning.

# Resumé

Modeldrevet udvikling stræber efter at bruge formelle artefakter under udviklingsprocessen. Formelle artefakter tillader automatisk analyse af nogle aspekter af systemet under udvikling. Dette øger forståelsen af (den tilsigtede) opførsel af systemet, og forbedrer fejlfinding og skubber fejlfinding til tidligere faser af udviklingen.

Kompleksiteten af modelleringen og størrelsen af systemer der kan analyseres bliver svært begrænset ved introduktion af data variable. Tilstandsrummet vokser eksponentielt i antallet af variable og i størrelsen af deres domæner. Dette leder hurtigt til for store tilstandsrum og gør ofte at data bliver abstraheret væk i modellerne. Dette virker godt for systemer hvor bestemte værdier ikke ændrer opførslen af systemet signifikant. Eksempler på denne type systemer er transport protokoller og systemer der kun gemmer værdier, hvor de faktiske værdier ikke er relevant for systemets opførsel. For mange systemer er værdierne vigtige. For eksempel kan systemets eksekverings sti ændre sig afhængigt af input værdier. Vi kalder denne type af systemer *data følsomme*, eftersom deres opførsel er følsom for variables værdier.

Denne afhandling forsøger at forbedre modeldrevet udvikling af datafølsomme systemer. Dette gøres ved at gribe tre forsknings spørgsmål an. I det første kombinerer vi modeldrevet udvikling og abstrakt fortolkning for at gøre det lettere at modellere datafølsomme systemer, og samtidig tillade effektiv verifikation og modelbaseret test. I det andet udvikler vi automatisk abstraktions læring som bliver brugt sammen med model læring, for at muliggøre fuldautomatisk model læring af datafølsomme systemer og muliggør læring af større systemer. I det tredje udvikler vi en fremgangsmåde for modellering og modelbaseret test af tilstandsløse system med meget store input of output domæner.

**Nøgleord:** Modelbaseret udvikling, modelbaseret testing, datafølsomme systemer, model læring.

# Contents

# Chapter 1

# Model-Driven Development

Software is usually developed in phases. The prevalent phases identified in the traditional waterfall model, as seen in Figure 1.1, are [87]: Analysis, Design, Implementation, Test, and Operation.

Most system development starts from some ideas. These stem from the (end) user which identifies some system or area which needs improving and often only has a vague description of wanted change. The development of the system is done in the five phases.

1. The problem area and desired improvements are analyzed to gain knowledge about the domain and to specify requirements. In larger projects, this analysis traditionally results in a requirements document, detailing the requirements of the system.

2. The requirements are used to develop a design of the system. Traditionally the design can consist of various architectures, design patterns, class diagrams, etc.

3. The design is used to guide an implementation of the system.

4. The implementation is tested. This can be anything from white-box unit testing to black-box integration testing.

5. The system is then put into operation and possibly monitored for operational status.

The waterfall model states that these phases are carried out in progression. More iterative (and realistic) approaches often specify that parts of the system are analyzed, designed, implemented and tested, before moving on to other parts of the system. Though the life-cycles differ, the main activities are usually the same five.

The assets produced during the development are largely informal. We call them informal since they have no formal semantics. The lack of formal semantics generally has two consequences. Firstly, the requirements can be interpreted

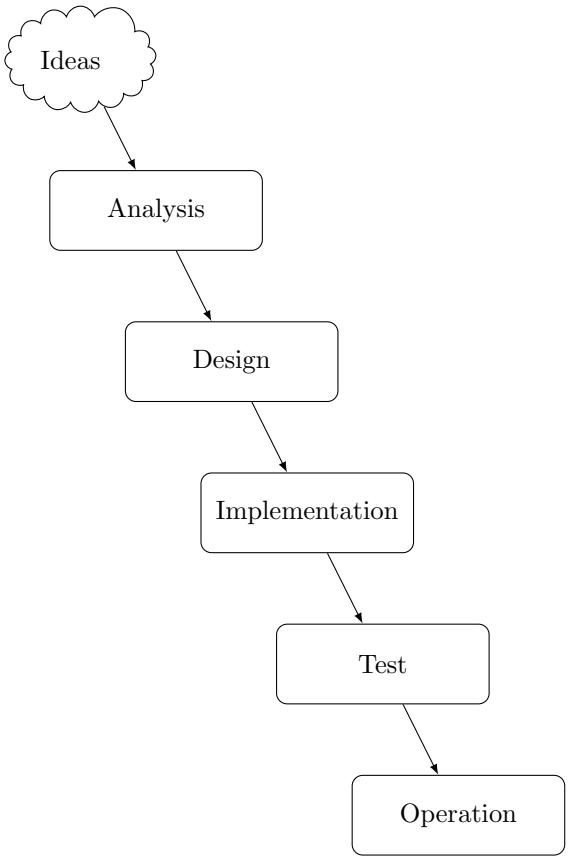


Figure 1.1: Waterfall development process

in different ways. For instance, two developers reading the same informal requirements might understand them differently, or one developer explaining a requirement to another might not get his idea across as intended. This discrepancy in understanding of the requirements makes it unlikely that the delivered system behaves as intended.

Second, the requirements can be inconsistent, since no automatic analysis can be performed. Conflicting requirements could for instance be, requiring that a coffee machine always outputs coffee within 10 seconds when a button is pressed, and also requiring that the coffee machine should be able to produce tea on the same button. For this simple example it is easy to spot a conflict, but with a large requirements document describing complex systems, it is difficult to guarantee that no inconsistencies are present.

Model-Driven Development (MDD) focuses on formal requirements and models as artifacts in the development process in order to bridge the gap between informal requirements and implemented system. Figure 1.2 gives an overview

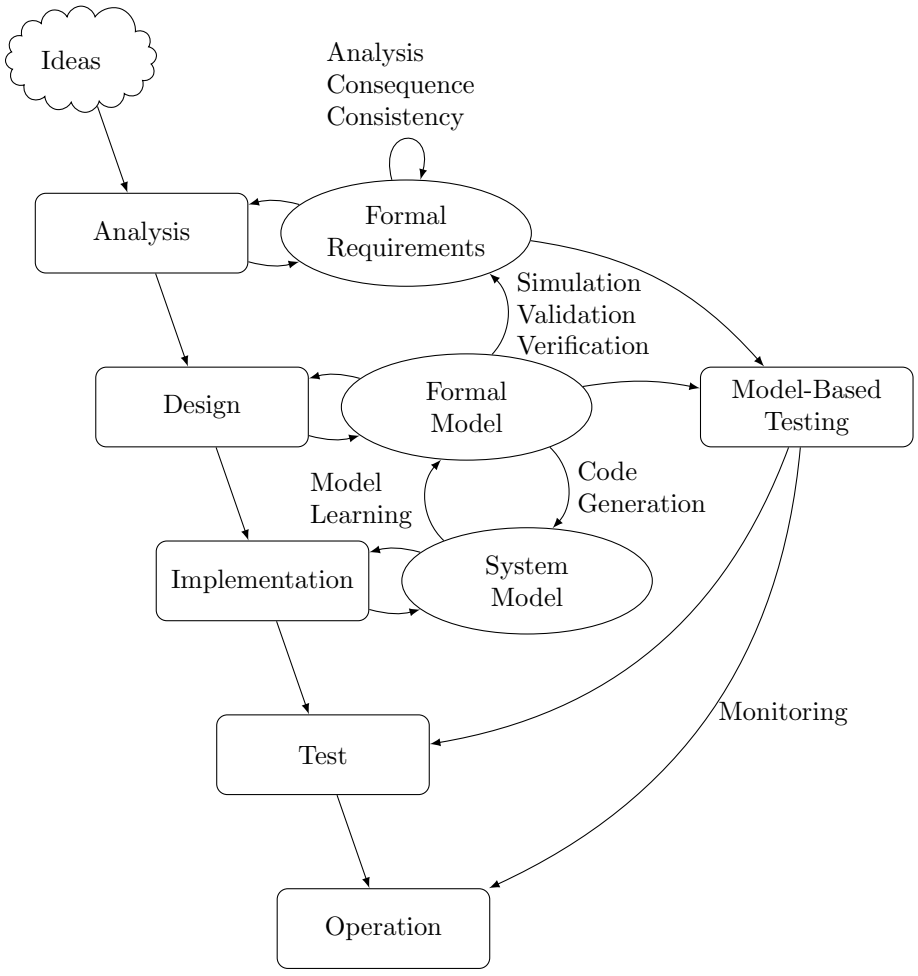


Figure 1.2: Overview of MDD

of MDD in the context of the waterfall model. MDD has been successfully used with other development processes [50], but for the sake of simplicity we will describe it in the context of the waterfall model here.

The analysis phase focuses on developing a set of formal requirements for the system. The requirements are often a set of properties that must hold for the system. These describe *what* the system must do. Since these requirements are formal, they can be automatically analyzed with respect to consistency and the consequences of these requirements can be analyzed.

The design phase focuses on formal behavioral models of the system, often in the form of state machines. These models describe *what* the system should do, how it should react to events, etc. Again, since the models are formal, automatic analysis can be performed. The model can be held up against the

requirements with simulation, validation, and verification. These analyses give good indications of the correctness of the model, and can state if the model adheres to the requirements.

The implementation is done as usual, with inputs from the formal requirements and models. Having formally verified requirements and models, reduces the chance of the requirements changing during the implementation phase, as any inconsistencies would have been caught.

As seen on Figure 1.2, two more activities can be involved in the implementation. The formal model can be used to generate code for the implementation. This has been used for complete controller synthesis (as done by UPPAAL TIGA [18]) and for generating the scaffolding for the implementation (as done by such tools as Rational Rhapsody). In the case of legacy systems or third party software, it is possible to learn a model of the implementation [15]. This learned model can be composed with existing models to check for interoperability with systems where no models are available.

During testing, the formal model can be used to automatically generate test cases with an approach called Model-Based Testing (MBT) [97]. Test cases are generated that cover the model. This coverage can be done using different metrics, e.g. state coverage or transition coverage. The test cases can be executed on the system under test (SUT) and the output can be verified using the model. This can give a good indication as to whether the SUT implements the model correctly and thereby adheres to the requirements.

Since testing can be seen as environment emulation combined with implementation monitoring, the same techniques can be used for monitoring during operation [53]. Operation monitoring can be done using the monitoring and output verification techniques from MBT. No test cases are generated, but the system is monitored. The inputs and outputs are checked with respect to the model. This can be used to ensure the system is operating as intended, and to monitor the actual environment the system is running under. This can also be done a posteriori by collecting data during operation and analyzing them later.

In recent year the industry has started adopting MDD. Several research projects, such as MBAT[1], strive to mature MDD and MBT for use in the industry. MBAT includes several industrial partners, such as Daimler (project coordinator), Airbus, Siemens, and Volvo.

This thesis deals with modeling both the requirements and the behavior and how to use these together with model checking and model-based testing. The following chapters explain these aspects in greater detail.

[1] http://www.mbat-artemis.eu/home/

# Chapter 2

# Requirements

Capturing requirements is about specifying *what* the system should do; what properties it should adhere to. Good requirements generally have the following ten characteristics [38]: Unitary, Complete, Consistent, Non-Conjugated (Atomic), Traceable, Current, Feasible, Unambiguous, Specify Importance, and Verifiable. Building requirements with these characteristics is difficult, and much work and research has gone into improving this process and reducing errors in the requirements. In the formal approach, requirements are written in formal languages for which some properties can be verified.

Since the formal languages have unambiguous semantics, the requirements written in them are also unambiguous. Formal semantics also allow automatic or semi-automatic consistency checks.

The following sections describe some formalisms for capturing requirements.

## 2.1 Temporal Logics

For reactive systems the correctness of a system can not simply be stated as a function from input to output, since the systems continuously react to inputs and produce outputs. The correctness of reactive systems is often expressed as properties of the traces the system can produce. The traces can either focus on the states visited during the run (state based) or focus on the transitions taken (action based). Traces can also record both states and actions, but for complexity reasons, usually only one is tracked.

Temporal logics can be used to express behavior, in terms of traces or trees. An often desired property of reactive systems is deadlock freedom. A deadlock is a state in which no progress is possible. The classic example of a deadlock is a set of processes circularly waiting for each other. Process $p_1$ holds resource $a$ and waits for resource $b$ to be available, process $p_2$ holds resource $b$ and waits for resource $a$ to become available. The processes can never progress beyond this point; we have a deadlock.

The property stating that deadlocks do not occur is called a *safety* property;

intuitively it states that something bad never happens. To ensure progress in the system a different kind of property is used, called *liveness* property, or sometimes *progress* property. Informally, a liveness property states that something good will happen. Other types of properties exist, and liveness and safety properties can be further divided into subtypes. For instance a liveness property can express [16, p. 121]

- eventuality (eventually the machine will produce coffee),
- repeated eventuality (the machine will produce coffee infinitely often), and
- starvation freedom (if the machine is asked, then it will eventually produce coffee).

To verify such properties we need formal languages to specify the desire property. Popular languages include *Linear Temporal Logic* (LTL) [80] and *Computational Tree Logic* (CTL) [31].

### 2.1.1 Linear Temporal Logic

LTL is a temporal logic expressing properties of traces or runs of systems. It extends propositional or predicate logic with modalities to refer to infinite behavior. LTL is called linear because time is viewed as path based and progresses in a linear fashion, i.e. each step has a single successor.

The basic LTL-formulas are formed over atomic propositions, Boolean conjunction $\wedge$, Boolean negation $\neg$, and the two basic temporal modalities $\bigcirc$ (next) and $\mathbf{U}$ (until). The atomic propositions ($a \in AP$) are state-labels in a transition systems. These are often used to express assertions on the model, e.g. $x < 5$ or that the discrete location in the model is $y$. The next-modality states properties about the next step, i.e. $\bigcirc\varphi$ holds if $\varphi$ holds in the next step. The until-modality holds if one property holds until another holds, i.e. $\varphi_1 \mathbf{U} \varphi_2$ holds in the current step if $\varphi_1$ holds in this step and all future steps until at some point $\varphi_2$ holds. LTL-formulas are formed according to the following grammar:

$$\varphi ::= true \mid a \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \mid \bigcirc \varphi \mid \varphi_1 \mathbf{U} \varphi_2 \mid (\varphi)$$

where $a \in AP$.

Using this basic grammar other Boolean connectives such as disjunction $\vee$ and implication $\rightarrow$ can be derived. The modalities present in most temporal logics are

- $\Diamond$ *eventually*, or $\mathbf{F}$ for *Future* (derived from $true \mathbf{U} \varphi$) and
- $\Box$ *always*, or $\mathbf{G}$ for *Globally* (derived from $\neg\Diamond\neg\varphi$).

These can be derived using the modalities present in the basic grammar. Deriving more complex modalities and Boolean connectives allows more complex formulas to be expressed more succinctly.

Modalities can be combined to form new modalities. For instance, $\Diamond\Box a$ (eventually forever $a$) describes a path where at some moment $a$ will hold and continue to hold forever.

### Semantics of LTL

An LTL formula describes properties of traces or runs of systems. This means a trace can either satisfy an LTL formula or not. A trace can be expressed as a word of (a set of) atomic propositions observed along the trace. That is, an infinite sequence of elements from $2^{AP}$. (This assumes all traces are infinite. This is generic since any transition system with finite traces can be transformed into one with only infinite trace, by adding transitions from terminal states to a sink state, satisfying the empty set of atomic propositions and with itself as sole successor. The semantics can also be lifted to support finite traces.) The semantics of an LTL formula $\varphi$ can now be defined as a language $Words(\varphi)$ containing all infinite words over the alphabet $2^{AP}$ that satisfy $\varphi$ [16].

$$Words(\varphi) = \{\sigma \in (2^{AP})^{\omega} \mid \sigma \models \varphi\}$$

Here the satisfaction relation $\models \subseteq (2^{AP})^{\omega} \times$ LTL is the smallest relation with the following properties [16]:

$\sigma \models true$
$\sigma \models a$ iff $a \in \sigma[0]$
$\sigma \models \varphi_1 \wedge \varphi_2$ iff $\sigma \models \varphi_1$ and $\sigma \models \varphi_2$
$\sigma \models \neg\varphi$ iff $\sigma \not\models \varphi$
$\sigma \models \bigcirc\varphi$ iff $\sigma[1 \ldots] \models \varphi$
$\sigma \models \varphi_1 \mathbf{U} \varphi_2$ iff $\exists j \geq 0. \sigma[j \ldots] \models \varphi_2$ and $\sigma[i \ldots] \models \varphi_1,$ forall $0 \leq i < j$

Where, for $\sigma = A_0 A_1 A_2 \ldots \in (2^{AP})^{\omega}$, $\sigma[j] = A_j$ and $\sigma[j \ldots] = A_j A_{j+1} A_{j+2} \ldots$. The semantics for the derived operators can be derived from these semantics.

### Examples

The no-deadlocks property mentioned can be expressed in LTL if the *deadlock* proposition is defined as an atomic proposition that holds for a state when it has no successors. The property can then be expressed as:

$$\Box\neg deadlock$$

Mutual exclusion can be expressed using propositions $crit_1$ and $crit_2$, stating that processes $p_1$ and $p_2$ are in the critical section, respectively. The following formula states that it is always the case that one process is not in the critical section:

$$\Box(\neg crit_1 \vee \neg crit_2)$$

A deadlock can be seen as a non-desirable property of reactive systems. Conversely, termination can be seen as a desirable property of non-reactive systems. The property that a system eventually terminates can be expressed as follows:

$$\Diamond terminate$$

The following specifies that a request will always lead to a grant:

$$\Box(request \rightarrow \Diamond grant)$$

The three liveness properties mentioned above can be expressed as:

- $\Diamond\varphi$, eventually $\varphi$
- $\Box\Diamond\varphi$, always eventually $\varphi$
- $\Box(\varphi_1 \rightarrow \Diamond\varphi_2)$, if $\varphi_1$ then eventually $\varphi_2$

As mentioned LTL views time in a linear fashion, and each state only has one successor. But transition systems often have branches, and possible executions often fold out into a tree structure. For an LTL formula to hold in a state it is implicitly specified that it must hold for all possible executions from that state. This makes it impossible to specify properties in LTL which should only hold on some execution paths. CTL can be used for these properties.

### 2.1.2 Computational Tree Logic

In contrast to LTL, CTL is a branching-time logic, and sees the entire tree of possible runs through the system. A branching-time logic uses path quantifiers to express which paths a property holds for. The quantifiers are the existential quantifier $\exists$ and the universal quantifier $\forall$. As an example, $\exists\Diamond\varphi$ expresses that there exists a path on which there is a state where $\varphi$ holds, and $\forall\Box\varphi$ expresses that on all paths $\varphi$ holds in all states.

The syntax of CTL is separated into path formulas and state formulas, where path formulas express properties that hold for entire paths (infinite sequence of states) and state formulas express properties that hold for a single state.

CTL *state formulas* over the set $AP$ of atomic proposition are formed according to the following grammar [16, p. 317]:

$$\Phi ::= \text{true} \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid (\Phi) \mid \exists\varphi \mid \forall\varphi$$

where $a \in AP$ and $\varphi$ is a path formula. CTL *path formulas* are formed according to the following grammar:

$$\varphi ::= \bigcirc\Phi \mid \Phi_1 \mathbf{U} \Phi_2$$

where $\Phi$, $\Phi_1$, and $\Phi_2$ are state formulas.

#### Semantics of CTL

The semantics of CTL formulas are defined by two satisfaction relations; one for state formulas, and one for path formulas.

Let $a \in AP$ be an atomic proposition, $TS = (S, Act, \rightarrow, I, AP, L)$ be a transition system without terminal states, state $s \in S$, $\Phi$, $\Psi$ be CTL state

formulas, and $\varphi$ be a CTL path formula. The satisfaction relation $\models$ is defined for state formulas by [16, p. 320]:

$$\begin{array}{lll} s \models a & \text{iff} & a \in L(s) \\ s \models \neg\Phi & \text{iff} & \text{not } s \models \Phi \\ s \models \Phi \wedge \Psi & \text{iff} & (s \models \Phi) \text{ and } (s \models \Psi) \\ s \models \exists\varphi & \text{iff} & \pi \models \varphi \text{ for some } \pi \in Paths(s) \\ s \models \forall\varphi & \text{iff} & \pi \models \varphi \text{ for all } \pi \in Paths(s) \end{array}$$

For path $\pi$, the satisfaction relation $\models$ for path formulas is defined by:

$$\begin{array}{lll} \pi \models \bigcirc\Phi & \text{iff} & \pi[1] \models \Phi \\ \pi \models \Phi\mathbf{U}\Psi & \text{iff} & \exists j \geq 0.(\pi[j] \models \Psi \wedge (\forall 0 \leq k < j.\pi[k] \models \Phi)) \end{array}$$

where for path $\pi = s_0 s_1 s_2 \ldots$ and integer $i \geq 0$, $\pi[i]$ denotes the $(i+1)$th state of $\pi$, i.e., $\pi[i] = s_i$.

LTL and CTL are incomparable. There are formulas that can be expressed in LTL but not CTL and vice versa.

**Examples**

Deadlock absence and Mutual exclusion can be described similarly to LTL:

$$\forall\Box\neg deadlock$$

and

$$\forall\Box(\neg crit_1 \vee \neg crit_2).$$

The following formula specifies that both $p_1$ and $p_2$ will enter the critical section infinitely often:

$$(\forall\Box\forall\Diamond crit_1) \wedge (\forall\Box\forall\Diamond crit_2)$$

## 2.2 Sequence Charts

An important aspect to model is the communication between systems, or subsystems. Communication between systems can be seen as a scenario of messages sent back and forth. Scenarios can represent behavior which is allowed or forbidden. These types of scenarios can be modeled using *scenario based modeling*. Modeling scenarios can be used to give the developers an overview of the communication channels in the system. It can also be used to specify use-scenarios of the system as a part of the requirements.

One of the original scenario based modeling formalisms, are the Message Sequence Charts (MSC) [14]. Figure 2.1 shows an example of a MSC. The MSC describes a set of processes, depicted as boxes along the top of the chart. The vertical line going down each process illustrates the *lifeline* of the process. A $m$-labeled arrow from process $p$ to process $q$, represents an allowed transfer of message $m$ from $p$ to $q$. The chart is read from top to bottom, so the order of messages in the system must follow that of the MSC.

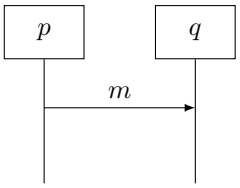


Figure 2.1: Example MSC

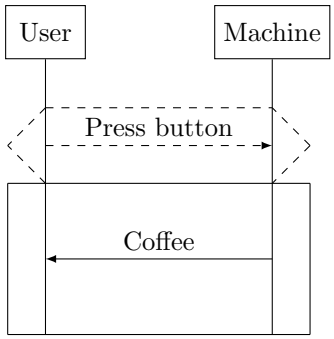


Figure 2.2: Example LSC

MSC have very limited power, since only allowed behavior can be expressed, it can not describe required or forbidden behavior. Live sequence charts are proposed as a more powerful replacement for MSC.

### 2.2.1 Live Sequence Charts

Live Sequence Charts (LSC) [36] can describe required behavior, as well as forbidden behavior. The LSC is divided into a *pre-chart* (which specifies the *premise*) and a *main chart* (which specifies the *conclusion*). If in any system run, the behavior in the pre-chart is observed, then the behavior in the main chart must follow immediately. Using LSC more behavior can be expressed.

LSC model scenarios and communication in a very intuitive and graphical way. LSC focus on statements of the form: *When event* A *occurs, perform action* B or *Perform action* A *then action* B *then require event* C *to occur*. Figure 2.2 shows an excerpt of the requirements for a coffee machine as a LSC. The dashed area represents the pre-chart and the square area represents the main chart. Uncontrollable actions are represented with dashed arrows and solid arrows represent controllable actions.

## 2.3 Constraint Systems

The previously mentioned formalisms are good for modeling requirements for reactive and state-based systems. However, some systems are not characterized

by a system state, but rather by the generation of output values, based on input values. Such systems are known as *transformational systems* or *constraint systems* [45] (or *functions* in mathematical terms).

A constraint system is a set of constraints on the output values, based on the input values. A constraint is a boolean formula, where the binary variables may be replaced with predicates over the values. This can be used to specify requirements for systems. For instance, the simple constraint: $x^2 = y$ states that the squared value of $x$ should equal the value of $y$. This type of requirement leaves the details to the implementation, while precisely expressing the desired behavior.

The example does not distinguish inputs and outputs. If $x$ is the input, the system is required to calculate the square of $x$ and output it in $y$. If $y$ is the input, the system is calculates the square root of $y$ and outputs it in $x$. However, if $y$ is negative, the system can not calculate the square root. The formula $\neg(y \geq 0) \vee x^2 = y$, ensures that the square root is only calculated when $y$ is non-negative. To make modeling this type of requirement less cumbersome, the implication $\rightarrow$ operator is used. Now the requirement can be expressed as $y \geq 0 \rightarrow x^2 = y$.

As a real world example, consider the controller inside a printer. Given a document in some standard format, such as PS or PDF, it translates the document into a printer specific format which can be printed directly on the hardware. Say the printer supports duplex printing, the number of printed pages can be verified by the formula

$$PageCount > 0 \wedge Printing = Duplex \rightarrow SheetCount = \lceil PageCount/2 \rceil$$

specifying that if the number of pages printed is greater than zero and the printing type is duplex, then the number of sheets printed must be half the number of pages, rounded up.

## 2.4 Property Specification Language

Property Specification Language (PSL) [1, 2] (formerly Sugar Formal Property Language) is an assertion language developed by Accellera. PSL is used to specify properties for simulation and verification of hardware designs. The language was initially developed by IBM, and has been chosen as standard by Accellera, and is now an IEEE 1850 standard.

A property can consist of four types of expressions composed to form the property [1]. *Boolean expressions* describe behavior over one clock cycle. *Sequential expressions* describe multi-cycle behavior. *Temporal expressions* describe relations over time between boolean expressions and sequences.

Lets consider an example (taken from [1]). Boolean expressions are formed using normal syntax, for example the formula

```
ena || enb
```

specifies a clock cycle in which at least one of the signals `ena` and `enb` are asserted. The sequential expression

```
{req; ack; !cancel}
```

describes a sequence where `req` is asserted in the first cycle, `ack` in the second, and `cancel` does not hold in the third. These are connected using temporal operators to get the property

```
always {req;ack;!cancel}(next[2] (ena || enb))
```

specifying that the sequence `{req;ack;!cancel}` is always followed by either `ena` or `enb` two cycles late. Adding the `assert` directive

```
assert always {req;ack;!cancel}(next[2] (ena || enb))
```

specifies that this property should hold for the design and needs to be verified.

Specifications written in the standard PSL language can be compiled to a formula in LTL. Specifications written using the PSL Optional Branching Extension can be compiled to a CTL formula.

## 2.5 Quantitative Requirements

So far we have dealt with properties for constraint systems and reactive systems, for both of which the correctness depends on the computations of the system. Some systems have other quantitative properties which define correct behavior. This type of system is called a quantitative system. The correctness of a quantitative system depends not only on the inputs and outputs, but also on the quantitative boundaries. The quantitative properties could for instance be time or price, or some other cost function. Desired properties of such systems specify boundaries on the quantitative properties of the system, e.g. *when the user presses the button, produce coffee within 5 seconds*, or *produce a plastic chair using less than 1.5 liters of oil.*

### 2.5.1 Timed Linear Temporal Logic

Several timing extension have been proposed to LTL, e.g. Metric Temporal Logic (MTL) [12], Timed Propositional Temporal Logic (TPTL) [13], Explicit-Clock Temporal Logic (XCTL) [49], and Metric Interval Timed Logic (MITL) [11].

Each uses different approaches to extending LTL with timing. For example MTL adds timing bounds to the next and until operators, and two new time bounded operators: since and previous. This approach has been shown to be undecidable in general. MITL uses a slightly different approach, where only a time bounded until operator is available, and the time bound can not be a singular interval, i.e. it can not specify a precise moment in time. MITL formulas are formed according to the syntax:

$$\varphi ::= a \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \mathbf{U}_I \varphi_2,$$

where $a \in AP$ is an atomic proposition and $I$ is a non-empty interval of the form: $[a,b], [a,b), [a,\infty), (a,b], (a,b), (a,\infty)$, where $a \leq b$ for $a, b, \in \mathbb{R}_+$. The case where $I$ is singular is not allowed; where $I = [a,a]$.

Using MITL (with the standard derived operators) it is possible to specify properties such as, *every press will be followed by coffee within 5 seconds.*

$$\Box_{[0,\infty)}(Press \rightarrow \Diamond_{[0,5)} Coffee).$$

However it is not possible to specify that coffee will follow a press after exactly 5 seconds.

$$\Box_{[0,\infty)}(Press \rightarrow \Diamond_{[5,5]} Coffee).$$

### 2.5.2 Timed Computation Tree Logic

To specify timing requirements CTL has been extended with time into timed computation tree logic (TCTL) [16]. TCTL allows timing boundaries by adding a time interval to the until operator. This can be used to extend the modal operators with time intervals as well.

The syntax of TCTL is again separated into state formulas and path formulas. The state formulas $\Phi$ are the same as CTL with the addition of atomic clock constraints. The path formulas $\varphi$ are changed to only contain the until operator, which is extended with a time interval.

$$\Phi ::= \text{true} \mid a \mid g \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid (\Phi) \mid \exists\varphi \mid \forall\varphi$$

$$\varphi ::= \Phi_1 \mathbf{U}^J \Phi_2$$

where $a \in AP$, $g \in ACC(C)$ is the set of atomic clock constraints over the set of clocks $C$, and $J \subseteq \mathbb{R}_{\geq 0}$ is an interval whose bounds are natural numbers.

The coffee machine property mentioned above can be expressed with the formula

$$\forall\Box(Press \rightarrow \forall\Diamond^{[0,5)} Coffee).$$

Uppaal [19, 22], a modern timed automaton model checker uses a simplified version of TCTL. The Uppaal query language does not support nested path formulas and has slightly different syntax with some added syntactic sugar. The universal and existential quantifiers are written `A` and `E`, respectively, and the always and eventually are written `[]` and `<>`, respectively. To express the formula above we add a clock `x` which is reset when the user presses the button. The formula can be expressed as

```
A[] (Press imply A<>(Coffee and x<5)).
```

However, this makes use of nested path formulas and is not possible. For this type of query Uppaal has a *leads to* operator $\rightsquigarrow$ (written `-->`), where $\varphi \rightsquigarrow \psi$ is equivalent to $\forall\Box(\varphi \rightarrow \forall\Diamond\psi)$. The formula can now be expressed as

```
Press --> (Coffee and x<5).
```

UPPAAL has also been extended for cost optimal reachability analysis in UPPAAL CORA [20]. CORA is used to specify priced timed automatons and can calculate the minimum cost for which a property can be verified.

# Chapter 3

# Behavioral Models

Developing the behavioral models is often considered the most time consuming phase of MDD. Modeling the system requires carefully studying the informal and formal requirements, and developing a formal model which conforms to them.

The modeling phase has the inherent choice of granularity in the model. If the model is too precise about all minor aspects of the system, it becomes too complicated, both for maintaining and for verification. If, on the other hand, the model is too coarse, there might be errors is the design which are not caught.

There are many different languages for representing formal models. The choice of modeling formalism highly depends on the system and what types of analysis and testing to be performed.

## 3.1 Labeled Transition Systems

Transition systems are a general class of models, which can be represented as graphs, with nodes as states, and directed edges as transitions. Transition systems are generally seen as an intuitive format for modeling state-based reactive systems, since a state in the transition system can represent one (or more) states in the implemented system.

Labeled Transition Systems (LTS) [96] are a very simple state-based modeling formalism. Its simplicity makes it possible to analyze, verify and prove properties. Since LTS are simple and well understood, they are often used as an underlying semantics for other formalisms, where the semantics are described as a translation to LTS.

**Definition 1.** *A* labeled transition system *is a 4-tuple* $(S, L, T, s_0)$*, where:*

- $S$ *is a non-empty set of* states*,*
- $L$ *is a countable set of* labels*,*
- $T \subseteq S \times (L \cup \{\tau\}) \times S$ *is the* transition relation*, and*

- $s_0 \in S$ *is the* initial state.

$L$ represents the observable interaction of a system, $\tau \notin L$ represents an unobservable internal action.

An example of a vending machine modeled as an LTS can be seen in Figure 3.1. The state $p_1$ is the initial state, symbolized by the anonymous incoming

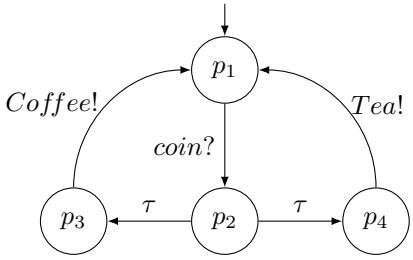


Figure 3.1: Example LTS

arrow. The LTS models a system which accepts a coin and non-deterministically produces coffee or tea. Even though LTS have no notion of inputs and outputs, the standard convention of symbolizing outputs with ! and inputs with ? is used. Once the coin is inserted it is not possible to known which state the system is in until coffee or tea is observed.

## 3.2 Extended Finite-State Machines

Extended Finite-State Machines (EFSM) [101] are an extension allowing data variables in the machine, and conditional branching depending on the value of variables.

**Definition 2.** *An* extended finite-state machine *is a 7-tuple* $(Q, q_0, \Sigma, \Gamma, V, \psi, \delta)$*, where:*

- $Q$ *is a finite, non-empty* set of states.
- $q_0 \in Q$ *is the* initial state.
- $\Sigma$ *is the* input alphabet*, a non-empty finite set of events.*
- $\Gamma$ *is the* output alphabet*, a non-empty finite set of events.*
- $V$ *is a finite* set of variable names.
- $\psi \subset V \to Val$ *assigns initial values to the variables.*
- $\delta \subseteq Q \times (\Sigma \cup \Gamma \cup \tau) \times \mathcal{B}(V) \times \mathcal{A}(V) \times Q$ *is the* state transition relation.

*$\delta$ relates a source state $q$, and an label $l$, to a target state $q'$, given the current state of variables, $\psi$. This is written: $q \xrightarrow{l} q'$, and corresponds to a* transition *in the system. The labels are on the form $e[g]/a$, where $e \in \Sigma \cup \Gamma \cup \tau$, $g \in \mathcal{B}(V)$ is a guard of boolean formula over the variables, and $a \in \mathcal{A}(V)$ is a set of assignments with a standard expression language. A transition is only active if the guard evaluates to true.*

EFSM can be used to model systems with data dependencies. An example EFSM can be seen is Figure 3.2. In the example a label with no guard is always

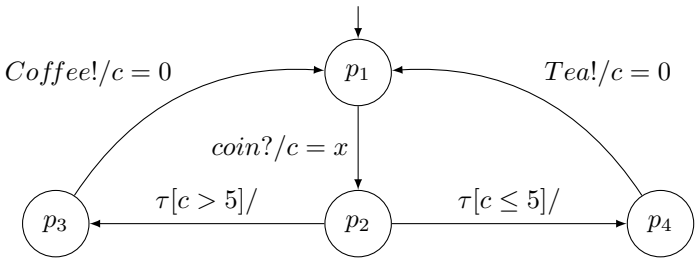


Figure 3.2: Example EFSM

enabled. The variable $c$ is internal and $x$ represents an input variable. The machine models a system which produces coffee if a coin of value more than 5 DKK is inserted, otherwise tea is produced.

As mentioned LTS is often used as underlying semantics for formalisms. This is also done of EFSM, where the semantics are described as a translation from EFSM to LTS. The semantics for $\mathcal{EFSM} = (Q, q_0, \Sigma, \Gamma, V, \psi, \delta)$ are defined as $\mathcal{LTS} = (S, L, T, s_0)$ where,

- $S = Q \times Val^V$,
- $L = \Sigma \cup \Gamma$,
- $T$ contains an edge $(q, \bar{v}) \xrightarrow{l} (q', \bar{v}')$ for all $(q, l, g, a, q') \in \delta$ where the guard $g$ evaluates to true given the values in $\bar{v}$, and $\bar{v}' = \bar{v}[a]$,
- $s_0 = (Q, \bar{v})$, where $\bar{v}$ is a vector containing the values of variables according to $\psi$.

Here, $\bar{v}[a]$ refers to the values of the variables in $\bar{v}$ after the assignments in $a$ have been performed. The number of state in $\mathcal{LTS}$ (or the size of the *state-space* of the $\mathcal{EFSM}$) is $|Q| \times |Val|^{|V|}$. In case the set of possible values $Val$ is infinite, then the $\mathcal{LTS}$ becomes infinite.

## 3.3 Timed Automatons

Timed Automaton (TA) [10, 19] have been created specifically to model embedded systems with timing properties. TAs describe real-time systems, whose

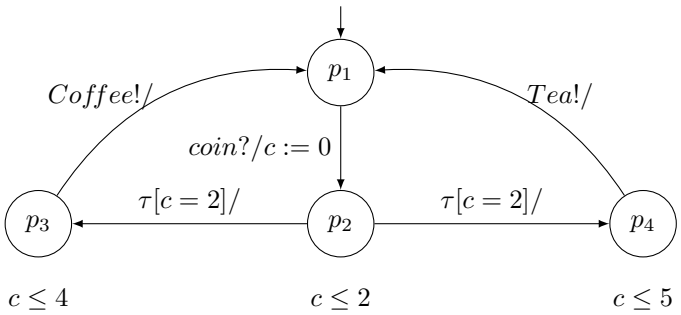


Figure 3.3: Example TA

semantics can be described using infinite state labeled transition systems, where actions of transitions are either delay quantities (non-negative reals), or discrete actions. This class of labeled transition systems is often referred to as timed labeled transition systems.

The syntax of TAs is essentially finite-state automatons extended with a finite collection of real-valued clock variables. The locations and edges of the TAs are decorated with constraints on the clock variables, restricting (and enforcing) when discrete transitions corresponding to the edges of the timed automaton may (or must) be taken.

A *clock constraint* is a conjunctive formula of atomic constraints of the form $x \sim n$ or $x - y \sim n$, where $x, y \in \mathcal{C}$ are real-valued clock variables, $\sim \in \{\leq, <, =, >, \geq\}$, and $n \in \mathbb{N}$. $\mathcal{B}(\mathcal{C})$ denotes the set of clock constraints. Note that even though the clocks are real-valued, the constraints are limited to integer values.

**Definition 3.** *A* timed automaton *is a tuple* $(S, s_0, \Sigma, E, I)$ *where*

- $S$ *is a finite set of states,*
- $s_0 \in S$ *is the initial state,*
- $\Sigma$ *is a finite alphabet of labels,*
- $E \subseteq S \times \mathcal{B}(\mathcal{C}) \times \Sigma \times 2^{\mathcal{C}} \times S$ *is the set of edges, and*
- $I : N \to \mathcal{B}(\mathcal{C})$ *assigns invariants to locations.*

*We write* $s \xrightarrow{g,a,r} s'$ *when* $(s, g, a, r, s') \in E$*, to represent an edge from* $s$ *to* $s'$ *with clock constraints* $g$*, label (or action)* $a$*, and* $r \in 2^{\mathcal{C}}$ *represents the clocks which will be reset on the transition.*

Figure 3.3 shows the example coffee machine as a timed automaton with clock constraints. Edges in the example are on the form $a[g]/r$, invariants are written next to their associated state. The example models a system which after a coin is inserted waits two seconds before deciding to make coffee or tea, and then produces coffee within two seconds, or tea within three seconds.

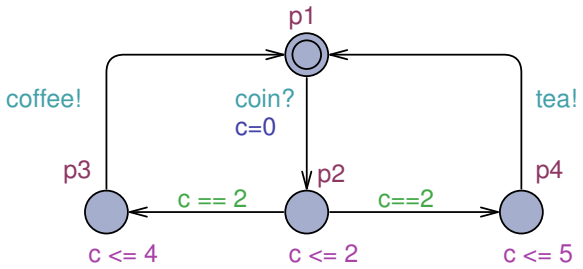


Figure 3.4: Example TA in UPPAAL syntax

TA are used as formalism in several model checking tools, e.g. UPPAAL [22] and KRONOS [104]. The tools extend the formalism with new features as syntactic sugar to ease the development of models, such as data variables, arrays, structures etc. The example coffee machine from Figure 3.3 is shown using UPPAAL syntax in Figure 3.4.

Again the semantics can be expressed as LTS. States in the LTS are made for each $S \times \mathbb{R}_+^{\mathcal{C}}$, so the state of the LTS is seen as a pair $(s, u)$ where $s \in S$ and $u$ maps each clock in $\mathcal{C}$ to a value from $\mathbb{R}_+$. Two types of transitions are possible in the LTS, delays and actions, defined as

- $(s, u) \xrightarrow{d} (s, u + d)$ if $u$ and $(u + d)$ satisfy $I(s)$, where $d \in \mathbb{R}_+$ and
- $(s, u) \xrightarrow{a} (s', u')$ if $s \xrightarrow{g,a,r} s'$, $u$ satisfies $g$, $u'$ is the same as $u$ with the clocks in $r$ reset, and $u'$ satisfies $I(s')$.

An example trace for the automaton in Figure 3.3 could be:

$$(p_1, 0) \xrightarrow{3.4} (p_1, 3.4) \xrightarrow{coin?} (p_2, 0) \xrightarrow{2.0} (p_2, 2.0) \xrightarrow{\tau} (p_3, 2.0) \xrightarrow{1.3} (p_3, 3.3) \xrightarrow{Coffee!} (p1, 3.3).$$

## 3.4 Other Modeling Formalisms

### 3.4.1 Process Algebra

Process algebra (or process calculi) (e.g. $\pi$-calculus [71], CSP [54], CCS [70], or ACP [24]) focus on modeling concurrent systems. They allow modeling of processes with channel communication and parallel composition of processes. Process algebra are often used to model communication protocols and have been used for verifying e.g. authentication [91] and key-exchange protocols [86].

Generally process algebra have a notion of processes and communication between processes using message-passing through channels. The precise syntax and semantics differ, and some formalisms incorporate timing, stochastic behavior, etc. As an example of a process algebra we consider Calculus of Communicating Systems (CCS).

CCS is a formal language for describing patterns of interaction in concurrent systems. CCS has syntax for defining processes and for composition of processes and describing the interaction between the processes.

Consider the process

$$CM = coin.\overline{coffee}.0.$$

This process describes a system which accepts a coin, then produces coffee, and then stops. This can be extended with a non-deterministic choice

$$CM = coin.(\overline{coffee}.0 + \overline{tea}.0),$$

describing a machine which produces either coffee or tea after receiving a coin. By making the machine recursive we get an infinite coffee/tea producing machine

$$CM = coin.(\overline{coffee}.CM + \overline{tea}.CM).$$

This machine can be interleaved with a process for a person inserting coins

$$Person = \overline{coin}.Person.$$

The interleaving machine describes a system which produces beverages infinitely often

$$(CM|Person)\backslash coin \equiv$$

$$M = (\overline{coffee}.M + \overline{tea}.M)$$

### 3.4.2 Petri Nets

A Petri net (or Place/Transition Net) [79, 84] is a modeling formalism often used for modeling distributed systems. The Petri net consists of a Petri net graph and a marking, assigning tokens to places in the Petri net graph. A Petri net graph consists of places, transitions, and arcs between them.

**Definition 4.** *A* Petri net graph *is a 3-tuple* $(S, T, W)$*, where*

- $S$ *is a finite set of* places*,*
- $T$ *is a finite set of* transitions*, and*
- $W : (S \times T) \cup (T \times S) \to \mathbb{N}$ *is a multiset of arcs, which assigns* arc multiplicatives *to arcs between places and transitions (and vice versa).*

**Definition 5.** *A* Petri net *is a 4-tuple* $(S, T, W, M_0)$*, where*

- $(S, T, W)$ *is a Petri net graph and*
- $M_0 \in M : S \to \mathbb{N}$ *is the initial marking.*

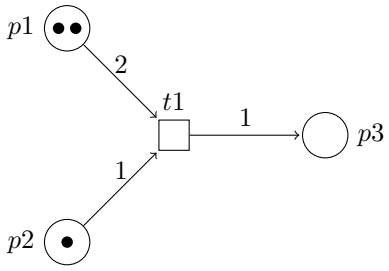


Figure 3.5: Example Petri Net

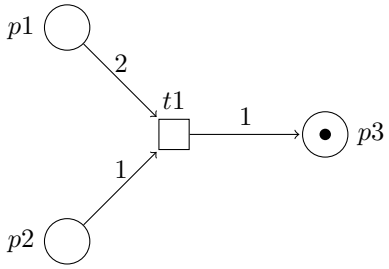


Figure 3.6: Example Petri Net after transition is fired

An example of a Petri net can be seen in Figure 3.5. Circles represent places, squares represent transitions, and arrows between them represent arcs. The number next to the arcs represent the multiplicative. The dots in the places represent the marking. In the example place $p1$ has two tokens and $p2$ has one. Arcs going into a transition represent tokens the transition consumes, arcs going out of transitions represent tokens the transition produces. The multiplicative on the arcs represents how many tokens are consumed/produced. When all incoming arcs to a transition have enough tokens the transition is enabled and can be fired. When the transition fires, all incoming tokens are consumed (removed), and all outgoing are produced (created). In the example the transition $t1$ is enabled because $p1$ has two tokens, which is required by the arc $p1 \xrightarrow{2} t1$, and $p2$ has one token, which is required by the arc $p2 \xrightarrow{1} t1$. When $t1$ fires, two tokens are consumed from $p1$ and one token from $p2$ leaving both places empty. One token is produced in place $p3$, resulting in the Petri net shown in Figure 3.6. Petri nets are good for modeling concurrency and are often used for modeling distributed systems and in particular work flows. Petri nets have been extended with quantitative properties, e.g. time [28] and cost [6].

### 3.4.3 UML State Machines

A well-known modeling formalism is the Unified Modeling Language (UML) [48]. UML is a standardized general-purpose modeling language. As such it has a lot of features and is intended for modeling several different areas of systems, such

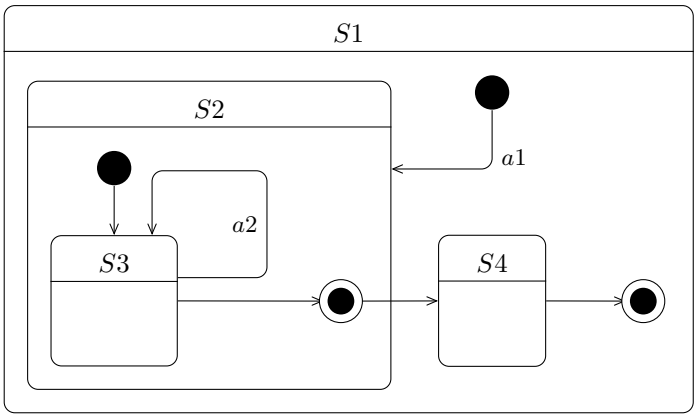


Figure 3.7: Example UML State Machine

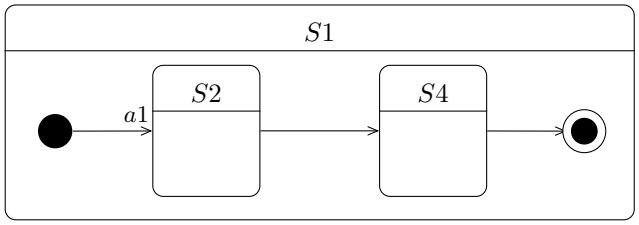


Figure 3.8: Example UML State Machine Simplified

as system structure, behavior, and architecture, it has even been used to model business processes and data structure.

For modeling the behavior of systems, (Behavioral) UML State Machines are used. UML State Machines have several extensions over the simpler finite automatons, most notably, they are hierarchical. This means that when an outer (or composite) state is reach, control is said to be transfered to an inner state machine. The inner state machine can have internal transitions as well as external transitions to other outer state machines. An example UML State Machine is seen in Figure 3.7. Frames with a heading represent (possibly composite) states. Solid black circles represent initial states. Black circles with a ring represent final states. $S1$ waits for action $a1$, then transfers control to the initial state of $S2$. Once control reaches the final state of $S2$ it is transfered outside $S2$ to $S4$.

Nesting state machines can be used to abstract detailed behavior and get an overview for the functionality of the machine. By hiding the internal behavior of $S2$ the machine is simplified, and an overview is easier to comprehend. This simplified machine is shown in Figure 3.8.

Due to the many features of UML, it has been difficult to develop a formal semantics. Some work has been done, but it has been found problematic to define all the more advanced features [63, 43]. Due to the large feature set

and the lack of formal semantics, it becomes difficult to make automatic model-checking and analysis of UML models. Those who try use a limited set of features from UML, e.g [60] and the Rhapsody environment [88].

# Chapter 4

# Model Checking

Now that we have examined how to model requirements and how to model behavior, we can combine these two with model checking, to analyze whether the behavioral models adhere to the requirements. How this is done and which techniques are used depends on which formalisms are used and which types of queries are being posed.

Some formalisms go well together, for instance specifying behavior of state machines with temporal logics. On the other hand specifying state-based behavior with constraint systems is not directly possible. This section focuses on model checking state machines with temporal logics.

## 4.1 State Reachability

Using LTS as modeling formalism, a simple property to check is whether a given desired goal state $s_g$ is reachable. A general state reachability algorithm is shown in Algorithm 1.

The algorithm keeps two sets, *waiting* and *passed*. The waiting set contains the states which have been reached but not yet checked. The passed set contains the states which have been checked. The algorithm starts with the initial state in the waiting set. The algorithm keeps removing states from the waiting set and checking all successors of this state for the goal state. If the goal state is not found the successor states are added to the waiting set, unless they are already in either the waiting set or the passed set.

If the goal state is reached the algorithm terminates positively. If the waiting set becomes empty the entire state space has been traversed. If the goal state has not been reached the algorithm terminates negatively.

This simple algorithm can only check if a certain state is reachable. The temporal logic formulas defined in Chapter 2 are formed over *atomic propositions* $AP$. Atomic propositions are simple facts known about the state of the system, e.g. "$x$ equals 5" or more abstract "the coffee machine is empty". These are often used when translating more complex formalisms to transition systems. We

**Algorithm 1**: **Reach**$(\mathcal{F}, q_g)$

**Input**: $LTS$ $\mathcal{L} = (S, L, T, s_0)$, a goal state $s_g \in S$
**Output**: "$s_g$ is reachable" or "$s_g$ is not reachable"

```
1:  waiting := {s_0}
2:  passed := ∅
3:  while waiting ≠ ∅ do
4:     Select and remove s from waiting
5:     passed := passed ∪ {s}
6:     for all s', where T(s, l) = s' do
7:        if s' = s_g then
8:           return "s_g is reachable"
9:        end if
10:       if s' ∉ waiting ∪ passed then
11:          waiting := waiting ∪ {s'}
12:       end if
13:    end for
14: end while
15: return "s_g is not reachable"
```

extend LTS with atomic propositions:

**Definition 6.** *A* transition systems $TS$*, over a set of atomic propositions* $AP$*, is a 5-tuple* $(S, L, T, s_0, P)$*, where:*

- $(S, L, T, s_0)$ *is a LTS,*
- $P : S \to 2^{AP}$ *is a proposition labeling function.*

With the atomic propositions we can modify the reachability algorithm to check if a state with the desired propositions is reachable. Algorithm 1 can only be used to check reachability (or its dual, safety). So this is a very specific class of simple (non-nested) LTL properties. To check nested properties a more complex LTL model-checking algorithm is needed.

## 4.2 LTL Model-Checking

An LTL model-checking algorithm is a decision procedure, which given a transition system $TS$ and LTL formula $\varphi$ returns "yes" if $TS$ satisfies $\varphi$ (written $TS \models \varphi$) and returns "no", together with a counter-example, if $TS \not\models \varphi$.

The model-checking algorithm relies on the following observations:

$$\begin{aligned} TS \models \varphi \quad &\iff \quad Traces(TS) \subseteq Words(\varphi) \\ &\iff \quad Traces(TS) \cap Words(\neg\varphi) = \emptyset \\ &\iff \quad Traces(TS) \cap \mathcal{L}_\omega(\mathcal{A}) = \emptyset \end{aligned}$$

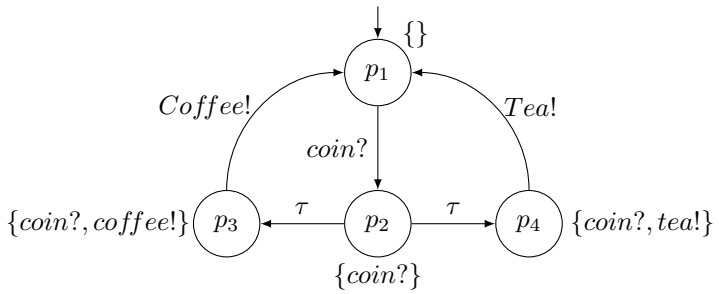


Figure 4.1: LTS using atomic propositions

With $Traces(TS)$ being the set of all atomic proposition traces of $TS$, $Words(\varphi)$ being all words of atomic propositions in $AP$ that satisfy $\varphi$, and $\mathcal{L}_\omega(\mathcal{A}) = Words(\neg\varphi)$ being the language accepted by a non-deterministic Büchi automaton (NBA) for $\neg\varphi$. The observations specify that $TS$ satisfies $\varphi$ iff its set of traces is a subset of the set of all words satisfying $\varphi$. In other words, if the intersection of the set of traces and the words not satisfying $\varphi$ is the empty set.

The idea of the model-checking algorithm is to generate a NBA $\mathcal{A}$ which accepts on $Words(\neg\varphi)$ and constructing the product transition system $TS \otimes \mathcal{A}$. If there exists an infinite run $\pi$ in $TS \otimes \mathcal{A}$ satisfying the accepting condition of $\mathcal{A}$ then $TS \not\models \varphi$ and (in the case of safety properties) a prefix of $\pi$ is returned as counter example, otherwise $TS \models \varphi$. For more information see [16].

To illustrate the approach, let us consider an example. Recall the example LTS in Figure 3.1, on page 16. To allow model checking on this LTS, we will add atomic propositions to the states. The set of atomic propositions will be $AP = \{coin?, coffee!, tea!\}$. The atomic proposition $coin?$ represents a state where a coin has been entered, but no beverage has been produced. The atomic propositions $coffee!$ and $tea!$ represent coffee and tea producing states, respectively. This LTS is shown in Figure 4.1.

On this LTS we would like to verify that coffee is always returned after a coin is inserted. This can be done with the LTL property

$$\Box(coin? \rightarrow \Diamond coffee!).$$

We need the negation of this property

$$\neg(\Box(coin? \rightarrow \Diamond coffee!)).$$

An NBA of this property can be generated using different algorithms. Since the complexity of the model checking problem depends on the size of the NBA, much research has gone into reducing the size of the NBA (see e.g. [47]). A reduced NBA accepting precisely the traces which satisfy the LTL property is shown in Figure 4.2. This NBA accepts if at some point a state is observed which does not have the proposition $coffee!$ but has the proposition $coin?$ and

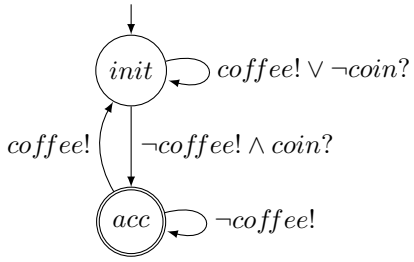


Figure 4.2: Büchi automaton of LTL property

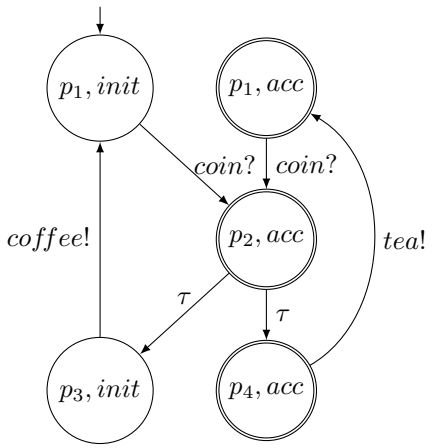


Figure 4.3: Product transition system

all states hereafter do not have $coffee!$. This precisely represents a state in which entering a coin does not lead to coffee being produced.

We now construct the product transition system, $TS$, of this NBA and the LTS of our system. The states in $TS$ are the product of states in the NBA and the LTS. Transitions are allowed in $TS$ if there is a transition in the LTS and the atomic propositions in the goal state agree with the transition label in the NBA. The product transition system is shown in Figure 4.3. Unreachable states and atomic propositions are omitted, and accepting states in the NBA are marked as accepting. The task is now to check that there is no infinite trace in $TS$ which visits an accepting state the NBA infinitely often. Quite obviously the trace $\langle p_1, init\rangle(\langle p_2, acc\rangle\langle p_4, acc\rangle\langle p_1, acc\rangle)^\omega$ visits $acc$ of the NBA infinitely often. Thus we have $Traces(TS) \cap \mathcal{L}_\omega(\mathcal{A}) \neq \emptyset$ and therefore $TS \not\models \varphi$.

This should not be surprising since LTS can choose to produce tea instead of coffee. If we remove $p_4$ in the LTS, and thereby remove to possibility to make tea, then the product transition system would be unable to go to $\langle p_4, acc\rangle$ and would thus be unable to make an infinite trace with an accepting state.

## 4.3 Symbolic States

Algorithm 1 traverses the state space by enumerates all states. For small LTSs this is not a problem, but as the LTS grows performance issues arise. The problem becomes evident when LTS are used as underlying semantic for other formalisms. For instance EFSM can be translated into LTS with one state for each variable valuation. This reduces the problem of model checking EFSM to that of model checking LTS. However, the state space of the LTS grows exponentially in the number of variables and the domain size of the variables of the EFSM. In the case of infinite domains, the state space becomes infinite and enumeration is impossible.

To solve this problem, symbolic states are introduced. A symbolic state covers several states in the underlying LTS. In the case of EFSM, this can be done using predicate abstraction over the variables. Instead of having a state in the LTS for each valuation of the variables in the EFSM, we can abstract the values into predicates and store the value of the predicates instead.

Consider the example in Figure 3.2 on page 17. We have one variable $c$, so the number of states in the underlying LTS is $|Q| \times |Val|$ (the number of states in the EFSM times the size of the domain of $c$). If $Val$ is infinite, the state space becomes infinite, but we can bound the size of $Val$ to enable model checking. For instance $Val = \{1, 2, \ldots, 100\}$ makes the size of the state space $4 \times 100 = 400$ states, which can easily be enumerated. If, however, we had two variables the size would be $4 \times 100^2 = 40000$, and for three variables $4 \times 100^3 = 4000000$. We can see that the state space quickly grows too big for model checking, this is commonly referred to as state-space explosion.

If we examine the example we can see that the precise value of the variable is not used. The variable is only checked for being less than or equal to 5 or greater than 5. Using this we can abstract the value away and store the value relative to 5. We only need a single predicate $c \leq 5$, if this predicate is true, we know the original value was less than or equal to 5, otherwise it is greater than 5. When a value is input to the machine, we update the predicate instead of storing the value, and the guards check the predicate for which transitions are enabled.

Using this predicate abstraction the size of the state space becomes $4 \times 2 = 8$ states. For two predicates the size is $4 \times 2^2 = 16$ states, even for ten predicates the size is only $4 \times 2^{10} = 4096$, which is quite manageable. This abstraction even works when $Val$ is infinite, so long as the number of predicates is finite.

Binary encodings such as this can be very efficiently handled using Binary Decision Diagrams (BDD) [7]. A BDD is a data structure for representing boolean functions. By encoding the states and state transition function as binary functions and using symbolic states as demonstrated above, the entire model can be encoded as BDDs. This allows for very efficient implementations of model checkers, which can handle large state spaces.

In the case of TA, the variables are real-valued clocks. This makes the state space uncountable, which makes it impossible to enumerate all states. To abstract the exact values of the clock variables we may first benefit from

observing the largest value $k_c$ which clock $c$ is compared to in any clock guard or invariant in the TA. Once $c$ reaches a value above $k_c$ the precise value becomes irrelevant, since all values above $k_c$ show the same behavior. Since the clocks are only used in invariants and guards, both of which are formed over clock constraints, we can use techniques similar to predicate abstraction, however, since the values are real-valued, and the values represent clocks, it becomes more complicated. The idea is to split the clock valuations into regions, where clocks in the same region satisfy the same clock constraints and allow the same behavior. For each clock, $c$, its value domain is separated into regions based on the positive integer numbers as such:

- A region is made for each integer, $c = n$, for $n \in 0, 1, \ldots, k_c$,
- for each pair of adjacent integers, a region is made, $n < c < n + 1$, for $n \in 0, 1, \ldots, k_c - 1$, and
- a region is made based on the highest integer $k_c$ a clock $c$ is compared to in a guard or invariant, $k_c < c$.

This separates the possible values of each clock into finitely many regions. For a set of clocks, regions are made for all combinations of regions of each individual clock. To guarantee correct behavior for time delays, a diagonal constraint is added:

- For each pair of distinct clocks $c$ and $d$, and integers $n < k_c$ and $m < k_d$. The region described by $n < c < n + 1 \wedge m < d < m + 1$ is separated into three regions. One for each of the cases: $frac(c) < frac(d)$, $frac(c) = frac(d)$, and $frac(c) > frac(d)$. Where $frac(c)$ represents the fractional part of the real number $c$.

This makes the state space finite and enumerable, however it becomes very large, as the entire domain of valuations is split into small, discrete areas. Consider for instance the TA in Figure 3.3. For the one clock $c$ we can observe the largest constant $k_c = 5$. This means the set of regions is:

$$\{c = 0, c = 1, c = 2, c = 3, c = 4, c = 5,$$
$$0 < c < 1, 1 < c < 2, 2 < c < 3,$$
$$3 < c < 4, 4 < c < 5, c > 5\}$$

For just a single clock with highest constant 5 we have 12 regions. If we add a second clock, also with highest constant 5, the set of regions is bound by 1152. As shown in [10] the number of regions grows exponentially in the number of clocks and their highest constants.

To reduce the state space further, clock constraints are used to union the regions into zones, to obtain a coarser abstraction. This makes the size of the state space dependent on the number of clocks and clock constraints. Model checking tools often store zones in a data structure called difference bound matrix (DBM) [21]. This technique, with some further implementation tricks makes model checking TA much more efficient. For further details see [65, 16].

## 4.4 Model Checking and Static Analysis

Static analysis [74] (or program analysis) is a set of techniques for statically analyzing the dynamic behavior of programs. It is most known for its pervasive use in compilers, e.g. to remove dead code, or avoid re-computation of available results. Recently static analysis has been used to analyze code for bugs and to verify invariants. Rice's theorem [85] teaches us that deciding non-trivial properties of programs is impossible. Therefore, static analysis performs under-(over-)approximations of the possible computations of the program to be able to guarantee that some behavior can (not) be exhibited.

For several years a link between model checking and static analysis has been suspected and researched. The traditional view is that static analysis performs approximations of the solution set very efficiently, while model checking performs an exact analysis of the solution set with more performance issues. However, advances in both areas have blurred this simplistic view. It has been shown how static analysis can be seen as an instance of model checking [89, 90, 92] and how model-checking can be seen as an instance of static analysis [75].

The general idea is to use techniques from one field and applying them in the other, or to use both techniques in tandem, to increase precision of the approach, while reducing complexity and execution time. Much recent research tries to bridge the gap between static analysis and model checking. For instance [93] uses the static analysis technique of *slicing* (normally used to reduce the syntactical size of applications) to reduce the size of models in UPPAAL before performing model checking, to improve verification speed. In [26] static analysis is used to compute partial order information which is used to reduce the state space for model checking. In [89] static analysis is used to create an abstract form of the system, which is then analyzed using model checking.

# Chapter 5

# Model-Based Testing and Learning

The verification techniques used in model checking can prove correctness of the system design. It must be noted that this is correctness of the models of the system, but in principle says nothing about the actual system. Model-Based Testing (MBT) is used to bridge the gap between the models and the actual system (often called system under test (SUT)).

MBT uses models describing required behavior to automatically generate test cases, and uses the models as oracles to automatically validate the outcome of test case execution. The description of MBT presented here is based on the formal testing approaches developed by Jan Tretmans [98, 95]. The approach is said to be formal, specification based, active, black-box, functionality testing. Figure 5.1 shows an overview of the process. This process does not assume any specific formalisms. Rather it presents a generic overview of how a MBT process is arranged.

We start with a specification of required behavior. This specification is some model presented in a formal language, such as those mentioned in Chapter 3. Let $SPEC$ be the set of all valid expressions in this language, the specification $s$ is an element in this language $s \in SPEC$.

The implementation is the system we want to test, the SUT. This can be anything from a simple software system to physical hardware devices with sensors and actuators. The implementation is viewed as a black-box system, as such, we can only interact with it through its interfaces. The process of testing aims to judge correctness based on behavior on these interfaces.

MBT aims to check whether the SUT *conforms to* a specification $s$. Since the specification is a mathematical object taken form the formal domain $SPEC$ and the implementation is a real system, we can not formally reason about conformity between them. The trick here is to assume that the SUT can be modeled by some formal model in the domain $MOD$. We assume that for an implementation there exists a model $i \in MOD$ which precisely describes

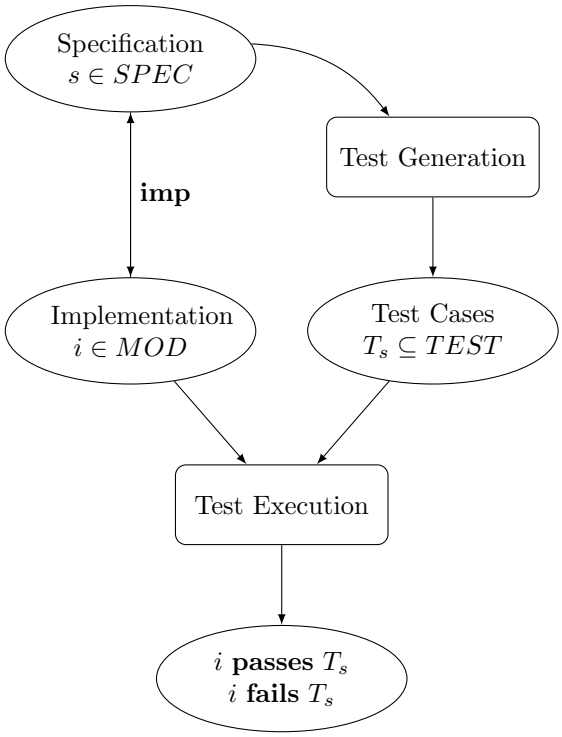


Figure 5.1: Model-Based Testing process (adapted from [98])

the behavior of the implementation. This allows us to reason formally about implementations, and we can now define an implementation relation **imp** $\subseteq MOD \times SPEC$. An implementation model $i \in MOD$ is said to be correct with respect to $s \in SPEC$ if $i$ **imp** $s$.

A test case is an experiment with stimuli applied to the SUT and expect responses from the SUT. Test cases are also elements of a formal language $t \in TEST$. A successful execution of test case $t$ on implementation $i$ is expressed as $i$ **passes** $t$, unsuccessful execution is expressed as $i$ **fails** $t \iff i$ **passes** $t$. For a test suite we have $T \subseteq TEST : i$ **passes** $T \iff \forall t \in T : i$ **passes** $t$.

Test cases are generated in test suites by some algorithm $gen_{\mathbf{imp}} : SPEC \to \mathcal{P}(TEST)$. We require this algorithm to generate test suites such that we can learn whether the implementation conforms to the specification. That is, we care looking for test suites such that $\forall i \in MOD : i$ **imp** $s \iff i$ **passes** $T_s$. Such a test suite is said to be complete (sound and exhaustive) since a correct implementation will pass, and a failed test case means the implementation is not correct. Such an algorithm is said to be complete if the generated suite is complete for all specifications.

To summarize: We have a formal specification of required behavior $s \in SPEC$; we assume a model exists which precisely describes the implementa-

tion $i \in MOD$ ; we generate a suite of test cases $T_s \in \mathcal{P}(TEST)$ such that $i$ **passes** $T_s \iff i$ **imp** $s$.

The test case generation and execution can be done in two separate phases as it is implied above. This approach is called *off-line* testing, where a complete batch of test cases is generated, to be executed on the SUT later. This approach has the benefits that it is easier to reason about the test cases regarding coverage and the exact same sequence of tests can be re-run. Off-line testing has two major disadvantages. First, it requires the specification to be analyzed in its entirety. This limits the size of systems that can be handled due to state-space explosion. Second, in the case of non-deterministic systems, a test case is not simply a series of inputs and outputs, but a tree which branches for every non-deterministic choice in the system. This may lead to very large test cases.

Test case generation and execution can also be interleaved, where a single step in a test case is generated, executed, and validated, before generating the next step in the test case. This approach is called *on-line* testing. In on-line testing the test case generation tool runs a simulation of the specification and is connected directly with the SUT. The testing tool observes the current state of the specification and selects an enabled input. The input is sent to the SUT, and possible outputs are observed. The state of the simulation is updated according to inputs and outputs. This approach is very good at handling large and non-deterministic systems. The state space of the system is more manageable, since only the subset of possible current states is stored. If this set becomes empty, then there is no possible way for the system to be in a correct state, i.e. the test case fails. Non-determinism is less of a problem since outputs are directly observed. In highly non-deterministic systems, the number of possible states can still be too big. However, the size of manageable systems is usually significantly larger, than for complete model-checking.

This generic testing framework can be instantiated with several different formalisms and its principles are used in several tools, e.g. TorX [99], Uppaal TRON [100]. TorX uses LTS and the **ioco** conformance relation as formalisms. TRON is as testing tool for testing real-time systems. It uses Uppaal TA formalism for modeling systems, and the relativized timed conformance relation.

## 5.1 Model Learning

So far we have assumed that a formal model is available as part of the system development. However, in some cases a formal model is not available. This could be in the case of legacy systems, or third-party systems, where formal specifications are normally not available. In these cases, automatic learning of a formal representation of an already implemented system is desirable.

This can be done using the $L^*$ algorithm developed by Angluin [15]. While the algorithm is originally developed for learning languages of deterministic finite automatons (DFA), it has been converted to allow learning of Mealy machines [76] as well as I/O automata [5]. The algorithm assumes a *teacher* which knows a Mealy machine (MM) $\mathcal{M}$, and a *learner* which initially only knows

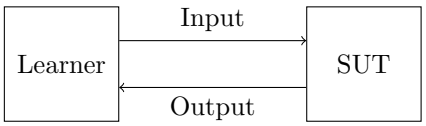


Figure 5.2: Model learning

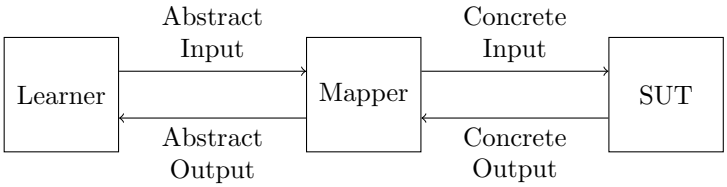


Figure 5.3: Model learning with mapper

the input language of $\mathcal{M}$. To infer a MM equivalent to $\mathcal{M}$, the learner can pose two types of queries to the teacher: *membership queries* and *equivalence queries*. Membership queries ask if a string $w$ is in the language accepted by $\mathcal{M}$. Equivalence queries ask if an hypothesized MM $\mathcal{H}$ is equivalent to $\mathcal{M}$. In case the hypothesis is wrong the learner provides a counterexample. This general setup is depicted in Figure 5.2.

If instead of a MM, the teacher has an actual system, this setup can be used to learn a model of the behavior of this system. This is implemented in the tool LearnLib [81][1]. While this process is automatic, it is very limited in the size of systems, it is possible to learn. The limit seems to be in the order of 40.000 states [69]. Particularly the algorithms are unable to handle data dependent systems, due to state-space explosion.

To alleviate this problem Aarts et al. [3] have proposed to introduce a *mapper* between the learner and teacher. This mapper translates concrete values from the system and teacher into abstract values used by the learner. This way the learner only observes a small set of abstract values, as opposed to the full set of concrete values. The setup with the added mapper is depicted in Figure 5.3.

These abstractions have been shown to be useful for learning realistic systems. The problem is that the abstractions are unknown a priori. Finding these abstraction requires either intimate knowledge of the SUT or requires running the learning process and observing the system manually, to determine possible abstractions. This abstraction can either be too coarse (the learned model is too simplistic), too detailed (the state space will be too large to allow efficient learning), or simply incorrect, thereby introducing non-determinism to the learner, in which case the learning will fail.

The ability to automatically learn such abstractions would greatly increase the usability the model learning, and increase the size of system which can be learned automatically.

[1] http://www.learnlib.de

# Chapter 6

# Thesis

The issues mentioned in the preceding chapters usually deal with state-space issues related to data variables. As mentioned these issues are often dealt with by abstracting away the data related parts during modeling. This technique can work very well for some types of systems, where data is only transported and stored. In some systems, however, the control flow depends heavily on the values of variables. We call such systems (or part of a system) *data sensitive*, since the execution of the system is sensitive to the data values.

In this thesis we deal with improving the quality of data-sensitive systems. Specifically we employ formal methods, such as model checking and model-based testing, and combine these with techniques from static analysis to reduce the state spaces. If we recall Figure 1.2 on page 3 showing an overview of model-driven development. Figure 6.1 shows an updated version with dashed boxes representing contributions from this thesis. The thesis improves the processes in three areas:

(I) Modeling and model checking of data-sensitive state-based systems,

(II) Model learning of data-sensitive state-based systems, and

(III) Model-based testing of data-sensitive systems, through the use of constraint systems.

The following sections describe the research question for each area.

## 6.1 Research Question I

Several modeling formalisms have been presented, several of which include data variables of some sort. All of these operate on data at a low level, which is not well suited for modeling complex data. This combined with the state-space explosion problem makes modeling and model checking of data-sensitive systems difficult. Therefore we pose the following research question:

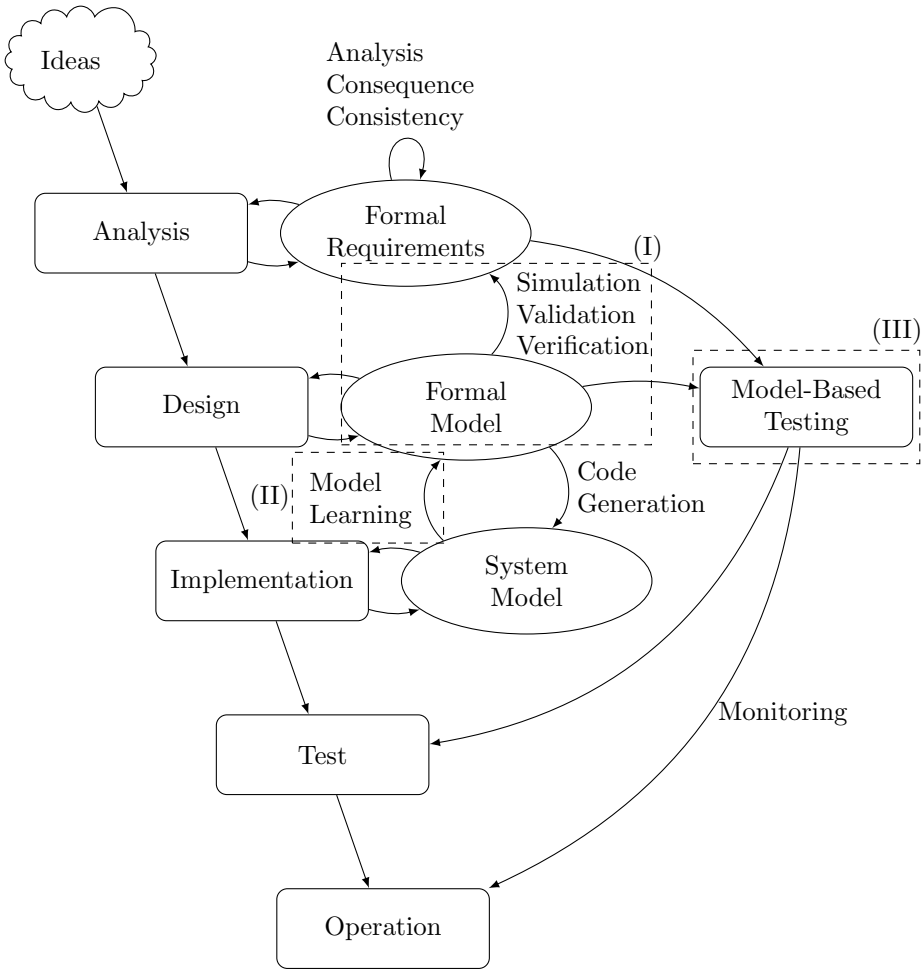


Figure 6.1: Contributions

> Can we make a general modeling formalism, combining state-based modeling and abstract interpretation, which covers a wide variety of applications, while allowing efficient modeling and model checking.

This research question is dealt with in three papers, Paper A, Paper B, and Paper C.

The first paper originated from ideas for improving MBT of data sensitive systems. The idea was to model a database as two sets: present-set and absent-set. The present-set contains elements which are known to be present in the database, while the absent-set contains elements known to be absent from the database. Elements which are neither in the present- nor absent-set, are said to be unknown, and can be either present or absent from the database. This

research led to the definition of the present-absent abstraction and how it can be used in MBT.

The second paper strives to generalize the modeling formalism presented in the first paper. It was observed that the abstraction employed fitted very well with the lattice framework used in abstract interpretation. It was also observed that other abstractions used in a similar manner fitted the same framework, notably the abstraction used for cache modeling [35]. This led to joint work formulating the *lattice transition system (LaTS)*. A LaTS is a transition system where the state of the system is a combination of a discrete state and an element from a lattice. This enables abstract model checking, where instead of traversing all data values individually, the algorithm can use *least upper bound* and *greatest lower bound* to make an abstract state space exploration. This approach was extended with a CeGAR approach, able to refine the abstraction in case of an inconclusive answer.

The third paper details the implementation of this approach in the prototyping tool `oppaal`.

Together, these three papers present a novel approach to modeling data-sensitive systems, together with algorithms and an implementation improving the performance of model checking data-sensitive systems.

## 6.2 Research Question II

This work builds on previous work of Fides Aarts and Frits Vaandrager. As mentioned, the problem with automatically learning systems, is the state space. This is particularly true when learning data-sensitive systems. To enable learning of larger systems, abstractions are introduced. This lead us to the research question:

> Is it possible to automatically learn an abstraction to allow automatic efficient learning of large systems.

In Paper D we introduced an approach to automatically learn such abstractions. The approach works on a restricted class of extended finite state machines. The system is not allowed to perform data operations on values, the system is only allowed to observe values on inputs and store them, check for equality on values, and return observed values as outputs. The system is only allowed to store the first and last occurrence of a value, but we expect this restriction to be easy to lift.

The approach is to have a mapper which learns the abstraction running in-between the learner (LearnLib) and teacher (SUT), during the learning phase. The mapper holds the currently known abstraction and tries to translate between concrete values and abstract values. If the mapper is unable to translate values according to the current abstraction, or the current abstraction produces non-determinism to the learner, the abstraction needs to be refined.

The idea in refining the abstraction is to examine the trace which shows non-deterministic behavior. Due to the restrictions on the SUT, we know that

only values that are present more than once on a trace can cause the non-determinism. In case several values are present more than once, the algorithm introduces fresh values until a new abstraction has been found.

The approach has been implemented in a prototype and has been shown to work on real life examples (biometric passport, session initiation protocol) as well as academic examples (alternating bit protocol).

## 6.3 Research Question III

We are presented with the problem of testing controllers for professional printers. The problem is posed by the printer manufacturer Océ-Technologies B.V. Océ were experiencing increasing load during testing, and were interested in reducing the number of test cases but not suffer in reduced test quality. The printer controller is a stateless system. The input is a set of parameter values describing a document to be printed and the output is a set of parameter values to the printer hardware describing how each sheet should be printed. Since this is not a reactive system, state machines are not suited for modeling. Therefore we pose the research question:

> Are constraint systems useful as oracle and for test case generation, in an industrial context.

In Paper E we present an approach to MBT, using constraint systems as oracle, and combinatorial testing as test case generation. Constraint systems are used to model the relationships between input values and output values. The constraint system is formed as a set of boolean formulas with implications. The premise of the implication is a conjunction of boolean formulas on the input values, and the conclusion is on the output values. E.g. if $A$ represents an input value and $B$ represents an output value, then a formula could be: $A < 3 \Rightarrow B = 5$. These models were used as oracle during test case execution. For test case generation, combinatorial testing was used to generate test cases with pair-wise coverage. The approach was implemented in the Python testing framework used at Océ, and was entered into their nightly test runs.

This approach resulted in measurable coverage (which they did not have before), and we got good coverage using a comparatively small number of test cases. Having models describing the relations between inputs and outputs also increased the developer's knowledge about the system, and reduced the effort required to locate the error in the code.

The following sections briefly present the papers of this dissertation. Each is presented with coauthor information, publication outlet, and an overview for contributions.

# Paper A: Present and Absent Sets

Petur Olsen, Kim G. Larsen, and Arne Skou

In: *Proc. Sixth Workshop on Model-Based Testing*, Paphos, Cyprus. Electr. Notes Theor. Comput. Sci., 264(3):53–68, 2010

We present a new abstraction of reactive systems interacting with databases. This abstraction is intended to be used for model-based testing. We abstract the database into two sets: present set and absent set, and present a proof of this abstraction. We present two extensions of FSM, the DBFSM and PAFSM. DBFSM are a form of FSM incorporating databases. PAFSM are an abstraction of DBFSM using present-absent sets. Depending on what type of testing is to be done, the translation is tailored to fit this purpose. We show how this translation is related to the present-absent abstraction. Finally, we illustrate the approach through a small example and show how this can be used for testing with the model-based testing tool Uppaal TRON.

## Contributions

- Abstraction of reactive system interacting with databases.
- Extension of FSM to DBFSM, representing systems interacting with databases.
- Abstraction of DBFSM to PAFSM, corresponding to the abstraction of systems.
- Describe MBT of PAFSM.

## Paper B: Model Checking with Lattices

Andreas Engelbredt Dalsgaard, René Rydhof Hansen, Kim Gulstrand Larsen, Mads Chr. Olesen, Petur Olsen, and Jiří Srba

Unpublished.

Model checking is a verification technique based on searching through a state space. As the state spaces are often large or even infinite due to unbounded data structures, techniques to tackle this problem have been investigated. We introduce a novel abstract model of labelled transition systems where states of the system are paired with elements from a lattice that provides a suitable abstraction of the real data. We present a general reachability algorithm with different update functions to account for different degrees of abstraction and prove the correctness and termination of the algorithm. Furthermore, we develop the notion of Lattice Guided Abstraction Refinement (LaGAR) for iterative recovery of precision that might be lost due to the applied abstraction. The usability of the framework is demonstrated on a number of applications that include communication protocols, databases, cache analysis and zones in timed automata. Prototype implementations in some of the application domains indicate promising results.

### Contributions

- Generalize the approach to fit other applications.
- Provide model-checking algorithms, with proofs.
- Provide CeGAR approaches to automatically refine the abstractions.

# Paper C: opaal: A Lattice Model Checker

Andreas Engelbredt Dalsgaard, René Rydhof Hansen, Kenneth Yrke Jørgensen, Kim Gulstrand Larsen, Mads Chr. Olesen, Petur Olsen, and Jiří Srba

In: *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 487–493. Springer, 2011.

We present a new open source model checker, `opaal`, for automatic verification of models using lattice automata. Lattice automata allow the users to incorporate abstractions of a model into the model itself. This provides an efficient verification procedure, while giving the user fine-grained control of the level of abstraction by using a method similar to Counter-Example Guided Abstraction Refinement. The `opaal` engine supports a subset of the Uppaal timed automata language extended with lattice features. We report on the status of the first public release of `opaal`, and demonstrate how `opaal` can be used for efficient verification on examples from domains such as database programs, lossy communication protocols and cache analysis.

## Contributions

- Implement the approach.
- Executed experiments.

# Paper D: Automata Learning Through Counterexample-Guided Abstraction Refinement*

Fides Aarts, Faranak Heidarian, Petur Olsen, and Frits Vaandrager

Unpublished.

Updated version published in: *FM 2012: Formal Methods - 18th International Symposium*, Volume 7436 of *Lecture Notes in Computer Science*, Paris, France, August 27-31, 2012.

State-of-the-art tools for active learning of state machines are able to learn state machines with at most in the order of 10.000 states. This is not enough for learning models of realistic software components which, due to the presence of program variables and data parameters in events, typically have much larger state spaces. Abstraction is the key when learning behavioral models of realistic systems. Hence, in most practical applications where automata learning is used to construct models of software components, researchers manually define abstractions which, depending on the history, map a large set of concrete events to a small set of abstract events that can be handled by automata learning tools. In this article, we show how such abstractions can be constructed fully automatically for a restricted class of extended finite state machines in which one can test for equality of data parameters, but no operations on data are allowed. Our approach uses counterexample-guided abstraction refinement: whenever the current abstraction is too coarse and induces nondeterministic behavior, the abstraction is refined automatically. Using a prototype implementation of our algorithm, we have succeeded to learn – fully automatically – models of several realistic software components, including the biometric passport and the SIP protocol.

## Contributions

- Present algorithm for automatic learning abstraction while learning a system.
- Present CeGAR approach to automatically refine abstractions.
- Present prototype implementation.
- Learn real-life systems.

# Paper E: Model-Based Testing of Industrial Transformational Systems

Petur Olsen, Johan Foederer, and Jan Tretmans

In: *Proc. 23rd IFIP Int. Conference on Testing Software and Systems (ICTSS'11)*, Paris, France, November, 2011

We present an approach for modeling and testing transformational systems in an industrial context. The systems are modeled as a set of boolean formulas. Each formula is called a clause and is an expression for an expected output value. To manage complexities of the models, we employ a modeling trick for handling dependencies, by using some output values from the system under test to verify other output values. To avoid circular dependencies, the clauses are arranged in a hierarchy, where each clause depends on the outputs of its children. This modeling trick enables us to model and test complex systems, using relatively simple models. Pairwise testing is used for test case generation. This manages the number of test cases for complex systems. The approach is developed based on a case study for testing printer controllers in professional printers at Océ. The model-based testing approach results in increased maintainability and gives better understanding of test cases and their produced output. Using pairwise testing resulted in measurable coverage, with a test set smaller than the manually created test set. To illustrate the applicability of the approach, we show how the approach can be used to model and test parts of a controller for ventilation in livestock stables.

## Contributions

- Present approach to model requirements for non-state-based systems.
- Present modeling tricks to manage complexities.
- Present approach for generating test-cases for the requirements.
- Implement the approach in industrial context.

# Chapter 7

# Future Work

MDD of data-sensitive systems is a complex problem with ongoing improvements from several fronts, both academic and industrial. On the academic side ongoing work is striving to invent and improve modeling formalisms to ease modeling, developing new (or improving old) algorithms to improve performance and quality, while on the industry side tool development and integration are needed.

Several avenues of improvement are available for the subjects discussed in this thesis. These are discussed below.

## 7.1 Research Question I

With regards to modeling databases in Uppaal several improvements can be made. The simplistic view on databases can be extended, modeling patterns can be made for each type of relational pattern (one-to-one, one-to-many, etc.). This can also be extended to automatic translation of a relational database schemes to models capturing these relations. Currently, ongoing work is striving to integrate test-case generation directly into the Uppaal GUI. This work could be extended to include testing of database systems.

The LaTS modeling formalism can be implemented in Uppaal, and model-checking and MBT algorithms can be extended to support them. This can be extended to include the CeGAR algorithm to allow dynamic abstraction refinement.

All this needs to be tried and performance tested on real-life applications.

## 7.2 Research Question II

The obvious improvement in this area, is lifting the restrictions on the SUT. First we can lift the restriction that only first and last occurrences of values may be stored. This restriction was mainly introduced to simplify the algorithm by

reducing the number of possible abstractions. Lifting this restriction greatly increases the complexity of the algorithm and its running time.

Currently, the available abstractions for the value of a parameter are *first* and *last*. Adding the possibility to store arbitrary values, the available abstractions would need to increase, but how? We expect a possible abstraction is to make the abstraction based on where in the model the value was stored. E.g. the value of a parameter store from state $A$ would get a different abstraction from the value store from state $B$. This would possibly allow learning of more systems.

The second restriction to be lifted is that no data operations are allowed on parameter values. This restriction was introduced to be able to track a value through the system. This restriction can be lifted (or reduced) by allowing few and simple known operations on values. For instance by allowing values to be added, it can be checked which of the input values were added to produce the output values.

For unknown operations, it might be possible to use invariant checkers, such as Daikon [42], to learn relationships between input values and output values.

Finally, a possible research area could be to incorporate the database abstractions from research question I into the learning. Using the approach of present-absent sets it might be possible to learn the behavior of a system, regardless of the initial internal state. This would require knowledge of the structure of the database. It might also be possible to learn the structure of the database while learning the abstraction and behavior.

## 7.3 Research Question III

This project illustrates how MBT can be employed in an industrial context, but the systems developed were prototypes, so improvements need to be made here. Currently the boolean formulas were implemented directly in the Python framework. Some way of abstractly describing the system as boolean formulas, and using these as oracle would be preferable.

This would also enable analysis and verification of the requirements. Sanity checks could be added to increase the confidence on the requirements. This could also be combined with other models of other aspects of the printer controller.

Finally, only a small part of the controller was modeled, this could be extended. Doing this would also give indications on how well suited this approach is to model complex industrial systems. Also, some experience on the maintainability of the models is needed.

# Paper A

# Present and Absent Sets: Abstraction for Testing of Reactive Systems with Databases

Petur Olsen

*Department of Computer Science*
*Aalborg University*
*Aalborg, Denmark*
petur@cs.aau.dk

Kim G. Larsen

*Department of Computer Science*
*Aalborg University*
*Aalborg, Denmark*
kgl@cs.aau.dk

Arne Skou

*Department of Computer Science*
*Aalborg University*
*Aalborg, Denmark*
ask@cs.aau.dk

**Abstract** We present a new abstraction of reactive systems interacting with databases. This abstraction is intended to be used for model-based testing. We

abstract the database into two sets: present set and absent set, and present a proof of this abstraction. We present two extensions of FSM, the DBFSM and PAFSM. DBFSM are a form of FSM incorporating databases. PAFSM are an abstraction of DBFSM using present-absent sets. Depending on what type of testing is to be done, the translation is tailored to fit this purpose. We show how this translation is related to the present-absent abstraction. Finally, we illustrate the approach through a small example and show how this can be used for testing with the model-based testing tool Uppaal TRON.

# 1 Introduction

Testing is generally considered the most widely used technique for error detection in software systems. Many systems today are heavily dependent on databases, there is however no efficient technique for testing systems using databases available. Several problems arise when testing systems dependent on databases. For instance, the test executed is dependent on the state of the database. Consider a test case requiring a user to be created. Running this test case after the user has been created is not possible without deleting the user first. Another problem is the huge amount of data stored in such databases.

Recently automated techniques and formal approaches have been developed for testing. One such being model-based testing (MBT) [53, 97, 98]. Doing MBT of a database systems is not trivial however. Consider modeling the entire database, this would require huge models and would not be practically possible. Some abstraction is needed in order to make MBT applicable to testing database systems.

This paper presents one such abstraction. We model the database as two sets, the present set and the absent set. The present set is an under-approximation of the data present in the database, and the absent set is an under-approximation of the data not present in the database. This way we can abstract over an infinite amount of databases with two small sets.

To enable model-based testing using this abstraction we present two new forms of FSM: DBFSM and PAFSM. We show that the present-absent abstraction is used to translate from DBFSM to PAFSM, and how specifications for testing can be developed using PAFSM. Additionally we show an example and how test cases can be generated from this example.

In this paper we consider reactive systems which interact with databases in a shallow manner, meaning no complex operations on the data are performed. Rather the system can insert or remove values to and from the database and the control flow of the systems can depend on the presence or absence of values. We refer to this simplistic view as databases althought databases are far more complex. The simplistic view in this paper is a starting point and is intended to be extended with a more complex view of databases.

The paper is structured as follows: Section 2 describes some related work. Section 3 describes model-based testing in its two forms, online and offline. Section 4 through 7 describe the theoretical parts of this paper. First the

present-absent abstraction is explained and proved. Then extended finite-state machines are explained, and these are further extended to include databases and present absent sets. The abstraction and translation between DBFSM and PAFSM is described in Section 8. A short example is presented in Section 10, and Section 11 concludes the paper.

# 2 Related Work

Ran et al. [83, 82] have proposed a similar approach, in a system they call AutoDBT. They model web-based systems using FSMs, and model the databases as two sets, the actual database, and a synthesized database. The synthesized database contains values not in the actual database, but available for testing. The synthesized database is used when the test is required to input some value into the database. These two databases are similar to our present-absent sets. They differ however, in that we only model a small subset of the data in the actual database. Additionally the testing algorithm differs in that AutoDBT generates guards to be executed before every test case, to ensure that the database is in a conforming state, whereas we populate the modelled databases according to the actual database to ensure that the model is always in a conforming state. Ran et al. do not specify what happens if the system never enters a conforming state for a specific test case. Additionally AutoDBT only supports offline testing whereas our approach supports both online and offline testing.

# 3 Model-Based Testing

Model-based testing originates in the formal approaches developed by Tretmans [97, 98], and implemented in the tool TorX [99]. These approaches have been extended to include real-time by Hessel et al. [53], and implemented in Uppaal TRON [64]. Also, a number of commercial UML-based tools are emerging, such as Qtronic and ATG.

Even though the aspects of this paper do not concern with real-time directly, it is intended to be used to extend Uppaal TRON to allow testing of data intensive systems. Uppaal TRON assumes timed automata as specification and supports conformance testing of real-time systems. Since the abstraction presented in this paper differs somewhat depending on whether the purpose is online or offline testing, a short description of these two types of testing is presented.

## 3.1 Online Testing

Online testing merges test-case generation and execution into one activity. The test cases are dynamically derived from a simulation of the model and sent to the implementation under test (IUT) directly. Output from the IUT is observed and the state of the model is updated accordingly. The advantages of online testing include easier handling of non-determinism and the reduction in

state-space. Non-determinism is easier to handle since the IUT is dynamically observed, thereby revealing which non-deterministic choices have been taken, eliminating the need for the test tool to track unnecessary states. The state-space is reduced for the same reason. Disadvantages include the difficulty to reason about coverage and the arbitrarily long traces complicating the process of linking an erroneous test case to an error in the IUT.

### 3.2 Offline Testing

Offline testing involves generating a batch of test cases prior to executing them on the IUT. Test cases are generated by model-checking for a specific purpose and storing the trace from the model-checker. This trace serves as a test case to be executed to test the purpose. The advantages of offline testing include the ability to specify and reason about coverage in a very precise manner. Disadvantages include problems with handling non-determinism and the requirement of model-checking the model, requiring the entire state space to be explored, which can lead to state-space explosion. Handling non-determinism is a problem since the test case needs to take into account all possible outcomes of a test purpose. Consider a test case requiring a user to be present in the database. If the user is not present he needs to be created before the test can proceed. Some test-case execution tools do not support such non-determinism. QTP, an industrially used test-case execution tool, only supports static test cases of produced inputs and observed outputs. This problem of requiring static test cases is major when testing databases which inherently depend on an internal state and evolve dynamically during testing.

## 4 Present and Absent Sets

We now introduce the present and absent set abstraction originally proposed in [77]. The abstraction abstracts a database into two sets; the *present set* and the *absent set*. The present set is an under-approximation of the values which are present in the database and the absent set is an under-approximation of the values which are not in the database. This can be seen as a three-valued-logic, where if the value is in the present set it corresponds to *true*, if the value is in the absent set it corresponds to *false*, and if the value is in neither it corresponds to *unknown*. If the value is in both sets it corresponds to an erroneous state, this should be avoided. This abstraction allows us to abstract over an infinite number of databases and abstract away from the actual content of the database, using a relative small set of values.

We define the following sets [77]:

$\mathbb{D}$ is a set of *elements* (e.g. records, relations, tuples etc.) The complete set of values that can be entered into the database.

$D_n \subset \mathbb{D}$ is the concrete *state* of a database. The database used by the real system and can contain huge amounts of data.

$\mathbb{C} \subset \mathbb{D}$ is a set of representative elements. These can be chosen intuitively or by some heuristic, e.g. a few from each table.

$P_n \subseteq \mathbb{C}$ is the *present set*, containing the elements known to be present in database $D_n$.

$A_n \subseteq \mathbb{C}$ is the *absent set*, containing the elements known to be absent from database $D_n$.

$d \in \mathbb{D}$ is an element in the actual system.

$c \in \mathbb{C}$ is an element in the abstract system.

Figure 1 illustrates these sets. $P_n$ can grow to fill the entire space $\mathbb{C} \cap D_n$ and $A_n$ can grow to fill the entire space $\mathbb{C} \setminus D_n$. Some interesting observations follow from these sets:

$P_n \subseteq D_n$ every element in the present set must be in the database.

$A_n \cap D_n = \emptyset$ no element in the absent set can be in the database.

$P_n = A_n = \emptyset$ means no knowledge about the contents of database $D_n$.

$P_n \cup A_n = \mathbb{C}$ means everything is known about database $D_n$, given the current $\mathbb{C}$.

$P_n \cap A_n = \emptyset$ must always hold. The same element can never be present in and absent from the same database at the same time.

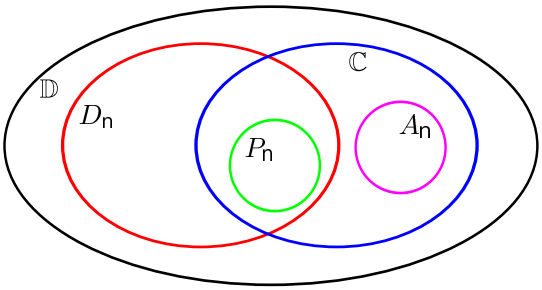


Figure 1: Sets of the present-absent abstraction

Three operations are allowed on the database: insert, remove, and query for presence. These operations are defined below on databases and present-absent sets. Queries are split into positive and negative since these are handled differently.

**Insert** Inserting into the database results in a new state with the inserted element added:

$$D'_n = D_n \cup \{c\}$$

This corresponds to adding the element to the present set and removing it from the absent set:

$$P'_n = P_n \cup \{c\}$$
$$A'_n = A_n \setminus \{c\}$$

**Remove** Removing from the database results in a new state without the removed element:

$$D'_n = D_n \setminus \{c\}$$

In the present-absent sets this results in removing from the present set and adding to the absent set:

$$P'_n = P_n \setminus \{c\}$$
$$A'_n = A_n \cup \{c\}$$

**Positive Query** This means that the element is in the database:

$$c \in D_n$$

If a query is positive we know the element is in the database, this means that we can add the element to the present set. We need to assert that the element is not in the absent, this is a consistency check.

$$P'_n = P_n \cup \{c\}$$
$$assert\ c \notin A_n$$

**Negative Query** This means that the element is not in the database:

$$c \notin D_n$$

We can update the present-absent sets similarly to positive query:

$$A'_n = A_n \cup \{c\}$$
$$assert\ c \notin P_n$$

**Theorem 1.** *Operations on $P_n$ and $A_n$ are consistent and sound with respect to an actual $D_n$ in the following sense:*

*(i) The $P_n$ and $A_n$ captured info does not contradict.*

*(a)* $\forall c \in D_n \Rightarrow c \notin A_n$

*(b)* $\forall c \notin D_n \Rightarrow c \notin P_n$

*(ii)* $P_n$ *and* $A_n$ *capture part of the* $D_n$ *state.*

*(a)* $\forall c \in P_n \Rightarrow c \in D_n$

*(b)* $\forall c \in A_n \Rightarrow c \notin D_n$

*Proof.* We construct a proof by induction. We show that the properties hold for $P_n \cup A_n = \emptyset$ and show for each action that the properties hold after applying it on arbitrary sets.

(i) We assume no knowledge about the database; $P_n \cup A_n = \emptyset$. Theorem 1.i.a holds since the right side of the arrow is always true since $A_n$ is empty, similarly for 1.i.b. Theorem 1.ii.a holds since $P_n$ is empty so the right side of the arrow never needs to be evaluated, similarly for 1.ii.b.

(ii) We assume arbitrary sets $P_n$ and $A_n$ adhering to the requirements above. For each operation we now show that the properties hold after applying the operation.

(a) *Insert:* After inserting $c$, 1.i.a holds since $c$ is removed from $A_n$. 1.i.b holds since $D_n$ contains $c$. 1.ii.a holds since both $P_n$ and $D_n$ contain $c$. 1.ii.b holds since $A_n$ does not contain $c$.

(b) *Remove:* After removing $c$, 1.i.a holds since $D_n$ does not contain $c$. 1.i.b holds since $c$ is removed from $P_n$. 1.ii.a holds since $P_n$ does not contain $c$. 1.ii.b holds since $D_n$ does not contain $c$.

(c) *Positive Query:* After a positive query for $c$, 1.i.a holds since we assert that $A_n$ does not contain $c$. 1.i.b holds since $D_n$ contains $c$. 1.ii.a holds since we add $c$ to $P_n$. 1.ii.b holds since $A_n$ does not contain $c$.

(d) *Negative Query:* After a negative query for $c$ 1.i.a holds since $D_n$ does not contain $c$. 1.i.b holds since we assert that $P_n$ does not contain $c$. 1.ii.a holds since $P_n$ does not contain $c$. 1.ii.b holds since $D_n$ does not contain $c$.

□

**Theorem 2.** *For any operation performed on* $D_n$*,* $P_n$*, and* $A_n$*, yielding* $D'_n$*,* $P'_n$*, and* $A'_n$*, the captured info in* $P'_n$ *and* $A'_n$ *is more precise, i.e.*

$$P_n \cup A_n \subseteq P'_n \cup A'_n.$$

*Proof.* The proof is easy to see. When ever we remove an element from either $P_n$ or $A_n$ (during insert and remove operations) we always add the same element to the other set. This means the union of the sets can never shrink. During query operations we add an element to one of the sets and don't remove anything, meaning the union grows. □

**Corollary 1.** *Once $P_n$ and $A_n$ capture the entire knowledge of $D_n$, i.e. $P_n \cup A_n = \mathbb{C}$, performing operations will always keep the property*

$$P_n \cup A_n = \mathbb{C}$$

Since the knowledge can never shrink, once we have reached maximum knowledge we will stay at maximum knowledge.

# 5 Extended Finite-State Machines

Before introducing the novel FSMs we present EFSMs [101] on which DBFSM and PAFSM are based. An EFSM is an FSM extended with internal variables.

**Definition 1.** *An* extended finite-state machine *is a 7-tuple* $(Q, q_0, \Sigma, \Gamma, V, \psi, \delta)$, *where:*

- $Q$ *is a finite, non-empty* set of states.
- $q_0 \in Q$ *is the* initial state.
- $\Sigma$ *is the* input alphabet, *a non-empty finite set of labels.*
- $\Gamma$ *is the* output alphabet, *a non-empty finite set of labels.*
- $V$ *is a finite* set of variable names.
- $\psi \subset V \times Int$ *assigns integer values to the variables.*
- $\delta$ *is a* state transition relation.

*$\delta$ relates a source state $q$, and an action $a$, to a target state $q'$, given the current state of variables, $\psi$. This is written: $q \xrightarrow{a} q'$, and corresponds to a* transition *in the system. There are five types of actions:*

- *inputs,* $\sigma \in \Sigma$
- *outputs,* $\gamma \in \Gamma$
- *the null action,* $\tau$
- *boolean conditions*
- *variable updates*

*A boolean condition action is only enabled if the condition evaluates to true. Boolean conditions and variable updates may use regular arithmetic and relational operators.*

# 6 Database FSM

A *database finite-state machine* (DBFSM) is an EFSM where variables can have a type we call *database*. The database type has three operations: insert, remove, and query for membership, corresponding to the same operations on a real database. In the context of the present-absent abstraction, DBFSM should be seen as a system containing a real database.

**Definition 2.** *A* database finite-state machine *is a 8-tuple* $(Q, q_0, \Sigma, \Gamma, V, \psi, D, \delta)$*, where:*

- $Q$*,* $q_0$*,* $\Sigma$*,* $\Gamma$*,* $V$*,* $\psi$*, and* $\delta$ *are defined as for EFSM.*
- $D$ *is a* set of databases*.*

*A database can hold an infinite amount of values from variables. Three functions are defined for operating on databases:* $Insert(d, v)$*,* $Remove(d, v)$*,* $Query(d, v)$*, where* $d \in D, v \in V$*.* $Insert(d, v)$ *inserts the value of* $v$ *into database* $d$*,* $Remove(d, v)$ *removes the value of* $v$ *from database* $d$*, and* $Query(d, v)$ *returns a boolean, being true if the values of* $v$ *is present in the database and* $false$ *otherwise.*

This formalism gives us a convenient way to model systems using databases. However DBFSMs are infinite state systems and therefore not suited for modeling and testing.

# 7 Present-Absent FSM

We now introduce PAFSM which are an abstraction of DBFSM. The abstraction is done according to the present-absent abstraction.

**Definition 3.** *A* present-absent finite-state machine *is a 9-tuple* $(Q, q_0, \Sigma, \Gamma, V, \psi, DP, DA, \delta)$*, where:*

- $Q$*,* $q_0$*,* $\Sigma$*,* $\Gamma$*,* $V$*, and* $\delta$ *are defined as for DBFSM.*
- $\psi \subset V \times Values$ *assigns integer values to the variables.* $Values$ *is a finite set of integer values.*
- $DP$ *is a* set of sets*, each of size* $|Values|$*, representing the present sets. One for each database in the DBFSM.*
- $DA$ *is a* set of sets*, each of size* $|Values|$*, representing the absent set. One for each database in the DBFSM.*

$DP(d)$ *represents the present set for database* $d$*,* $DA(d)$ *represents the absent set for database* $d$*.*

This abstraction allows us to abstract over an infinite set of databases with a small set of sets. Additionally the PAFSM is finite-state, which enables straight forward state-space exploration. The requirement for integer values can easily be lifted to any value. Additionally the restriction is not a problem in practice, since the integer values can be translated into real database properties in the adapter prior to sending them to the IUT.

# 8 Translation

We now present the translation from DBFSM to PAFSM. Two translations are presented, they differ in the way unknown values are handled. The first translation assumes full knowledge of the database, and enters an error state if at any time an unknown value is observed. The second assumes no knowledge and is allowed to nondeterministically choose whether an unknown value should be treated as present or absent.

The DBFSM and PAFSM are the same in every aspect except transitions using one of the three operations on databases; insert, remove, and query. For the two translation it is explained who these are handled.

## 8.1 No Knowledge

This translation assumes no knowledge about the database, i.e. $P_n \cup A_n = \emptyset$. This is suited for online testing where the knowledge of the database can be derived during test execution. This translation can also be used to generate abstract traces, or trees, where branches in the tree correspond to choices in the model. This way offline test cases can be generated.

### Insert

If the value is in the present set this transition has no effect. If the value is in the absent set, it is added to the present set and removed from the absent set. If the value is in neither present nor absent the value is added to the present set.

### Remove

If the value is in the present set, it is removed from the present set and added to the absent set. If the value is in absent set this transition has no effect. If the value is in neither present nor absent the value is added to the absent set.

### Query

If the value is in the present set, return true. If the value is in the absent set return false. If the value is in neither, non-deterministically choose true or false and add the value to the corresponding set.

This translation is conforming to the present-absent abstraction, in that each action updates the sets according to the abstraction. When using this translation for online testing, the non-deterministic choices allow the model-checker to be in both states at the same time, and reduce the state space when observations from the IUT reveal which choice was correct. When trees are generated for offline testing the tester can traverse the tree and follow branches according to the output observed. How the non-determinism is handled in practice is shown in more detail in the concrete example in Section 10.

## 8.2 Full Knowledge

This translation requires full knowledge about the database, i.e. $P_n \cup A_n = \mathbb{C}$. This translation is basically the same as above, except we remove the unknown aspect of the three-valued-logic. This translation is suited for offline testing, where complete and static traces need to be generated to simplify the test execution.

### Insert

Since we have full knowledge about the database we know that the value is either in present or absent and never in both. Taking the transition adds the value to the present set and removes it from the absent set.

### Remove

Taking the transition adds it to the absent set and removes it from the present set.

### Query

If the value is in the present set we return true, otherwise return false. We do not need to consult the absent set since we have eliminated the unknown factor.

This translation also conforms to the present-absent abstraction. Since we start with maximum knowledge about the database we know that all values are in either the present or the absent set. From Corollary 1 we know that we never lose knowledge. This enables us to simplify the query operations. This translation is specifically well suited for offline testing where static traces are required.

There are two issues using this approach: It requires the state of the database to be known a priory and it requires the test-case generation to be re-executed prior to each execution of the test suite (not for each test case or test purpose, only for the entire test suite.) The state of the database only needs to be checked before executing the test suite the first time, since the state after executing the test suite can be stored and used as input for the next execution. The requirement to re-execute the test-case generation can be a major problem. The model checking and test-case generation might take a long time to complete,

and requiring this for each test-suite execution might significantly increase the execution time required to execute the test.

To alleviate this problem it might be possible to generate a strategy to bring the databases into a specific known state. This way test cases can be generated with this state as starting point, and at the end of test execution the database is returned to the desired state.

# 9 Advantages

There are several advantages using the present-absent abstraction over using databases. Initially the state-space is reduced considerably compared to modeling the entire database.

Traditional testing of databases require the database to be in a specific state when beginning the test, and require the tests to be executed in a specific order, to ensure the database is always in a known state. Using the present-absent sets and MBT, we can enter a subset of the state of the database into the present and absent sets, then rerun the test case generation, based on the current state of the database. This way we can abstract away from the initial state of the database, and still get automatic testing.

Traditionally testing is not performed on the system in actual use, since the test cases can interact arbitrarily with the actual database. By proving correctness on the present-absent sets, and proving that the test cases will only interact with the specific test data, the tests can be executed on the actual running system. By observing the state of the database the state of the system can be entered into the sets, and the tests can be executed, only affecting the test data in the database.

During online testing it is possible to start the testing process without any knowledge about the database, i.e. $P_n \cup A_n = \emptyset$. This way the state of the database can be dynamically learned by observing the system. As more knowledge is gained, the state space is reduced, and the testing can be guided in the desired direction.

# 10 Example

To illustrate the abstraction and test-case generation, an example is presented. This example is manufactured by hand since no tool support has been developed yet. The specification of the IUT is a network of timed automata in UPPAAL syntax. Three network of timed automata are presented: one modeling the system using databases, one translation assuming no knowledge and one assuming full knowledge.

The example is a simple system where users can login and perform some work. In the system we have a single database, consisting of the users which are currently logged in. The users have three actions: *login*, *logout*, and *work*. The user can only login if he has not already logged in. He can only logout if

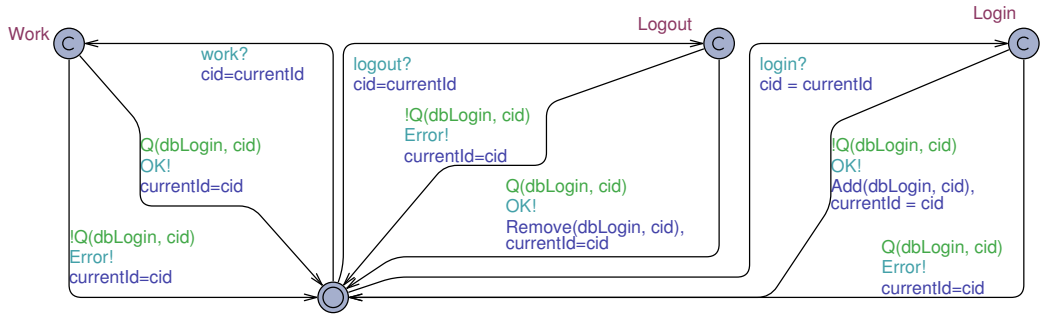


Figure 2: Example using databases

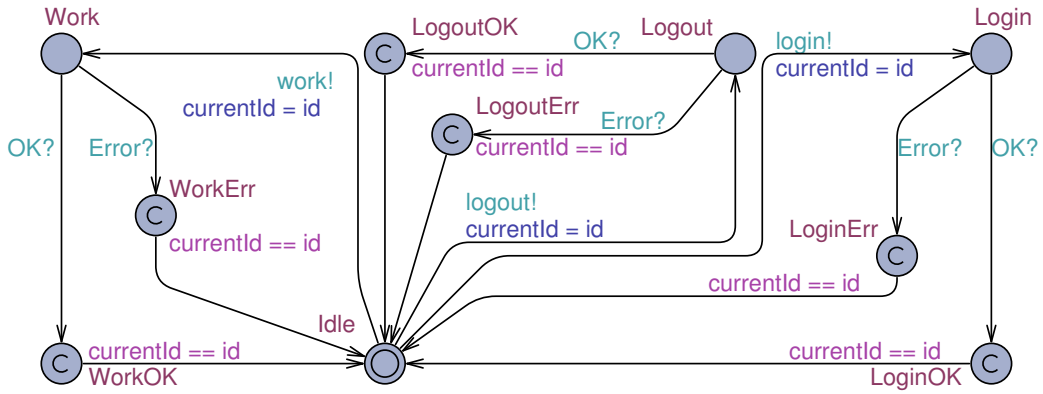


Figure 3: The user

he is logged in. The work can only be requested if he is logged in. When the user performs an action the system will return either *OK* or *Error*.

Figure 2 illustrates the system using databases. This is used as the specification of the IUT. The specification is a timed automaton implemented in Uppaal. The `Q` method queries the login database, called `dbLogin`, and returns true if `cid` is in the database.

The system has three input channels: `work?`, `login?`, and `logout?`, and two output channels: `OK!` and `Error!`. The input channels are use by the user to query the system, the output channels are used to return to the user. The shared variable `currentId` is used to pass the id of the calling user to the system, and used by the user to ensure the result is returned to the correct user. The `Add` and `Remove` methods are used to add and remove `cid` to and from the database respectively.

Figure 3 illustrates the user of the system. This is an unintelligent user which presses all buttons in random order. Another type of user is one which follows

the specification of the system. In this example such a user would for instance only try to request work if he knew he was logged in. The locations `WorkOK` and `WorkErr` are dummy locations used to specify test purposes (similar for login and logout). For instance to test whether a user is able to request work and get a positive response, UPPAAL is asked for a trace where the user template enters the `WorkOK` location. The same user is used in all the examples.

We now explain how the model is translated using the two approaches explained above. We also explain how these model can be used for testing.

## 10.1 No Knowledge

Figure 4 illustrates the system where no knowledge is assumed. Each query to the database is translated into a call to the method `IsLogin`, which has three possible outcomes: `TRUE`, `FALSE`, and `UNKNOWN`, corresponding to the three valued logic in the abstraction. `IsLogin` consults the present and absent sets and returns based on the values. If the value is unknown a non-deterministic choice is available, either return `OK` and add the user to the present set, or return `Error` and remove the user from the set. The methods `Login` and `Logout` handle this. This has the effect of updating the database when the correct choice is observed from the IUT. Notice that when the result of a login action is unknown both the OK and the Error choice add the user. This is because, returning Error means the user is in the database, therefore we add him. If we return OK the user is not in the database, and we should remove him, however, since the login was successful the user is now added to the database, therefore we add him.

It can be seen that we do not make any consistency check on the present-absent sets, i.e. check for $P_n \cap A_n \neq \emptyset$. This is because we can verify that this can never be the case using the model checker with the following query:

```
A[] forall (id:UserID) !(Present(id) && Absent(id))
```

This query states: It is always the case that no user is in present and absent at the same time. If this query verifies there can occur no inconsistencies in the model.

This system starts with both sets empty. Whenever a choice is taken the sets are updated accordingly. If an online test is executed with this system as the specification, UPPAAL would take both choices and keep track of two states in the system. When the actual action is observed from the IUT all nonconforming states are discarded. By running this system in the simulator in UPPAAL we can simulate an online test where UPPAAL makes the choice for the IUT. It can be seen that after executing for a while we reach a situation where we have full knowledge about the database, i.e. $P_n \cup A_n = \mathbb{C}$.

This system has been tested against an implementation using UPPAAL TRON. The systems was instantiated with ten users. A mutant is made, in which the login action has a 1/500 chance to fail to update the database. The system has been implemented such that the database is filled with random values at initialization. This way the tester has no way of knowing the state of the database

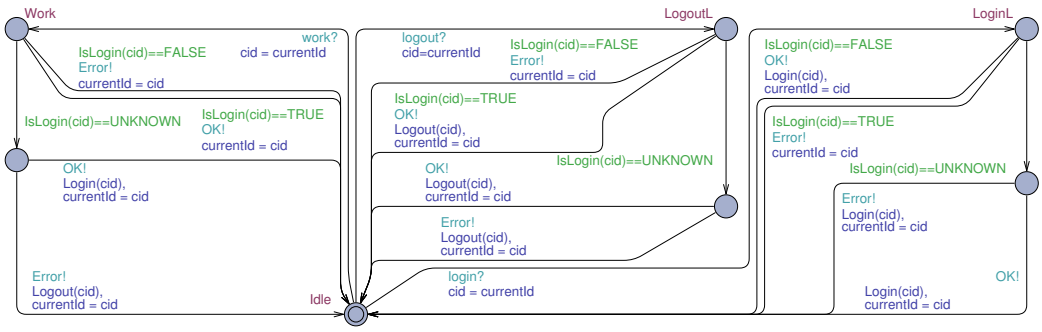


Figure 4: Example with no knowledge

when starting the test. The test was run ten times on the correct implementation and ten times on the mutant. Each successful test executed about 22.000 action (input and output combined). One of the mutant runs failed to detect the mutant, this is due to the randomness of the mutant. The tests have shown us that the present-absent set approach has the capabilities to automatically test a system which interacts with a database without knowledge about the state of this database prior to testing.

We can also use this translation to generate abstract traces. A tree will be generated for each test purpose. A branch in the tree corresponds to a choice in the model. UPPAAL COVER can be used to generate these trees.

## 10.2 Full Knowledge

Figure 5 illustrates the system with full knowledge assumed. This system is similar to the system with no knowledge. Since we have full knowledge we can remove all transitions where the database state is unknown. This simplifies the model. This model is useful for generating static traces to be used for offline testing, by a static testing tool. To generate the traces the UPPAAL model checker can be asked whether a template can reach a specific location and get a trace of how to reach this location. This trace can be used as a test case.

We use the dummy locations in the user template, Figure 3. The following query is used to test if the user with ID 0 can successfully login:

```
E<> Users(0).LoginOK
```

Verifying this with all users absent generates the following trace:

```
Users.Login[0]!
IUT.LoginOK[0]!
```

Meaning: First user with ID 0 sends a login request to the IUT, then the IUT sends `loginOK` to user 0. To show that we can generate different traces

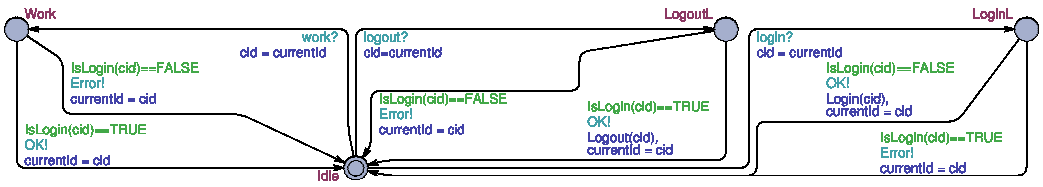


Figure 5: Example with full knowledge

depending on the state of the database, we run the verifier again with all users present in the database. This generates the following trace:

```
Users.Logout[0]!
IUT.LogoutOK[0]!
Users.Login[0]!
IUT.LoginOK[0]!
```

Here we can see that the user first has to log out, before he can log in successfully. This shows that by updating the state of the database we can generate static traces which conform to the state of the database.

This simplistic example serves to illustrate the abstraction, but it is not representative of a realistic database system. Extending the user table in the database with properties can easily be achieved by creating present and absent sets for each property. This approach can also be extended to include relations between tables. A one-to-many relation can be modeled using present and absent sets for each entry in the *one* relation, these sets can hold the values which the corresponding entry relates to. How this preforms in practice needs to be analyzed in future work.

# 11 Conclusion

We have introduced the abstraction from a database into present and absent sets and a proof of this abstraction. We have introduced two new forms of FSMs, the DBFSM and PAFSM and explained how to translate from DBFSM to PAFSM. Furthermore, we have explained two different translations and how these relate to the present-absent abstraction.

We have illustrated an example of a simple system using a database, and how this system can be translated into a system using present and absent sets. We have explained how test cases can be generated from this system, as well as the benefits of using our approach when performing online and offline testing.

We are able to perform online testing of systems without taking any assumptions about the state of the database into account. As the test progresses, we gradually gain more knowledge about the state of the database. This increase of

knowledge will reduce the state space of the simulation model, as well as enable us to potentially guide the testing in a desired direction.

We enable two forms of offline testing. One without assuming any knowledge about the state of the database. We are able to generate abstract traces which automatically learn the state of the database and make choices accordingly to reach the desired state. By examining the state of the database prior to generating the test cases, we are able to generate static traces which can be executed without any branching. This removes the problem of state dependency when performing offline testing on database systems. There are some potential performance issues with this approach, but we are hopeful as to finding a solution to these problems.

As future work we plan to extend the simplistic view of database presented in this paper. We plan to measure the effectiveness of this approach on larger examples, preferably industrial. We are currently working on extending the Uppaal model checker to improve the effectiveness of model checking systems using present and absent sets.

# Paper B

# Model Checking with Lattices

Andreas Engelbredt Dalsgaard
René Rydhof Hansen
Kim Gulstrand Larsen
Mads Chr. Olesen
Petur Olsen
Jiří Srba

*Department of Computer Science, Aalborg University,*
*Selma Lagerlöfs Vej 300, DK-9220 Aalborg East, Denmark.*
{andrease,rrh,kgl,mchro,petur,srba}@cs.aau.dk

**Abstract** Model checking is a verification technique based on searching through a state space. As the state spaces are often large or even infinite due to unbounded data structures, techniques to tackle this problem have been investigated. We introduce a novel abstract model of labelled transition systems where states of the system are paired with elements from a lattice that provides a suitable abstraction of the real data. We present a general reachability algorithm with different update functions to account for different degrees of abstraction and prove the correctness and termination of the algorithm. Furthermore, we develop the notion of Lattice Guided Abstraction Refinement (LaGAR) for iterative recovery of precision that might be lost due to the applied abstraction. The usability of the framework is demonstrated on a number of applications that include communication protocols, databases, cache analysis and zones in timed automata. Prototype implementations in some of the application domains indicate promising results.

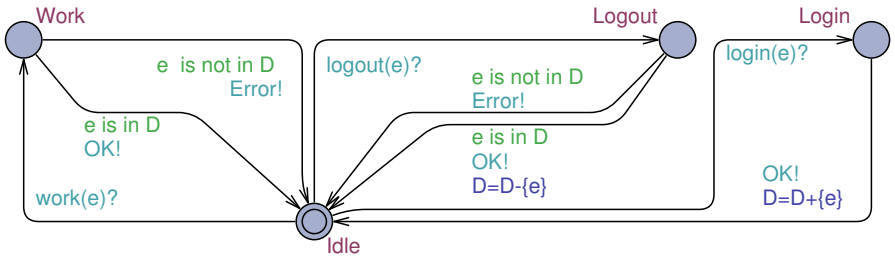


Figure 1: A small database system handling login, logout and work requests

# 1 Introduction

Over the past two decades, model checking has been applied with great success to validation and verification in a wide variety of areas, including hardware and software designs, control programs and critical safety and security properties of communication protocols. More recent applications include schedulability [25] and worst-case execution time (WCET) [35] analysis of embedded real-time systems as well as test-case generation for graphical user interfaces. This diversity of application areas is reflected in the availability of many different model checkers and special purpose logics inspired by particular application areas.

Common to almost all of these applications of model checking is the notion of an underlying concrete system with a very large—or sometimes even infinite—concrete state space. To enable model checking of such systems it is necessary to construct an abstract model of the concrete system, where some system features are only modelled approximately and system features that are irrelevant for given verification purposes are "abstracted away".

One feature that is often abstracted away is that of data. Many important systems such as communication protocols, memories, bounded buffers, queues, and databases operate on data values from a potentially infinite domain: an input operation can result in an infinite number of different states corresponding to the infinitely many data values. Traditionally, model checking has completely ignored the data part and opted for verification of only the control aspect (this is usually finite-state) and ignore the transfer of data. Although systems of the above type do operate on data values from a potentially infinite set of data, they often never perform any computation on these data values. Rather, such systems only input, store, compare, copy or output values as illustrated by the small database system in Figure 1, where the component $D$ is intended to hold the set of user-names currently logged in.

In fact, for such data-insensitive systems various behavioural properties may possibly be settled by only tracking a few representative data values (see the seminal work by Jonsson and Parrow [59]), thus leading to a significant reduction in the size of the state space to be considered. As an example the database system of Figure 1 may be abstracted based on a small set of selected user-names $\mathcal{S}$. In this over-approximate abstraction the database component $D$ is replaced

by two subsets $P$ and $A$ of $\mathcal{S}$ representing sets of user-names known to be present respectively absent in the database, and transitions are enabled provided their guards are not in conflict with this knowledge of present and absent user-names[1]. Ordered by component-wise reverse set-inclusion, the pairs $(P, A)$ constitute a (finite) complete lattice, where less knowledge about the database (i.e. smaller sets) allow for more behaviour in the abstraction.

Ordering abstract data values in this manner is very similar to the use of lattices as found in the area of program analysis in general, and abstract interpretation in particular [74], where the systematic construction of (sound) abstract models from concrete systems has been studied extensively. Here lattices, and operations on lattices, are used as the fundamental abstraction mechanism that facilitates the design and development of hierarchies of abstractions exploring the trade-off between precision of the model and the cost of analysis/verification.

In this paper we define and discuss a new formalism combining lattices and model-checking, *lattice transition systems (LaTS)*, specifically designed to be well-suited for modelling, model checking, and reasoning about abstract models as well the relation to the underlying concrete systems. Lattice transition systems extend labelled transition systems with elements drawn from a lattice that gives an abstract representation of (parts of) the concrete system. In addition to the formalism itself, we present a general reachability model checking algorithm for lattice transition systems. The general model checking algorithm can be instantiated with different update-algorithms allowing for varying degrees of approximation.

The approximative nature of abstract models may cause inconclusive verification results during model checking, i.e., it is inconclusive if it is not possible to realise the abstract error-trace in the underlying concrete system. One well-known approach to dealing with the problem of inconclusive results is *Counter Example Guided Abstraction Refinement* (CEGAR) [52, 17] where an inconclusive result (a counter example) is used to make (the pertinent parts of) the abstract model sufficiently concrete that a conclusive result is reached. We present a similar approach for lattice transition systems, called *Lattice Guided Abstraction Refinement* (LaGAR) where the structure of the lattices in the abstract model is used to guide and control the refinement (concretisation) of the abstract model.

We finish by presenting a variety of applications of the framework developed in this paper, including asynchronous communication protocols, present/absent technique for databases, abstract caches and zones in timed automata.

## 2 Lattice Transition Systems

In this section we define *lattice transition systems*, a model we will use in the rest of the paper. We start by briefly reviewing the basic lattice and order theoretic notions and definitions. For a more thorough treatment see e.g. [37].

[1] Dually, one may aim at an under-approximation, where transitions are enabled only if their guards can be determined satisfied by the knowledge of $P$ and $A$.

## 2.1 Preliminaries

A *partial order* on a set $L$ is a reflexive, anti-symmetric and transitive relation $\sqsubseteq \subseteq L \times L$. The pair $(L, \sqsubseteq)$ is called a *partially ordered set*. Let $(L, \sqsubseteq)$ be a partially ordered set and let $X \subseteq L$. An element $\ell \in L$ is an *upper bound* of $X$ if $x \sqsubseteq \ell$ for every $x \in X$. If furthermore $\ell \sqsubseteq \ell'$ for all upper bounds $\ell'$ of $X$ then $\ell$ is the *least upper bound* of $X$ and is denoted as $\bigsqcup X$. The binary least upper bound $\bigsqcup\{x, y\}$ is written as $x \sqcup y$. The notion of *lower bounds* of $X$ and the *greatest lower bound* of $X$ (denoted by $\bigsqcap X$ if it exists) are defined analogously. The binary greatest lower bound $\bigsqcap\{x, y\}$ is written as $x \sqcap y$. Note that least upper bounds and greatest lower bounds for a given set $X$ are unique.

An element $\ell \in L$ such that $\ell \sqsubseteq \ell'$ for all $\ell' \in L$ is called the *least element* of $(L, \sqsubseteq)$ and is denoted as $\bot_L$ or $\bot$ when $L$ is clear from the context. Conversely, the *greatest element* of $(L, \sqsubseteq)$ is an element $\ell \in L$ such that $\ell' \sqsubseteq \ell$ for all $\ell' \in L$ and is denoted as $\top_L$ or $\top$ when $L$ is clear from the context.

**Definition 1** (Join Semi-Lattice and Lattice)**.** *A partially ordered set $\mathcal{L} = (L, \sqsubseteq)$ where $L \neq \emptyset$ is a* join semi-lattice *if $\ell \sqcup \ell'$ exists for all $\ell, \ell' \in L$. If moreover $\ell \sqcap \ell'$ exists for all $\ell, \ell' \in L$ then $\mathcal{L}$ is called a* lattice*.*

In this paper we consider only *finite* lattices in order to ensure the termination of the algorithms.

*Example 1.* Consider the lattice for the abstraction of a database into present and absent sets mentioned earlier. Let $D \subseteq \mathcal{U}$ be a database containing elements from some universe $\mathcal{U}$ of possible values. As an abstraction over this universe we select a small set of representative values, $\mathcal{S} \subseteq \mathcal{U}$. We present two subsets of $\mathcal{S}$ the *present set* $P$, and the *absent set* $A$. Adapting terminology from the abstract interpretation community, this forms a *must* analysis, that is, a value in the present set *must* be in the database, and a value in the absent set *may not* be in the database.

The set of databases abstracted over with a concrete set of present and absent sets is the semantics of $(P, A)$: $[\![(P, A)]\!] = \{D \subseteq \mathcal{U} | P \subseteq D \cap \mathcal{S} \wedge A \subseteq \mathcal{S} \backslash D\}$. For instance the present-absent sets $(\{e\}, \{f\})$ abstract over all databases which contain $e$ while not containing $f$. These present-absent sets can be ordered in a lattice, where smaller sets abstract over more databases, and therefore give more behaviour in the model. This lattice is defined as $((\mathcal{S} \times \mathcal{S}), \sqsubseteq)$ where $(P, A) \sqsubseteq (P', A') \iff P' \subseteq P \wedge A' \subseteq A$ and $(P, A) \sqcup (P', A') = (P \cap P', A \cap A')$. An example of this lattice with $S = \{e\}$ is given in Figure 2.

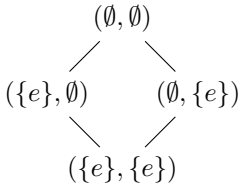


Figure 2: Lattice example

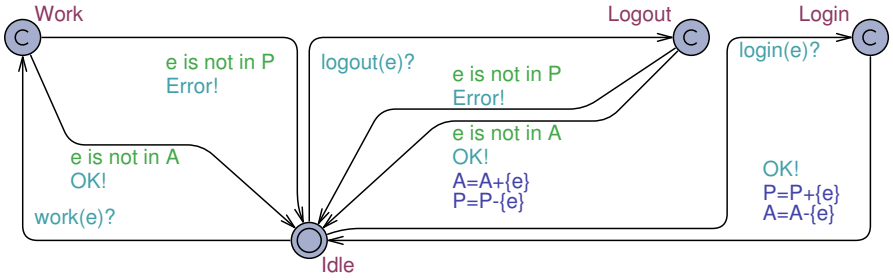


Figure 3: Abstraction of the database system using present and absent sets.

## 2.2 Lattice Transition System

**Definition 2** (Lattice Transition System)**.** *A* lattice transition system *(LaTS) is a triple* $\mathcal{T} = (S, \mathcal{L}, \longrightarrow)$ *where* $S$ *is a finite set of states,* $\mathcal{L} = (L, \sqsubseteq)$ *is a (finite) lattice and* $\longrightarrow \subseteq S \times L \times S \times L$ *is the transition relation, usually written as* $(s, \ell) \longrightarrow (s', \ell')$ *whenever* $(s, \ell, s', \ell') \in \longrightarrow$*, such that for all* $s_1, s_2 \in S$ *and* $\ell_1, \ell_2, \ell_1' \in L$ *if* $(s_1, \ell_1) \longrightarrow (s_2, \ell_2)$ *and* $\ell_1 \sqsubseteq \ell_1'$ *then* $(s_1, \ell_1') \longrightarrow (s_2, \ell_2')$ *for some* $\ell_2' \in L$ *with* $\ell_2 \sqsubseteq \ell_2'$*.*

The behavioural requirement used at the end of the definition is called the *monotonicity property.* Configurations of an LaTS are pairs of the form $(s, \ell)$ where $s \in S$ and $\ell \in L$ and $\longrightarrow^*$ denotes the reflexive and transitive closure of $\longrightarrow$.

**Definition 3** (Path)**.** *A finite path in an LaTS* $\mathcal{T}$ *is a finite sequence* $\sigma = (s_0, \ell_0)(s_1, \ell_1) \cdots (s_n, \ell_n)$ *such that* $(s_i, \ell_i) \longrightarrow (s_{i+1}, \ell_{i+1})$ *for all* $i$*,* $0 \leq i \leq n-1$*.*

In addition to the standard notion of path we also define *abstracted paths.*

**Definition 4** (Abstracted Path)**.** *An abstracted finite path in an LaTS* $\mathcal{T}$ *is a finite sequence* $\hat{\sigma} = (s_0, \ell_0)(s_1, \ell_1) \cdots (s_n, \ell_n)$ *such that* $\exists \ell_{i+1}' \in L : (s_i, \ell_i) \longrightarrow (s_{i+1}, \ell_{i+1}')$ *and* $\ell_{i+1}' \sqsubseteq \ell_{i+1}$ *for all* $i$*,* $0 \leq i \leq n-1$*.*

In the section to follow, we find an efficient way to answer the following question (state-reachability problem): given an initial configuration $(s_0, \ell_0)$ and a target state $s_g$, is there some lattice element $\ell$ such that $(s_0, \ell_0) \longrightarrow^* (s_g, \ell)$?

*Example 2.* We have seen how a database can be abstracted into present and absent sets in Example 1. We will now present an LaTS which produces over-approximate behaviour with respect to a concrete database system. The abstract system is presented in Figure 3. The queries and updates of the database have been changed to querying and updating the present and absent sets. For example `e is in D` is changed to querying whether `e` may be present, i.e., `e is not in A`.

The initial configuration of the system without any knowledge of the database is $(Idle, (\emptyset, \emptyset))$. Assuming we have at least two users: $e$ and $f$. Logging in with

**Algorithm 2**: **Reach**$(\mathcal{T}, (s_0, \ell_0), s_g)$

**Input**: LaTS $\mathcal{T} = (S, \mathcal{L}, \longrightarrow)$, initial configuration $(s_0, \ell_0)$, a goal state $s_g \in S$
**Output**: "$s_g$ is reachable" or "$s_g$ is not reachable"

```
1:  waiting := {(s_0, ℓ_0)}
2:  passed := ∅
3:  while waiting ≠ ∅ do
4:     Select and remove (s, ℓ) from waiting
5:     passed := passed ∪ {(s, ℓ)}
6:     for all (s', ℓ'), where (s, ℓ) ⟶ (s', ℓ') do
7:        if s' = s_g then
8:           return "s_g is reachable"
9:        end if
10:       waiting := Update(passed, waiting, (s', ℓ'))
11:    end for
12: end while
13: return "s_g is not reachable"
```

**Algorithm 3**: **Update**$(passed, waiting, (s, \ell))$ *** Simple Update ***

**Input**: Sets of states *passed* and *waiting* and a configuration $(s, \ell)$
**Output**: Updated set *waiting*

```
1: if ∃(s, ℓ) ∈ waiting ∪ passed then
2:    return waiting
3: else
4:    return waiting ∪ {(s, ℓ)}
5: end if
```

$e$ gives the following path: $(Idle, (\emptyset, \emptyset)) \rightarrow (Login, (\emptyset, \emptyset)) \rightarrow (Idle, (\{e\}, \emptyset))$. Performing a login with user $f$ produces the following steps: $(Idle, (\{e\}, \emptyset)) \rightarrow (Login, (\{e\}, \emptyset)) \rightarrow (Idle, (\{e, f\}, \emptyset))$. We can also generate an abstract path: $(Idle, (\emptyset, \emptyset)) \rightarrow (Login, (\emptyset, \emptyset)) \rightarrow (Idle, (\emptyset, \emptyset))$, since $(\{e\}, \emptyset) \sqsubseteq (\emptyset, \emptyset)$, because the unknown database has at least as much behaviour as the database with $e$ logged in.

# 3 General Model Checking Algorithm

In Algorithm 2 we present the pseudocode for a general model checking reachability algorithm. The algorithm explores the graph using *waiting* and *passed* sets and depending on the chosen update function performs a different level of abstraction.

**Lemma 1.** *Algorithm 2 terminates for any of the update functions given in Algorithms 3, 4 and 5.*

**Algorithm 4**: **Update**$(passed, waiting, (s, \ell))$ *** Cover Update ***

**Input**: Sets of states *passed* and *waiting* and a configuration $(s, \ell)$
**Output**: Updated set *waiting*

```
1: if ∃(s, ℓ') ∈ waiting ∪ passed : ℓ ⊑ ℓ' then
2:    return waiting
3: else
4:    return waiting ∖ {(s, ℓ'') | ℓ'' ⊑ ℓ} ∪ {(s, ℓ)}
5: end if
```

*Proof.* Termination follows from the fact that once a pair is removed from the *waiting* list it is never again added to the *waiting* list. This can be easily verified by inspecting all three update functions that never add any $(s, \ell)$ to the *waiting* list if this pair is already present in the *passed* list. This, together with the assumption that the number of states as well as the number of lattice elements are finite sets, implies termination. □

It is easy to realize that Algorithm 2 with the simple update implements a search though the whole state space of the lattice transition system. Depending on the way the *waiting* list is organized, it can implement e.g. depth-first search, breath-first rearch, random search and others. The correctness of this search algorithm is well known.

We shall now argue that the monotonicity property is enough to prove the soundness and completeness also for our algorithm with the cover update.

**Theorem 1** (Correctness of Algorithm 2 with Cover Update)**.** *Let $\mathcal{T} = (S, \mathcal{L}, \longrightarrow)$ where $\mathcal{L} = (L, \sqsubseteq)$ be a lattice transition system. Let $(s_0, \ell_0)$ be its initial configuration such that $s_0 \in S$ and $\ell_0 \in L$ and let $s_g \in S$. The call* **Reach**$((s_0, \ell_0), s_g)$ *of Algorithm 2 using the cover update in Algorithm 4 returns "$s_g$ is reachable" if and only if there is some $\ell \in L$ such that $(s_0, \ell_0) \longrightarrow^* (s_g, \ell)$.*

*Proof.* "⇒": We want to show that if the algorithm returns "$s_g$ is reachable" then $(s_0, \ell_0) \longrightarrow^* (s_g, \ell)$ for some $\ell \in L$. This fact can be derived by observing the validity of the following loop invariant: "whenever $(s, \ell) \in$ *waiting* then $(s_0, \ell_0) \longrightarrow^* (s, \ell)$". The invariant is surely true before entering the while loop of Algorithm 2 (because initially *waiting* $= \{s_0, \ell_0\}$) and it is preserved during the execution of the algorithm as the set *waiting* can only contain the pairs of the form $(s', \ell')$ where $(s, \ell) \longrightarrow (s', \ell')$ for some $(s, \ell)$ already present in the *waiting* set (and hence satisfies the invariant).

"⇐": We will show by induction on $n$ that if $(s_0, \ell_0) \longrightarrow^n (s_n, \ell_n)$ then the pair $(s_n, \ell'_n)$ for some $\ell'_n$ such that $\ell_n \sqsubseteq \ell'_n$ will eventually appear in the waiting set (unless the state $s_g$ was discovered before that). The base case $n = 0$ is trivial. Assume that the claims holds for some $n > 0$ and consider a computation $(s_0, \ell_0) \longrightarrow^n (s_n, \ell_n) \longrightarrow (s_{n+1}, \ell_{n+1})$. By induction hypothesis the pair $(s_n, \ell'_n)$ where $\ell_n \sqsubseteq \ell'_n$ will eventually appear in the *waiting* set. The definition of the cover update function guarantees that a pair $(s_n, \ell''_n)$ for some

**Algorithm 5**: **Update**$(passed, waiting, (s, \ell))$ *** Join Update ***

**Input**: Sets of states *passed* and *waiting* and a configuration $(s, \ell)$
**Output**: Updated set *waiting*

1: **if** $\exists (s, \ell') \in waiting \cup passed : \ell \sqsubseteq \ell'$ **then**
2: **return** *waiting*
3: **else if** $\exists (s, \ell') \in waiting \cup passed$ **then**
4: **return** $waiting \setminus \{(s, \ell')\} \cup \{(s, \ell' \sqcup \ell)\}$
5: **else**
6: **return** $waiting \cup \{(s, \ell)\}$
7: **end if**

$\ell''$ such that $\ell'_n \sqsubseteq \ell''_n$ will be eventually removed from the *waiting* set at line 4 of the reachability algorithm. Because $(s_n, \ell_n) \longrightarrow (s_{n+1}, \ell_{n+1})$ we get (thanks to monotonicity) that $(s_n, \ell''_n) \longrightarrow (s_{n+1}, \ell'_{n+1})$ for some $\ell'_{n+1}$ such that $\ell_{n+1} \sqsubseteq \ell'_{n+1}$. As long as $s_{n+1} \neq s_g$ the pair $(s_{n+1}, \ell'_{n+1})$ will be added to the *waiting* set, unless some pair $(s_{n+1}, \ell''_{n+1})$ with $\ell'_{n+1} \sqsubseteq \ell''_{n+1}$ is already present in the *waiting* or *passed* set (and hence it was previously present in the *waiting* set too). In any of these two cases, the induction step is established. □

We now define an alternative update function presented in Algorithms 5 that provides an overapproximation.

**Theorem 2** (Correctness of Algorithm 2 with Join Update)**.** *Let $\mathcal{T} = (S, \mathcal{L}, \longrightarrow)$ where $\mathcal{L} = (L, \sqsubseteq)$ be a lattice transition system. Let $(s_0, \ell_0)$ be its initial configuration such that $s_0 \in S$ and $\ell_0 \in L$ and let $s_g \in S$. The call* **Reach**$((s_0, \ell_0), s_g)$ *of Algorithm 2 using the join update in Algorithm 5 returns "$s_g$ is not reachable" only if there is no $\ell \in L$ such that $(s_0, \ell_0) \longrightarrow^* (s_g, \ell)$.*

*Proof.* The proof is similar to the direction "⇐" in the proof of Theorem 1. The only difference is that when adding a pair to the *waiting* set we change the lattice value to the join with the lattice value of some other already investigated element with the same state component (if it exists). The correctness follows from the fact that such join is always above the lattice value of the added pair. The rest is the same as in Theorem 1. □

Notice that the other direction in Theorem 2 does not hold in general.

*Example 3.* Consider again the database system from Example 2. We want to check that no two users can work at the same time. Using the simple update algorithm, Algorithm 2 returns the correct answer that such a state is not reachable. However, doing so explores all permutations of present and absent sets, because the database might be any of them. Using cover update will also answer correctly, but generates a much smaller state space due to exploiting the lattice ordering.

Another property we want to check is that the database can never become full. In such a setting the property cannot be proven with an initial unknown

database, since then the database might already be full. Proving the property with an initial empty database can be done using the simple update, but this explores all permutations. Cover update can also prove the property, but will not perform better, since every reachable state has one concrete database. Join update will on the other hand join lattice elements, and cannot give a conclusive answer for this query. For a system with 2 users $e, f$ it will find two states $(Init, (\{e\}, \{f\}))$ and $(Init, (\{f\}, \{e\}))$ joining them to reach the state $(Init, (\emptyset, \emptyset))$. This state abstracts over the full database, but is not reachable in the concrete system.

In Section 4 we give a method for doing a more refined reachability search, when using the join update gives inconclusive results.

# 4 Lattice Guided Abstraction Refinement

The approximative nature of model checking with join update, as described in the previous section, may result in *inconclusive* verification results such that a verification result in the abstract model may not be necessarily realizable in the underlying concrete system. In this case it may be possible to use the lattice structure to derive a more precise approximation that avoids the inconclusive verification result previously encountered.

In this section we describe such an approach to abstraction refinement, called *Lattice Guided Abstraction Refinement (LaGAR)*, inspired by the CEGAR (counter example guided abstraction refinement) principle [52, 17] The LaGAR approach depends on a few application specific heuristics: a method for determining the feasibility of a path and a method for refining an approximation given an infeasible path. These are formalised in the following.

**Definition 5** (Path feasibility function)**.** *A path feasibility function determines, in a domain-specific manner, whether an abstracted path is feasible in an LaTS* $\mathcal{T} = (S, \mathcal{L}, \longrightarrow)$*:*

$$pathfeasible : (S \times L)^* \rightarrow \{True, False\}$$

The path feasibility function usually corresponds to finding concrete lattice elements for each step in the path, i.e., a concrete path.

Some way of recording the abstractions used in the current state space exploration is needed. At its most abstract this can be viewed as an oracle, answering queries as to whether two lattice elements are allowed to be joined in a given state.

**Definition 6** (Joining oracle)**.** *A joining oracle is a function able to answer questions of the form:*

$$strategy_{joining} : S \times L \times L \rightarrow \{True, False\}$$

*given an LaTS* $\mathcal{T} = (S, \mathcal{L}, \longrightarrow)$*.*

The joining oracle can answer that at one state all lattice elements are to be joined, or no elements are to be joined, but it can also answer very selectively which lattice elements to join. In this way the oracle can exploit additional knowledge about the domain: e.g. for integer values, in some parts of the state space the exact value of a variable is needed, in other parts only the parity is relevant, and in yet other parts the signed-ness is important.

The oracle need not be perfect. It might give an approximation that leads to an abstracted path to a goal state, which is then deemed infeasible by the path feasibility function. In this situation the oracle is allowed to reconsider some of its answers, at the cost of recomputing the parts of the state space that depended on those answers: a join at some state is allowed to be split. Which state to split at is given by a state split heuristic.

**Definition 7** (State split heuristic)**.** *A state split heuristic determines, in a domain-specific manner, which state to split at, given an infeasible abstracted path in an LaTS* $\mathcal{T} = (S, \mathcal{L}, \longrightarrow)$*:*

$$h_{splitstate} : (S \times L)^* \to S$$

We can now present the complete algorithm for LaGAR exploration of a LaTS in Algorithm 6. The explorations will start from an abstraction level close to that of the join update (depending on the joining oracle) and become less abstract until it at some point becomes the same as the cover update, unless a conclusive answer is found before that. The correctness of the algorithm follows from the fact that it returns "$s_g$ is reachable" only if the path is indeed feasible (if-test at line 11) and from the correctness of the algorithm with cover update studied in the previous section.

The termination of the LaGAR algorithm depends on the behaviour of the heuristics. We will now give one set of sufficient criteria for ensuring termination of the algorithm.

**Theorem 3** (LaGAR Termination)**.** *The LaGAR algorithm terminates, if:*

1. $strategy_{joining}$*,* $pathfeasible$ *and* $h_{splitstate}$ *are computable functions.*

2. $strategy_{joining}$ *respects previous choices across splits such that it only splits further, i.e., if* $strategy_{joining}(s, \ell, \ell') = False$ *for some state* $s$ *and some lattice elements* $\ell, \ell'$*, then it holds invariantly from then on.*

3. *After a split, something is actually split, i.e., for at least one pair of lattice elements* $\ell, \ell'$ *and some state* $s$*, where before the split* $strategy_{joining}(s, \ell, \ell') = True$*, after the split* $strategy_{joining}(s, \ell, \ell') = False$*.*

*Proof.* The number of tuples $(s, \ell, \ell')$ is finite by assumption. Because of requirements 2 and 3, only finitely many splits can occur. The exploration can thus only be repeated a finite number of times.

□

**Algorithm 6**: **CEGAR**$(\mathcal{T}, (s_0, \ell_0), s_g)$

**Input**: LaTS $\mathcal{T} = (S, \mathcal{L}, \longrightarrow)$ (with path feasibility function *pathfeasible*, state split heuristic $h_{splitstate}$, and joining oracle $strategy_{joining}$ ), initial configuration $(s_0, \ell_0)$, a goal state $s_g \in S$

**Output**: "$s_g$ is reachable" or "$s_g$ is not reachable"

1: $waiting := \{(s_0, \ell_0)\}$
2: $passed := \emptyset$
3: $pred := \emptyset$ ; predecessor edges
4: **while** $waiting \neq \emptyset$ **do**
5: Select and remove $(s, \ell)$ from $waiting$
6: $passed := passed \cup \{(s, \ell)\}$
7: **for all** $(s', \ell'), where\ (s, \ell) \longrightarrow (s', \ell')$ **do**
8: $pred := pred \cup \{(s', \ell') \rightarrow (s, \ell)\}$ ; record predecessor
9: **if** $s' = s_g$ **then**
10: $\hat{\sigma} := (s_0, \ell_0) \dots (s_g, \ell')$ ; some abstracted path from $s_g$ to $s_0$ in the reverse configuration graph given by vertices $passed \cup \{(s_g, \ell')\}$ and edges $pred$
11: **if** $pathfeasible(\hat{\sigma})$ **then**
12: **return** "$s_g$ is reachable"
13: **else**
14: ; path was not feasible, due to abstraction
15: $s_{split} := h_{splitstate}(\hat{\sigma})$
16: ; redo exploration from $s_{split}$
17: $redo := \{(t, \ell'') | (s_{split}, \_) \rightarrow (t, \ell'') \in pred\}$
18: $passed := passed \setminus \{(t, \_) | t \text{ descendant of } s_{split}\}$
19: $waiting := waiting \setminus \{(t, \_) | t \text{ descendant of } s_{split}\} \cup redo$
20: $pred := pred \setminus \{(\_, \_) \rightarrow (t, \_) | t \text{ descendant of } s_{split}\}$
21: **end if**
22: **else**
23: ; add $(s', \ell')$ to waiting, possibly abstracting by joining
24: $joinelements := \{\ell'' | (s', \ell'') \in passed \cup waiting \wedge strategy_{joining}(s', \ell', \ell'') = True\}$
25: $\ell_{joined} := \ell' \sqcup (\bigsqcup joinelements)$
26: $pred := pred \cup \{(s', \ell_{joined}) \rightarrow (t, \ell''') | \ell'' \in joinelements$ s.t. $(s', \ell'') \rightarrow (t, \ell''') \in pred\}$
27: $passed := passed \setminus \{(s', \ell'') | \ell'' \in joinelements\}$
28: $waiting := waiting \cup \{(s', \ell_{joined})\} \setminus \{(s', \ell'') | \ell'' \in joinelements\}$
29: **end if**
30: **end for**
31: **end while**
32: **return** "$s_g$ is not reachable"

*Example 4.* Using the LaGAR approach on the "database not full" property from Example 3 we can give conclusive results, without resorting to full state

| Number of users | simple update | cover update |
|---|---|---|
| 2 | 224 (<1s) | 56 (<1s) |
| 3 | 2352 (2s) | 336 (<1s) |
| 4 | 21952 (28s) | 1792 (2s) |
| 5 | 192080 (8:22m) | 8960 (9s) |
| 6 | - | 43008 (48m) |
| 7 | - | 200704 (4:38m) |

Figure 4: Experimental data for the property "no two users can work at the same time".

space exploration. By employing our knowledge about the problem, we can implement a refined oracle which only allows joining of lattice elements when the outcome does not abstract over a full database. Implementing this refined oracle is in our case done by hand, but could also be deduced automatically by examining the error path found using a naïve always joining oracle.

# 5 Applications

To evaluate the applicability of the lattice model checking framework we have implemented a prototype lattice model checker in Python. With the prototype we have made a number of experiments. In this section our experiments and some of the applications of the framework will be described.

## 5.1 Present-Absent Sets

The two properties from the running example have been verified as described in Example 3 and Example 4. The results in Figure 4 illustrate that the cover update can significantly reduce the state space and the verification time, allowing verification of larger systems. The results in Figure 5 shows that join can be inconclusive, but the use of the LaGAR method can yield conclusive results while significantly reducing the state space. The results for the naïve oracle have been included to illustrate that the initial exploration is very fast, quickly finding an (infeasible) abstracted path to the error state. This path can then be used to deduce the refined oracle.

## 5.2 Protocols with Asynchronous Communication

A wide range of applications of our framework is provided by communication protocols where messages are asynchronously passed via an unreliable (lossy and duplicating) medium. As long as we are interested in safety properties, such a communication can be modelled as a set of already sent messages called *pool*. Initially the set *pool* is empty. Once a message it sent, it is added to the set *pool* and it remains there for ever (duplication). As the protocol parties are

| Number of users | simple update | join (naïve oracle) | join (refined oracle) |
|---|---|---|---|
| 6 | 1162 (2s) | (Inconclusive) 39 (<1s) | 174 (<1s) |
| 7 | 2746 (4s) | (Inconclusive) 45 (<1s) | 370 (<1s) |
| 8 | 6312 (15s) | (Inconclusive) 51 (<1s) | 787 (1s) |
| 9 | 14228 (56s) | (Inconclusive) 57 (<1s) | 1238 (2s) |
| 10 | 31614 (4:19m) | (Inconclusive) 63 (<1s) | 976 (2s) |
| 11 | 69478 (21:35m) | (Inconclusive) 69 (<1s) | 1036 (2s) |
| 12 | - | (Inconclusive) 75 (<1s) | 1707 (3s) |
| 13 | - | (Inconclusive) 81 (<1s) | 3112 (8s) |
| 14 | - | (Inconclusive) 87 (<1s) | 6083 (21s) |
| 15 | - | (Inconclusive) 93 (<1s) | 12380 (1:06m) |
| 16 | - | (Inconclusive) 99 (<1s) | 25900 (4:18m) |
| 17 | - | (Inconclusive) 105 (<1s) | 66490 (25:01m) |

Figure 5: Experimental data for the property "database cannot become full".

not forced to read any message from *pool* and we ask about safety properties, lossiness is covered by the definition too.

It is obvious that $2^{pool}$, i.e. the set of all subsets of *pool*, together with the subset ordering is a complete lattice. As long as the set of messages is finite and all parties in the protocol behave in the way that their steps are conditioned only on the presence of a message in the pool and not its absence, the system will satisfy the monotonicity property and we can apply our framework to verify safety properties.

As a very simple example let us discuss the asynchronous leader election protocol [46]. Here we have $N$ agents with their unique identifications $0, 1, \ldots, N-1$ and they select a leader with the highest id. Each agent is running concurrently and in a completely asynchronous way the piece of code in Algorithm 7.

**Algorithm 7**: **Agent**$(id)$

1: Send message $id$ to agent $(id + 1) \mod N$.
2: Receive a message $id'$ from agent $(id - 1) \mod N$.
3: **if** $id < id'$ send message $id'$ to agent $(id + 1) \mod N$.
4: **if** $id = id'$ output "I am the leader".
5: **if** $id > id'$ resend the message $id$ to agent $(id + 1) \mod N$.
6: **goto** line 2

The safety property we are interested in is that during any run of the protocol at most one agent will ever output "I am the leader".

The messages stored in the pool can be modelled as pairs $(target, id) \in \{0, \ldots, N-1\} \times \{0, \ldots, N-1\}$ where *target* is the id of the agent to whom the message is sent and $id$ is the actual content of the message so that the problem fits into our framework. We established the correctness of the protocol using the

| Number of agents | simple update | cover update | join update |
|---|---|---|---|
| 2 | 8 (<1s) | 6 (<1s) | 5 (<1s) |
| 4 | 144 (<1s) | 22 (<1s) | 12 (<1s) |
| 5 | 840 (5s) | 37 (<1s) | 17 (<1s) |
| 6 | 5760 (5:20m) | 58 (<1s) | 23 (<1s) |
| 7 | 45360 (671:02m) | 86 (1s) | 30 (<1s) |
| 10 | - | 222 (13s) | 57 (<1s) |
| 15 | - | 682 (4:21m) | 122 (2s) |
| 25 | - | 2927 (283:16m) | 327 (12s) |
| 50 | - | - | 1277 (4:19m) |
| 100 | - | - | 5052 (98:45m) |

Figure 6: Experimental data for the leader election protocol

classical search through the whole state space using the simple update, using our reachability algorithm with cover update and also using the join update. In all three cases the results were conclusive and enabled us to claim that the protocol is correct. The experimental data in Figure 6 confirm the large savings in the number of explored states (more precisely in the number of states that ever appear in the *waiting* list) necessary to establish the correctness result.

It is clear that already the cover optimization, which is always complete and sound, reduces the state space considerably. The fact that the join optimization gave conclusive results that were enough to establish the correctness of the protocol is a consequence of the simple nature of the studied protocol. In more complicated ones the LaGAR approach will be necessary in most situations. In the future work we plan to experiment with larger case studies from the Web Service protocol stack, in particular with WS-Atomic Transaction [72] and WS-Business activity [73] protocols.

## 5.3 Cache Analysis

To ensure safe scheduling of real-time systems the Worst-Case Execution Time (WCET) of each task in the system is required [103]. One major part of determining WCETs for modern processors is accounting for the effects of the memory cache. Efficient abstractions exist for analysing some types of caches [8], which we have implemented as a lattice. By recasting the cache analysis into our framework we gain the ability to give WCET guarantees, and gradually refining those guarantees by being more and more concrete with regards to the data-flow of the program.

On a simple program (binary search in array of size 100) and a simple cache we get the same WCET using all approaches. The complete state space is 5726 states (computed in 6s), cover update reduces this to 4043 states (3s), while join only needs to store 3944 states ($3s$). On more complex examples join will start to give overapproximated guarantees, which can be refined using LaGAR.

### 5.4 Timed Automata

The theory of timed automata [10] is a good example of the wide applicability of our framework. Timed automata are finite state machines equipped with a finite number of real-valued clocks. Transitions in timed automata can be conditioned on a particular range of clock values and as an effect some of the clocks can be reset. The transition system generated by timed automata has infinitely (even uncountably) many states as configurations in timed automata consist of the actual states together with the clock values.

The theory of *regions* [10] and *zones* (see e.g. [51, 21]) was developed in order to provide a suitable finite-state abstraction of the clock values. For example a zone is a collection of clock valuations that satisfy a set of integer upper- and lower-bound constraints on the clock values and differences of clock values. As one can without loss of generality restrict the integer values that appear in the constraints by the largest constant that is present in a given timed automaton, we end up with only finitely many possible zones. We can now consider an abstracted transition system where configurations are pairs $(s, Z)$ such that $s$ is a location of the automaton and $Z$ is a zone represented in a finite way by a set of constraints. Moreover, the set of all zones together with the classical set inclusion forms a (finite) lattice.

Our reachability algorithm with simple update then restates the basic reachability algorithm for timed automata, the cover update corresponds to zone inclusion check which is also well known in the theory of timed automata, and the join operation provides a convex-hull over-approximation. We refer to e.g. [21] for more information. All these algorithms are implemented in verification tools like Uppaal [19] and numerous case studies confirm their efficiency in practice.

## 6 Conclusion

We have presented a novel formal model of lattice transition systems. The model was motivated by numerous concrete applications and we generalized the domain-specific ideas into an abstract framework. We only require that the transition relation of our model is monotonic with respect to the lattice elements, in order words, the higher we are in the lattice the more behaviour is possible. This property allowed us to prove the correctness of our reachability algorithms with different levels of abstraction.

The cover update termination condition of the algorithm was proved sound and complete and already on its own provided a significant reduction of the searchable state space on our examples. Further reduction of the state space can be achieved by applying the join, however, at the expense of the possibility of inconclusive answers. To account for this, we suggested a LaGAR refinement algorithm that repeatedly splits selected join nodes and eventually provides a conclusive answer.

Finally, we have argued about the applicability of the framework based on several concrete examples from different domains. Our initial experimental find-

ings look very promising and in the future we plan to make the proposed algorithm more widely available by introduction of user-defined lattice-types in the verification tool Uppaal [19].

# Paper C

# opaal: A Lattice Model Checker

Andreas Engelbredt Dalsgaard
René Rydhof Hansen
Kenneth Yrke Jørgensen
Kim Gulstrand Larsen
Mads Chr. Olesen
Petur Olsen
Jiří Srba

*Department of Computer Science, Aalborg University,*
*Selma Lagerlöfs Vej 300, DK-9220 Aalborg East, Denmark.*
{andrease,rrh,kyrke,kgl,mchro,petur,srba}@cs.aau.dk

**Abstract** We present a new open source model checker, `opaal`, for automatic verification of models using lattice automata. Lattice automata allow the users to incorporate abstractions of a model into the model itself. This provides an efficient verification procedure, while giving the user fine-grained control of the level of abstraction by using a method similar to Counter-Example Guided Abstraction Refinement. The `opaal` engine supports a subset of the Uppaal timed automata language extended with lattice features. We report on the status of the first public release of `opaal`, and demonstrate how `opaal` can be used for efficient verification on examples from domains such as database programs, lossy communication protocols and cache analysis.

## 1 Introduction

Common to almost all applications of model checking is the notion of an underlying concrete system with a very large—or sometimes even infinite—concrete

state space. In order to enable model checking of such systems, it is necessary to construct an abstract model of the concrete system, where some system features are only modelled approximately and system features that are irrelevant for a given verification purpose are "abstracted away".

The `opaal` model checker described in this paper allows for such abstractions to be integrated in the model through user-defined lattices. Models are formalised by *lattice automata*: synchronising extended finite state machines which may include lattices as variable types. The lattice elements are ordered by the amount of behaviour they induce on the system, that is, larger lattice elements introduce more behaviour. We call this the *monotonicity property*. The addition of explicit lattices makes it possible to apply some of the advanced concepts and expressive power of abstract interpretation directly in the models.

Lattice automata, as implemented in `opaal`, are a subclass of well-structured transition systems [44]. The tool can exploit the ordering relation to reduce the explored state space by not re-exploring a state if its behaviour is *covered* by an already explored state. In addition to the ordering relation, lattices have a *join operator* that joins two lattice elements by computing their least upper bound, thereby potentially overapproximating the behaviour, with the gain of a reduced state space. Model checking the overapproximated model can however be inconclusive. We introduce the notion of a *joining strategy* affording the user more control over the overapproximation, by specifying which lattice elements are joinable. This allows for a form of user-directed CEGAR (Counter-Example Guided Abstraction Refinement) [52, 17]. The CEGAR approach can easily be automated by the user, by exploiting application-specific knowledge to derive more fine-grained joining strategies given a spurious error trace. Thus providing, for some systems and properties, efficient model checking and conclusive answers at the same time.

The `opaal` model checker is released under an open source license, and can be freely downloaded from our webpage: `www.opaal-modelchecker.com`. The tool is available both in a GUI and CLI version, shown in Fig. 1. The Uppaal [19] GUI is used for creation of models.

The `opaal` tool is implemented in Python and is a stand-alone model checking engine. Models are specified using the Uppaal XML format, extended with some specialised lattice features. Using an interpreted language has the advantage that it is easy to develop and integrate new lattice implementations in the core model checking algorithm. Our experiments indicate that although `opaal` uses an interpreted language, it is still sufficiently fast to be useful.

Users can create new lattices by implementing simple Python class interfaces. The new classes can then be used directly in the model (including all user-defined methods). Joining strategies are defined as Python functions.

An overview of the `opaal` architecture is given in Fig. 2, showing the five main components of `opaal`. The "Successor Generator" is responsible for generating a transition function for the transition system based on the semantics of Uppaal automata. The transition function is combined with one or more lattice implementations from the "Lattice Library".

The "Successor Generator" exposes an interface that the "Reachability Checker"

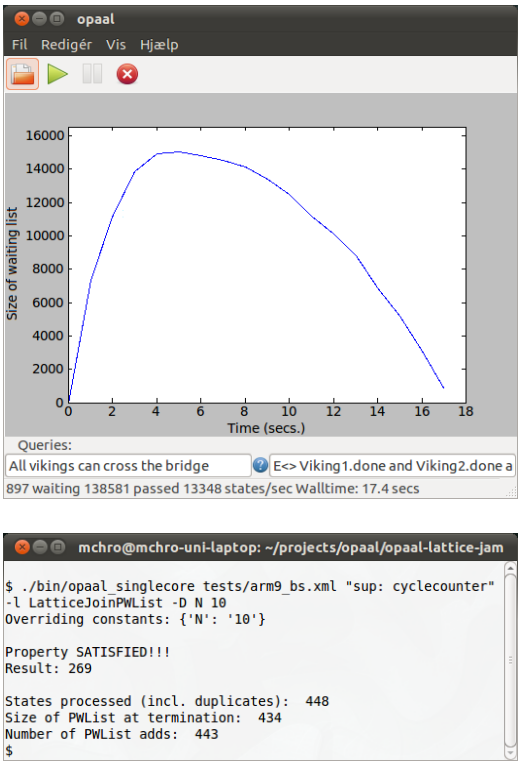


Figure 1: `opaal` GUI and CLI

can use to perform the actual verification. During this process a "Passed-Waiting List" is used to save explored and to-be explored states; it employs a user-provided "Joining Strategy" on the lattice elements of states, before they are added to the list.

## 2 Examples

In this section we present a few examples to demonstrate the wide applicability of `opaal`. The tool currently has a number of readily available lattices that are used to abstract the real data in our examples.

### 2.1 Database Programs

In recent work by Olsen et al. [78], the authors propose using present-absent sets for the verification of database programs. The key idea is that many behavioural properties may be verified by only keeping track of a few representative data values.

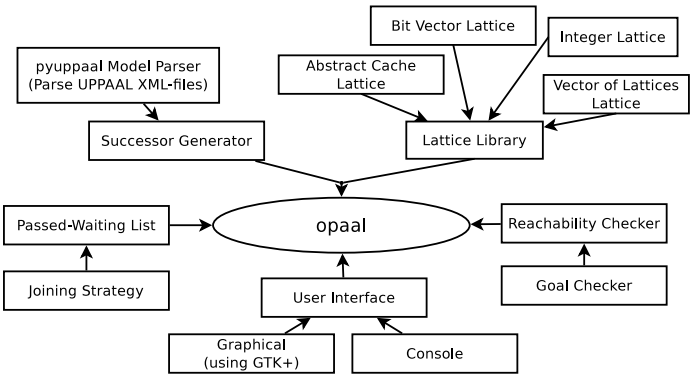


Figure 2: Overview of `opaal`'s architecture.

This idea can be naturally described as a lattice tracking the definite present- and absent-ness of database elements. In the model, this is implemented using a bit-vector lattice. For the experiment we adopt a model from [78], where users can login, work, and logout. The model has been updated to fit within the lattice framework, as shown in Fig. 3. In the code in Fig. 4, the construct **extern** is used on line 3 to import a lattice from the library. Subsequently two lattice variables, pLogin and aLogin, are defined at line 4 and 5, both vectors of size N_USERS. The lattice variables are used in the transitions of the graphical model, where e.g. a special method "num0s()" is used to count the number of 0's in the bitvector. The definition of a lattice type in Fig. 5 is just an ordinary Python class with at least two methods: join and the ordering.

We can verify that two users of the system cannot work at the same time using explicit exploration, or by exploiting the lattice ordering to do cover checks, see Fig. 6.

Another property to check is that the database cannot become full. For this property we can exploit a CEGAR approach: A naïve joining strategy will give inconclusive results, but refining the joining strategy not to join two states if the resulting state has a full database, leads to conclusive results while still preserving a significant speedup, see Fig. 7.

## 2.2 Asynchronous Lossy Communication Protocol: Leader Election

Communication protocols where messages are asynchronously passed via an unreliable (lossy and duplicating) medium can be modelled as a lattice automaton. As long as we are interested in safety properties, such a communication can be modelled as a set of already sent messages called *pool*. Initially the set *pool* is empty. Once a message it sent, it is added to the set *pool* and it remains there forever (duplication). As the protocol parties are not forced to read any message from *pool* and we ask about safety properties, lossiness is covered by

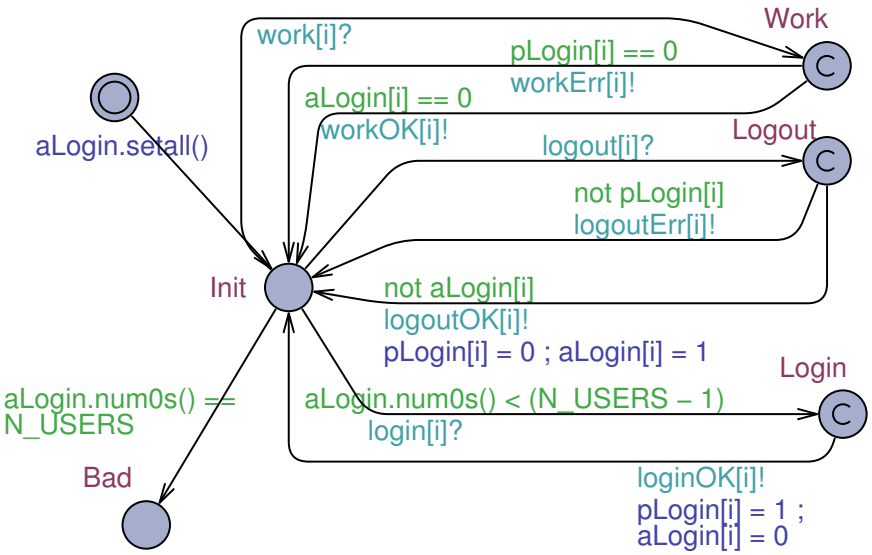


Figure 3: Database model

```
1 const int N_USERS = 17;
2 ...
3 extern IntersBitVector;
4 IntersBitVector pLogin[N_USERS];
5 IntersBitVector aLogin[N_USERS];
```

Figure 4: Lattice variables

the definition too.

It is obvious that $2^{pool}$, i.e. the set of all subsets of *pool*, together with the subset ordering is a complete lattice. As long as the set of messages is finite and all parties in the protocol behave in the way that their steps are conditioned only on the presence of a message in the pool and not on its absence, the system will satisfy the monotonicity property and we can apply our model checker.

We have modelled the asynchronous leader election protocol [46] in `opaal`. Here we have $N$ agents with their unique identifications $0, 1, \ldots, N-1$ and they select a leader with the highest id. Experimental data, for the property that

```
1 class IntersBitVector:
2     def join(self, other):
3         ...
4
5     def __le__(self, other):
6         ...
```

Figure 5: Lattice library (in Python)

| Number of users | explicit exploration | cover check |
|---|---|---|
| 2 | 224 (<1s) | 56 (<1s) |
| 3 | 2352 (2s) | 336 (<1s) |
| 4 | 21952 (28s) | 1792 (2s) |
| 5 | 192080 (8:22m) | 8960 (9s) |
| 6 | - | 43008 (48s) |
| 7 | - | 200704 (4:38m) |

Figure 6: Explored states and time for the property "no two users work at the same time"

| # of users | explicit exploration | joining (naïve) | joining (refined) |
|---|---|---|---|
| 8 | 6312 (15s) | (Incon.) 51 (<1s) | 787 (1s) |
| 9 | 14228 (56s) | (Incon.) 57 (<1s) | 1238 (2s) |
| 10 | 31614 (4:19m) | (Incon.) 63 (<1s) | 976 (2s) |
| 11 | 69478 (21:35m) | (Incon.) 69 (<1s) | 1036 (2s) |
| 12 | - | (Incon.) 75 (<1s) | 1707 (3s) |
| 16 | - | (Incon.) 99 (<1s) | 25900 (4:18m) |
| 17 | - | (Incon.) 105 (<1s) | 66490 (25:01m) |

Figure 7: Explored states and time for the property "database cannot become full"

| Number of agents | explicit exploration | cover check | joining |
|---|---|---|---|
| 5 | 840 (5s) | 37 (<1s) | 17 (<1s) |
| 6 | 5760 (5:20m) | 58 (<1s) | 23 (<1s) |
| 7 | 45360 (671:02m) | 86 (1s) | 30 (<1s) |
| 15 | - | 682 (4:21m) | 122 (2s) |
| 25 | - | 2927 (283:16m) | 327 (12s) |
| 50 | - | - | 1277 (4:19m) |
| 100 | - | - | 5052 (98:45m) |

Figure 8: Explored states and time for the leader election protocol

only the agent with the highest id can become leader, are provided in Fig. 8. The cover check column refers to using only the monotonicity property to reduce the explored state-space. We can see that while being exact (no overapproximation), the speed-up is considerable. Moreover, using the join strategy provides even more significant speed-up while still providing conclusive answers.

### 2.3 Cache Analysis

To ensure safe scheduling of real-time systems, the estimation of Worst-Case Execution Time (WCET) of each task in a given system is necessary [103]. One major part of determining WCETs for modern processors is accounting for the effects of the memory cache. Efficient abstractions exist for analysing some types of caches [8], which we have implemented as a lattice. By recasting the cache analysis into our framework we gain the ability to give WCET guarantees, and gradually refine those guarantees by being more and more concrete with respect to the data-flow of the program.

On a simple program (binary search in array of size 100) and a simple cache we get the same WCET using all approaches. The complete state space has 5726 states (computed in 6s), cover update reduces this to 4043 states (3s), while join only needs to store 3944 states (3$s$). On more complex examples join will start to give overapproximated guarantees, which can be further refined.

### 2.4 Timed Automata

It is well-known that the theory of *zones* of timed automata (see e.g. [51, 21]) is a finite-state abstraction of clock values with a lattice structure. A zone-lattice is currently being developed for use in `opaal`, but has not matured to a point where meaningful experiments can be made yet.

## 3 Conclusion

We presented a new model checker, `opaal`, for lattice automata and provided a number of applications. The expressiveness of the formalism, derived from well-structured transition systems, promises broad applicability of the tool. Our initial experiments indicate that careful abstraction using the techniques implemented in `opaal` lead to efficient verification.

We plan on extending the foundations of `opaal` to additional formalisms such as Petri nets, as well as on improving the performance of the tool by rewriting core parts in a compiled language. Of course, additional lattices and areas of application are also to be investigated.

# Paper D

# Automata Learning Through Counterexample-Guided Abstraction Refinement*

Fides Aarts[1]
Faranak Heidarian[1**]
Petur Olsen[2]
Frits Vaandrager[1]

[1]

Institute for Computing and Information Sciences
Radboud University Nijmegen
P.O. Box 9010
6500 GL Nijmegen
the Netherlands

[2]

Department of Computer Science
Aalborg University
Aalborg, Denmark

*This research was supported by European Community's Seventh Framework Programme under grant agreement no 214755 (QUASIMODO).
**Research supported by NWO/EW project 612.064.610 Abstraction Refinement for Timed Systems (ARTS).

**Abstract** State-of-the-art tools for active learning of state machines are able to learn state machines with at most in the order of 10.000 states. This is not enough for learning models of realistic software components which, due to the presence of program variables and data parameters in events, typically have much larger state spaces. Abstraction is the key when learning behavioral models of realistic systems. Hence, in most practical applications where automata learning is used to construct models of software components, researchers manually define abstractions which, depending on the history, map a large set of concrete events to a small set of abstract events that can be handled by automata learning tools. In this article, we show how such abstractions can be constructed fully automatically for a restricted class of extended finite state machines in which one can test for equality of data parameters, but no operations on data are allowed. Our approach uses counterexample-guided abstraction refinement: whenever the current abstraction is too coarse and induces nondeterministic behavior, the abstraction is refined automatically. Using a prototype implementation of our algorithm, we have succeeded to learn–fully automatically–models of several realistic software components, including the biometric passport and the SIP protocol.

# 1 Introduction

The problem to build a state machine model of a system by providing inputs to it and observing the outputs resulting, often referred to as black box system identification, is both fundamental and of clear practical interest. A major challenge is to let computers perform this tasks in a rigorous manner for systems with large numbers of states. Tools that are able to infer state machine models automatically by systematically "pushing buttons" and recording the resulting behavior will have numerous applications in different domains. For instance, they may help us to understand and analyze the behavior of legacy software and can be used to automatically derive tests to check that a protocol behaves in accordance with a reference implementation.

Within active learning it is assumed that a *learner* infers an automaton through interaction with a *teacher*. The well-known $L^*$ algorithm of Angluin [15], for instance, assumes that the teacher knows a FSM $\mathcal{M}$. The learner (initially) only knows the set of actions and her task is to learn a machine that is equivalent to $\mathcal{M}$. The teacher will answer two types of questions: *membership queries* ("is string $w$ in the language accepted by $\mathcal{M}$") and *equivalence queries* ("is an hypothesized machine $\mathcal{H}$ correct, i.e., equivalent to the machine $\mathcal{M}$?"). In case of a no-answer, the teacher will also provide a counterexample that proves that the learner's hypothesis is wrong, that is, a distinguishing word from the language. After posing a finite number of queries, the algorithm will terminate with a final hypothesis $\mathcal{H}$ that is equivalent to $\mathcal{M}$.

Figure 1 illustrates how active learning can be used to obtain models of "reactive" systems. The core of the teacher now is a *SUT (System Under Test)*, a (physical) system to which we can apply inputs and whose outputs we may

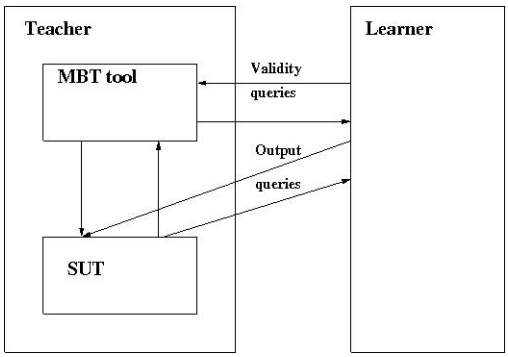


Figure 1: Active learning of reactive systems

observe. The learner interacts directly with the SUT to infer a model. Since the SUT cannot respond to equivalence queries, the teacher is also equipped with a tool for model based testing (MBT). Given a hypothesized model, this tool "approximates" equivalence queries by generating a long test sequence using some model based testing algorithm. If the SUT passes this test, that is, the output that is generated by the SUT agrees with the output predicted by the model, then we assume that the model is correct. If the output of the SUT is different from the output of the model, this constitutes a counterexample that is forwarded to the learner. It is important to note that in this setting the learner, in general, is not perfect: due to incomplete coverage, it may occur that the implementation $\mathcal{M}$ passes the test for an hypothesis $\mathcal{H}$, even though $\mathcal{M}$ and $\mathcal{H}$ are not equivalent. Using the scheme of Figure 1, Niese [76] developed an adaptation for active learning of deterministic Mealy machines. This approach has been implemented in the LearnLib tool [81] and has, for instance, been applied successfully to learn computer telephony integrated (CTI) systems [56]. LearnLib, which is the winner of the 2010 Zulu competition, is currently able to learn state machines with up to 30.000 states.

During the last few years important developments have taken place on the borderline of verification, model-based testing and automata learning, see e.g. [23, 66, 81]. There are many reasons to expect that by combining ideas from these three areas it will become possible to learn models of realistic software components with state-spaces that are many orders of magnitude larger than what state-of-the-art tools can currently handle. Last year, Aarts, Jonsson and Uijen proposed a new method for automatically learning models of large state machines [3]. Their idea is to place a so-called *mapper* $\mathcal{A}$ in between the SUT $\mathcal{M}$ and the learner, which transforms the interface of the SUT by an abstraction that maps (in a history dependent manner) the large set of actions of the SUT into a small set of abstract actions. By combining the abstract machine $\mathcal{H}$ learned in this way with information about the mapper $\mathcal{A}$, one can effectively learn a (symbolically represented) over-approximation of the behavior of SUT $\mathcal{M}$. Roughly speaking, the learner is responsible for learning the global "control

modes" in which the system can be, and the transitions between those modes, whereas the mapper records some relevant state variables (typically computed from the data parameters of previous input and output actions) and takes care of the data part of the SUT. The approach of [3] has been inspired by ideas from predicate abstraction [67]. The feasibility of the approach is illustrated by learning models of (fragments of) realistic protocols such as SIP and TCP [3], and of the new biometric passport [4]. The learned SIP model is an extended finite state machine with 29 states, 3741 transitions, and 17 state variables with various types (booleans, enumerated types, (long) integers, character strings,..). This corresponds to a state machine with an astronomical number of states and transitions, thus far fully out of reach of automata learning techniques. A major limitation of the approach of [3], however, is that the abstraction mapping has to be provided by the user, based on a priori knowledge of the SUT.

In this article, we address this problem and develop an algorithm that computes the mapper fully automatically. Nondeterminism arises naturally when we apply abstraction: it may occur that the behavior of a SUT is fully deterministic but that due to the mapper (which, for instance, abstracts from the precise value of certain input parameters), the system appears to behave nondeterministically from the perspective of the learner. In [3] LearnLib is used as the basic learning tool, and therefore the abstraction of the SUT as defined by the mapper may not exhibit any nondeterminism: if it does then one has to refine the abstraction. This is exactly what has been done repeatedly during the manual construction of the abstraction mappings in the case studies of [3]. In this article, we formalize this procedure and describe the construction of the mapper in terms of a counterexample guided abstraction refinement (CEGAR) procedure, similar to the approach developed by Clarke et al [32] in the context of model checking. Our algorithm applies to a restricted class of extended finite state machines in which one can test for equality of data parameters, but no operations on data are allowed. Using a prototype implementation of our algorithm, we have succeeded to learn – fully automatically – models of several realistic software components, including the biometric passport and the SIP protocol.

The idea to use CEGAR for learning state machines has also been explored recently by Howar at al [55], who developed and implemented a CEGAR procedure for the special case in which the abstraction is static and does not depend on the execution history. As we illustrate in this paper, our approach is applicable to a much richer class of systems, which for instance includes the SIP protocol and the various components of the Alternating Bit Protocol. We expect that our CEGAR based approach can be further extended to systems that may apply simple or known operations on data, using technology for automatic detection likely invariants, such as Daikon [42]. Even though the class of systems to which our approach currently applies is limited, the fact that we are able to learn models of systems with data fully automatically is a major step towards a practically useful technology for automatic learning of models of software components.

For reasons of space, all proofs have been omitted from this paper. (We

heavily use the characterization of the **ioco** preorder in terms of alternating simulations [5].) The models that we learned using our CEGAR algorithm and that are described in the experiments section, are available via `http://www.mbsd.cs.ru.nl/publications/papers/fvaan/`.

**Acknowledgement** We thank Gábor Angyal and Harco Kuppens for their help with linking our tool with the CADP toolset, which allowed us to check that the learned models are indeed equivalent to the original models of the SUT.

## 2 Preliminaries

We model reactive systems by a simplified *interface automata* [39].

**Definition 1** (IA)**.** *An* interface automaton (IA) *is a tuple* $\mathcal{I} = \langle I, O, Q, q^0, \rightarrow \rangle$ *where*

- $I$ *and* $O$ *are disjoint sets of input and output actions,*
- $Q$ *is a non-empty set of states,*
- $q^0 \in Q$ *is the initial state, and*
- $\rightarrow \subseteq Q \times (I \cup O) \times Q$ *is the transition relation. We write* $q \xrightarrow{a} q'$ *if* $(q, a, q') \in \rightarrow$.

*The output actions are assumed to be under the control of the system whereas input actions are under control of the environment. An action* $a$ *is* enabled *in state* $q$*, noted* $q \xrightarrow{a}$*, if* $q \xrightarrow{a} q'$ *for some state* $q'$*. We write* $out_{\mathcal{I}}(q)$ *or just* $out(q)$ *if* $\mathcal{I}$ *is clear from the context, for the set* $\{a \in O \mid q \xrightarrow{a}\}$ *of output actions enabled in state* $q$*.* $\mathcal{I}$ *is said to be:*

- input-deterministic *if for each state* $q \in Q$ *and for each input action* $i \in I$ *there is at most one outgoing transition of* $q$ *with label* $i$*:* $q \xrightarrow{i} q' \wedge q \xrightarrow{i} q'' \Rightarrow q' = q''$;
- output-deterministic *if for each state* $q \in Q$ *and for each output action* $o \in O$ *there is at most one outgoing transition of* $q$ *with label* $o$*:* $q \xrightarrow{o} q' \wedge q \xrightarrow{o} q'' \Rightarrow q' = q''$;
- deterministic *if it is both input- and output-deterministic;*
- output-determined *if each state has at most one outgoing output transition;*
- input-enabled *if each input action is enabled in each state, that is* $q \xrightarrow{i}$*, for all* $q \in Q$ *and all* $i \in I$*. An* I/O automaton*(IOA) is an input-enabled IA.*

A state $q$ is called *quiescent* if it enables no output actions. Let $\delta$ be a fresh action symbol (not in $I \cup O$). Then the $\delta$-extension of $\mathcal{I}$, notation $\mathcal{I}^\delta$, is the IA obtained by adding $\delta$ to the set of outputs, and $\delta$-loops to all the quiescent states of $\mathcal{I}$. We write $O^\delta = O \cup \delta$.

We extend the transition relation to sequences by defining, for $\sigma \in (I \cup O)^*$, $\rightarrow_*$ to be the transitive and reflexive closure of $\xrightarrow{a}$. We use $\epsilon$ to denote the empty sequence. We say that state $q$ is *reachable* if $q^0 \xrightarrow{\sigma}_* q$, for some $\sigma$. We write $q \xrightarrow{\sigma}_*$ if $q \xrightarrow{\sigma}_* q'$, for some state $q'$. We say that $\sigma \in (I \cup O)^*$ is a *trace* of $\mathcal{I}$ if $q^0 \xrightarrow{\sigma}_*$, and write $Traces(\mathcal{I})$ for the set of traces of $\mathcal{I}$. We write $\mathcal{I}$ **after** $\sigma$ for the set $\{q \in Q \mid q^0 \xrightarrow{\sigma}_* q\}$ of states of $\mathcal{I}$ that can be reached with trace $\sigma$. Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be IAs with the same signature. Then $\mathcal{I}_1$ and $\mathcal{I}_2$ are *input-output conforming*, notation $\mathcal{I}_1$ **ioco** $\mathcal{I}_2$, if

$$\forall \sigma \in Traces(\mathcal{I}_2^\delta) : out(\mathcal{I}_1^\delta \textbf{ after } \sigma) \subseteq out(\mathcal{I}_2^\delta \textbf{ after } \sigma).$$

The **ioco** relation is the main notion of conformance in the model based testing theory of Tretmans [94, 98].

An IA $\mathcal{I}$ is *behavior-deterministic* if, for each $\sigma \in Traces(\mathcal{I}^\delta)$, the set $out(\mathcal{I}^\delta$ **after** $\sigma)$ contains at most one element. Note that any deterministic output-determined IA is behavior-deterministic. Note also that if $\mathcal{I}$ is an IOA, $out(\mathcal{I}^\delta$ **after** $\sigma)$ always contains at least one element. Hence, for a behavior-deterministic IOA $\mathcal{I}$, $out(\mathcal{I}^\delta$ **after** $\sigma)$ is a singleton set, for each $\sigma$. Finally, note that when $\mathcal{I}_1$ **ioco** $\mathcal{I}_2$ and $\mathcal{I}_2$ is behavior-deterministic, then $\mathcal{I}_1$ is behavior-deterministic as well.

# 3 Inference of I/O Automata

## 3.1 Basic Framework for Inference of I/O Automata

We present a (slight variation of) the approach of [5] for active learning of I/O automata. In this approach we assume a teacher, who knows a behavior-deterministic IOA $\mathcal{T} = \langle I, O, Q, q^0, \rightarrow \rangle$, and a learner, who initially knows a deterministic IA $\mathcal{P} = \langle I, O, P, p^0, \rightarrow' \rangle$, called the *learning purpose*. We require $\mathcal{T}$ **ioco** $\mathcal{P}$. The task of the learner is to learn the part of $\mathcal{T}$ whose behavior is compatible with $\mathcal{P}$. The teacher records the current state of $\mathcal{T}$, which initially is $q^0$, and the learner records the current state of $\mathcal{P}$, which initially is $p^0$. Suppose the teacher is in state $q$ and the learner is in state $p$. The learner may engage in four types of interactions with the teacher:

1. If an input transition $p \xrightarrow{i} {}' p'$ is enabled in $\mathcal{P}$ then the learner may present input $i$ to the teacher. The learner then jumps to $p'$ and the teacher jumps to some state $q'$ such that $q \xrightarrow{i} q'$.

2. The learner may send an *output query* to the teacher. There are two possibilities. If state $q$ is quiescent, then the teacher remains in state $q$ and returns answer $\delta$ to the learner. Otherwise, an output transition

$q \xrightarrow{o} q'$ is selected by the teacher, the teacher jumps to $q'$, and returns answer $o$ to the learner. The learner then jumps to a state $p'$ that can be reached by the response $o$ or $\delta$ (by the assumptions we know such a state exists).

3. The learner may order a *reset*. Both the learner and the teacher then return to their initial state, $p^0$ and $q^0$, respectively.

4. The learner may present an *hypothesis* to the teacher, which is an IA $\mathcal{H}$ such that $\mathcal{H} \leq_{AI}^{\delta} \mathcal{P}$. We say that $\mathcal{H}$ is correct if $\mathcal{T}$ **ioco** $\mathcal{H}$. In this case the teacher returns the answer **yes**. If a hypothesis is not correct then $\mathcal{H}^\delta$ has a trace $\sigma$ such that the unique output $o$ enabled by $\mathcal{T}^\delta$ after $\sigma$ is not amongst the outputs enabled by $\mathcal{H}^\delta$ after $\sigma$. The teacher then returns the answer **no** together with counterexample $\sigma o$.

**Lemma 1.** *Suppose hypothesis $\mathcal{H}$ is behavior deterministic and output determined. Then $\mathcal{H}$ is correct iff $\mathcal{H}$ is behavior equivalent (bisimilar) to the synchronous product of $\mathcal{T}$ and $\mathcal{P}$.*

An algorithm for learning I/O automata is an effective procedure which, for all finite $\mathcal{T}$ and $\mathcal{P}$, allows the learner to come up with a correct, behavior-deterministic and output determined hypothesis $\mathcal{H}$ after a finite number of interactions with the teacher. In [5], it is shown that any algorithm for learning Mealy machines can be transformed into an algorithm for learning I/O automata. Efficient algorithms for learning Mealy machines (and hence I/O automata) have been implemented in the tool Learnlib [81].

## 3.2 Inference Using Abstraction

In order to learn an over-approximation of a "large" IOA $\mathcal{T} = \langle I, O, Q, q^0, \rightarrow \rangle$, we place a mapper between the teacher and the learner, which translates the concrete actions in $I$ and $O$ to abstract actions in $X$ and $Y$, and vice versa. The task of the learner is to infer an IA with alphabet $X$ and $Y$. The behavior of the mapper is fully determined by an *abstraction* $\mathcal{A}$.

**Definition 2** (Abstraction)**.** *An abstraction for a set of inputs $I$ and a set of outputs $O$ is a tuple $\mathcal{A} = \langle \mathcal{I}, X, Y, \Upsilon \rangle$, in which*

- *$\mathcal{I} = \langle I \cup O, \emptyset, R, r^0, \rightarrow \rangle$ is a deterministic IOA,*
- *$X$ and $Y$ are finite sets of abstract input and output actions, and*
- *$\Upsilon \subseteq R \times (I \cup O) \times (X \cup Y)$ is a relation that relates, for each local state, concrete actions to abstract ones. We write $a \Upsilon_r z$ instead of $(r, a, z) \in \Upsilon$ and require*

  1. *$a \Upsilon_r z$ implies $a \in I \iff z \in X$*
     *(inputs are related to inputs, and outputs to outputs)*

2. $\forall r \in R\ \forall z \in X \cup Y\ \exists a \in I \cup O : a \Upsilon_r z$
   *(abstract actions have related concrete actions)*
3. $\forall r \in R\ \forall a \in I \cup O\ \exists z \in X \cup Y : a \Upsilon_r z$
   *(concrete actions have related abstract actions)*
4. $\forall r \in R\ \forall o \in O\ \forall y, y' \in Y : o \Upsilon_r y \wedge o \Upsilon_r y' \Rightarrow y = y'$
   *(each concrete output has at most one related abstract output)*

The behavior of the mapper for $\mathcal{A}$ can be defined as follows:

- Initially, it is in state $r^0$.
- If the mapper is in state $r$ and receives an abstract input action $x$ from the learner, it non-deterministically picks a concrete action $i$ such that $i \Upsilon_r x$, forwards $i$ to the teacher, and updates its state to $r'$, where $r'$ is the unique state such that $r \xrightarrow{i} r'$.
- If the mapper receives an output query from the learner, it forwards this query to the teacher, without changing its current state,
- If the mapper is in state $r$ and receives an output $o \in O$ from the teacher, it forwards the unique abstract output $y$ such that $o \Upsilon_r y$ to the learner and updates its state to $r'$, where $r'$ is the unique state such that $r \xrightarrow{o} r'$. If the mapper receives a $\delta$ action from the teacher, this is forwarded to the learner and the state remains unchanged.
- If the mapper receives a *reset* from the learner, it changes its current state to $r^0$ and forwards the *reset* to the teacher.
- If the mapper receives an hypothesis $\mathcal{H} = \langle X, Y, S, s^0, \rightarrow \rangle$ from the learner, it constructs interface automaton $\mathcal{A} \parallel \mathcal{H}$ and forwards this as an hypothesis to the teacher. Here, $\mathcal{A} \parallel \mathcal{H}$ is the interface automaton $\langle I, O, R \times S, (r^0, s^0), \rightarrow_{\mathsf{conc}} \rangle$, where $\rightarrow_{\mathsf{conc}}$ is given by the rule:

$$\frac{r \xrightarrow{a} r' \quad s \xrightarrow{z} s' \quad a \Upsilon_r z}{(r, s) \xrightarrow{a}_{\mathsf{conc}} (r', s')}$$

  Intuitively, $\mathcal{A} \parallel \mathcal{H}$ is a concrete version of the abstract hypothesis $\mathcal{H}$.
- If the mapper receives **yes** from the teacher in response to a validity query, it forwards this response to the learner and the learning is done. If the mapper receives the answer **no** with a concrete counterexample, then the mapper translates this counterexample to an abstract counterexample using relation $\Phi$. Here relation $\Phi$, which relates sequences over $I \cup O_\delta$ to sequences over $X \cup Y_\delta$, is defined inductively by:
  - $\epsilon \Phi \epsilon$
  - if $\sigma \Phi \rho$, $r^0 \xrightarrow{\sigma}_* r$ and $a \Upsilon_r z$ then $(\sigma a) \Phi (\rho z)$

Define $\mathcal{T} \parallel \mathcal{A}$, the parallel composition of $\mathcal{T}$ and $\mathcal{A}$, to be the IOA $\langle X, Y, Q \times R, (q^0, r^0), \rightarrow_{\mathsf{abst}} \rangle$, where transition relation $\rightarrow_{\mathsf{abst}}$ is given by the rule:

$$\frac{q \xrightarrow{a} q' \quad r \xrightarrow{a} r' \quad a \Upsilon_r z}{(q, r) \xrightarrow{z}_{\mathsf{abst}} (q', r')}$$

**Lemma 2.** *Suppose $\mathcal{H}$ and $\mathcal{A} \parallel \mathcal{H}$ are behavior deterministic and output determined. Then $\mathcal{T} \parallel \mathcal{A}$ **ioco** $\mathcal{H}$ if and only if $\mathcal{T}$ **ioco** $\mathcal{A} \parallel \mathcal{H}$.*

**Theorem 1.** *Assume that we have a learner that only generates hypothesis $\mathcal{H}$ such that $\mathcal{H}$ and $\mathcal{A} \parallel \mathcal{H}$ are behavior deterministic and output determined. Then a teacher for $\mathcal{T}$ and a mapper for $\mathcal{A}$ together behave like a teacher for $\mathcal{T} \parallel \mathcal{A}$.*

Lemma 2 and Theorem 1 form the basis for our abstraction approach: if we succeed to propose a correct hypothesis for the "small" model $\mathcal{T} \parallel \mathcal{A}$, then we can convert this into a correct hypothesis for the "large" model $\mathcal{T}$. The main problem in practice is that $\mathcal{T} \parallel \mathcal{A}$ may not be behavior deterministic. In this case $\mathcal{T} \parallel \mathcal{A}$ **ioco** $\mathcal{H}$ will not hold, and hence Lemma 2 can not be applied. The challenge therefore is to find abstractions $\mathcal{A}$ such that $\mathcal{T} \parallel \mathcal{A}$ is behavior deterministic.

# 4 Symbolic Abstraction

Even though our general approach for using abstraction in automata learning is phrased most naturally at the semantic level, our CEGAR algorithm and the specific restrictions on SUTs that it requires can only be phrased using a syntactic (symbolic) formulation of interface automata. Therefore, in this section, we present a general syntax for symbolic interface automata and abstractions.

We assume a first-order language with (typed) variables, function, predicate and constant symbols. We assume that each variable $v$ comes equipped with a type $\mathsf{type}(v)$, which is the set of values that it may take. Also, each term $t$ has an associated type $\mathsf{type}(t)$. We use $\equiv$ to denote syntactic equality of terms. If $V$ is a set of variables, then a *valuation* for $V$ is a function that maps each variable in $V$ to an element of its domain. We write $\mathsf{Val}(V)$ for the set of all valuations for $V$. If $\xi$ is a valuation for $V$ and $\varphi$ is a formula with free variables in $V$, then we write $\xi \models \varphi$ to denote that $\xi$ satisfies $\varphi$. Similarly, if $t$ is a term then we write $[\![t]\!]_\xi$ for the value to which $t$ evaluates under valuation $\xi$. If $V' \subseteq V$ then $\xi \lceil V'$ denotes the restriction of $\xi$ to the variables in $V'$. If $v_1, \ldots, v_n$ are variables in $V$ and $t_1, \ldots, t_n$ are terms, then we write $\xi[v_1, \cdots, v_n := t_1, \cdots, t_n]$ for the valuation in which all the variables have the same values as in $\xi$ except for $v_1, \ldots, v_n$ which are evaluated to $[\![t_1]\!]_\xi, \ldots, [\![t_n]\!]_\xi$, respectively.

We employ a slight variation of Jonsson's [58] approach for specification of distributed systems and define a *symbolic interface automaton* by means of a program-like notation with guarded multiple assignments. Each assignment statement is labeled with an event which can denote either reception or transmission of a message.

**Definition 3.** *An* event signature $\Sigma$ *is a triple* $\langle E_I, E_O, T\rangle$*, where*

- $E_I$ *is a finite set of* input event primitives*,*
- $E_O$ *is a finite set of* output event primitives*, with* $E_I \cap E_O = \emptyset$*,*
- $T$ *is a set that contains exactly one event term for each* $\varepsilon \in E_I \cup E_O$*. Here an* event term *for* $\varepsilon$ *is an expression* $\varepsilon(p_1, \ldots, p_m)$ *in which* $p_1, \ldots p_n$ *are pairwise different variables. We require that the sets of variables occurring in different event terms of* $T$ *are disjoint.*

*Let* $E \subseteq E_I \cup E_O$*. We write* $T\lceil E$ *to denote the subset of event terms in* $T$ *build using event primitives in* $E$*.*

Intuitively, an event signature specifies the possible interactions between an interface automaton and its environment.

**Definition 4** (SIA)**.** *A* symbolic interface automaton $\mathcal{S}$ *is a tuple* $\langle \Sigma, V, \Theta, \mathfrak{A}\rangle$*, where*

- $\Sigma = \langle E_I, E_O, T\rangle$ *is an event signature,*
- $V$ *is a finite set of variables, referred to as* state variables*,*
- $\Theta$ *is an assertion, referred to as the* initial condition*, whose free variables are in* $V$*,*
- $\mathfrak{A}$ *is a finite set of* input transitions *and* output transitions*. Input transitions in are of the form*

$$\textbf{\textit{event}}\ \varepsilon_I(p_1, \cdots, p_m)\ \textbf{\textit{when}}\ g\ \textbf{\textit{do}}\ \langle v_1, \cdots, v_n\rangle := \langle t_1, \cdots, t_n\rangle$$

*where* $\varepsilon_I \in E_I$*,* $\varepsilon_I(p_1, \ldots, p_m) \in T$*,* $v_1, \cdots, v_n$ *are distinct variables in* $V$*,* $g$ *is an assertion and* $t_1, \cdots, t_n$ *are terms with free variables in* $V \cup \{p_1, \cdots, p_m\}$*.* Output transitions *in* $\mathfrak{A}$ *are of the form*

$$\textbf{\textit{event}}\ \varepsilon_O(u_1, \cdots, u_m)\ \textbf{\textit{when}}\ g\ \textbf{\textit{do}}\ \langle v_1, \cdots, v_n\rangle := \langle t_1, \cdots, t_n\rangle$$

*where* $\varepsilon_O \in E_O$*,* $\varepsilon_O(p_1, \ldots, p_m) \in T$*,* $v_1, \cdots, v_n$ *are distinct variables in* $V$*,* $g$ *is an assertion, and* $u_1, \cdots, u_m$ *and* $t_1, \cdots, t_n$ *are terms with free variables in* $V$*. We require that all the terms occurring in actions from* $\mathfrak{A}$ *are appropriately typed. For instance, we require for each output transition that, for each valuation* $q \in \mathsf{Val}(V)$ *with* $q \models g$ *an for each index* $j$*,* $[\![u_j]\!]_q \in \mathsf{type}(p_j)$*.*

*An SIA is* input enabled *if, for each input event primitive, the disjunction of the set of guards of transitions for that event primitive is equivalent to true. A* symbolic I/O-automaton $\mathcal{S}$ *is an input-enabled SIA.*

A transition $\tau$ states that if under a suitable instantiation of the variables the condition $g$ is true, then the assignment statement $\langle v_1, \cdots, v_n\rangle := \langle t_1, \cdots, t_n\rangle$ can be performed together with an interaction specified by $\varepsilon_I(p_1, \cdots, p_m)$ or $\varepsilon_O(u_1, \cdots, u_m)$. It is furthermore the intention that input transitions are controlled by the environment whereas output transitions are controlled by the automaton in which they occur. In an input transition, values of $d_1, \cdots, d_m$ are received from the environment. We require $d_i \in \mathsf{type}(p_i)$, for $p_1, \cdots, p_m$ the formal parameters of the event term. In an output transition, however, we can think of $u_1, \cdots, u_m$ in $\varepsilon_O(u_1, \cdots, u_m)$ as being generated by the specification and allow them to be arbitrary terms (of the right type).

**Definition 5** (SA)**.** *Let $\Sigma_c = \langle E_I, E_O, T_c\rangle$ be an event signature. A* symbolic abstraction (SA) *for $\Sigma_c$ is a triple $\mathcal{M} = \langle \mathcal{S}, \Sigma_a, \Psi\rangle$, where*

- *$\mathcal{S} = \langle\langle E_I \cup E_O, \emptyset, T_c\rangle, V, \Theta, \mathfrak{A}\rangle$ is a deterministic SIOA,*
- *$\Sigma_a = \langle E_X, E_Y, T_a\rangle$ is an event signature, referred to as the* abstract event signature*, such that the variables that occur in $T_c$ and $T_a$ are disjoint, and*
- *$\Psi$ is a set of triples of the form $\langle \varepsilon(p_1, \cdots, p_m), \langle \varepsilon'(q_1, \cdots, q_k), \varphi\rangle$, where $\varepsilon(p_1, \cdots, p_m) \in T_c$, $\varepsilon'(q_1, \cdots, q_k) \in T_a$, $\varepsilon \in E_I \Leftrightarrow \varepsilon' \in E_X$, and $\varphi$ is a formula over $\{p_1, \cdots, p_m\} \cup \{q_1, \cdots, q_k\} \cup V$. We require that $\varphi$ satisfies the four conditions stated in Definition 2.*

The behavior of the mapper in the symbolic case is essentially the same as the behavior of the non-symbolic mapper, except that if the symbolic mapper receives a validity query $\mathcal{H} = \langle X, Y, H, h^0, \rightarrow\rangle$ from the learner, it constructs a symbolic interface automaton $\mathcal{M} \parallel \mathcal{H}$ and forwards it as a validity query to the teacher. Here, $\mathcal{M} \parallel \mathcal{H}$ is the SIA $\langle \Sigma, \bar{V}, \bar{\Theta}, \bar{\mathfrak{A}}\rangle$, where $\bar{V} = V \cup \{\ell oc\}$, $\bar{\Theta} = \Theta \wedge (\ell oc = h^0)$ and For each transition

$$\textbf{event } \varepsilon_I(p_1, \cdots, p_m) \textbf{ when } g \textbf{ do } \langle v_1, \cdots, v_n\rangle := \langle t_1, \cdots, t_n\rangle$$

in $\mathfrak{A}$ with $\varepsilon_I \in E_I$ there are corresponding transitions in $\bar{\mathfrak{A}}$ generated by the rule:

$$\frac{\begin{array}{c}\langle \varepsilon_I(p_1, \cdots, p_m), \varepsilon_X(q_1, \cdots, q_k), \varphi(p_1, \cdots, p_m, q_1, \cdots, q_k, v_1, \cdots, v_n)\rangle \in \Psi \\ h \xrightarrow{\varepsilon_X(d_1, \cdots, d_k)} h' \\ \textbf{event } \varepsilon_I(p_1, \cdots, p_m) \textbf{ when } g \textbf{ do } \langle v_1, \cdots, v_n\rangle := \langle t_1, \cdots, t_n\rangle\end{array}}{\begin{array}{l}\textbf{event } \varepsilon_I(p_1, \cdots, p_m) \\ \textbf{when } g \wedge \ell oc = h \wedge \varphi(p_1, \cdots, p_m, \hat{d}_1, \cdots, \hat{d}_k, v_1, \cdots, v_n) \\ \textbf{do } \langle v_1, \cdots, v_n, \ell oc\rangle := \langle t_1, \cdots, t_n, h'\rangle\end{array}}$$

We postulate that, for each abstract parameter $q_j$ and for each value $d \in \mathsf{type}(q_j)$, there exists a constant $\hat{d}$ in our first order language defining it. Similarly, for each transition

$$\textbf{event } \varepsilon_O(u_1, \cdots, u_m) \textbf{ when } g \textbf{ do } \langle v_1, \cdots, v_n\rangle := \langle t_1, \cdots, t_n\rangle$$

in $\mathfrak{A}$ with $\varepsilon_O \in E_O$ there are corresponding actions in $\bar{\mathfrak{A}}$ generated by the rule:

$$\frac{\langle \varepsilon_O(p_1,\cdots,p_m), \varepsilon_Y(q_1,\cdots,q_k), \varphi(p_1,\cdots,p_m,q_1,\cdots,q_k,v_1,\cdots,v_n)\rangle \in \Psi \qquad h \xrightarrow{\varepsilon_Y(d_1,\cdots,d_k)} h' \qquad \textbf{event } \varepsilon_O(u_1,\cdots,u_m) \textbf{ when } g \textbf{ do } \langle v_1,\cdots,v_n\rangle := \langle t_1,\cdots,t_n\rangle}{\begin{array}{l}\textbf{event } \varepsilon_O(u_1,\cdots,u_m)\\ \textbf{when } g \wedge \ell oc = h \wedge \varphi(u_1,\cdots,u_m,\hat{d}_1,\cdots,\hat{d}_k,v_1,\cdots,v_n)\\ \textbf{do } \langle v_1,\cdots,v_n,\ell oc\rangle := \langle t_1,\cdots,t_n,h'\rangle\end{array}}$$

# 5 Counterexample-Guided Abstraction Refinement

## 5.1 Basic Assumptions on SUTs and Abstractions

Our tool allows us to use CEGAR to learn models from a very specific class of SIAs, which we call *scalarset SIAs*. The scalarset datatype was introduced by Ip and Dill [57] as part of their work on symmetry reduction in verification. Operations on scalarsets are restricted so that states are guaranteed to have the same future behaviors, up to permutation of the elements of the scalarsets. Using the symmetries implied by the scalarsets, Ip and Dill showed that a verifier can automatically generate a reduced state space. In order to simplify learning, we only allow scalarset datatypes. On scalarsets no operations are allowed, we only assume the presence of a finite set $C$ of constants. In addition, no predicate symbols may be used in SIAs except for the equality predicate symbol. The formal definition of a scalarset SIA is presented below.

**Definition 6** (SSIA)**.** *A* scalarset *SIA is an SIA* $\mathcal{S} = \langle \Sigma, V, \Theta, \mathfrak{A}\rangle$ *such that*

- *If* $\Sigma = \langle E_I, E_O, T\rangle$ *then the type of all the variables that occur in* $T$ *is equal to* $\mathbb{N}$*.*
- *All variables in* $V$ *have type* $\mathbb{N} \cup \{\perp\}$*.*
- *Input transitions in* $\mathfrak{A}$ *are of the form*

  $$\begin{array}{l}\textbf{\textit{event}}\ \varepsilon_I(p_1,\cdots,p_n)\\ \textbf{\textit{when}}\ g\\ \textbf{\textit{do}}\ (\langle v_1,\cdots,v_q\rangle := \langle t_1,\cdots,t_q\rangle)\end{array}$$

  *where* $t_j \in \{p_1,\cdots,p_n\} \cup C \cup V$ *and* $g$ *is a boolean combination of equalities of the form* $t = t'$ *such that* $t, t' \in \{p_1,\cdots,p_n\} \cup C \cup V$*.*

- *output transitions in* $\mathfrak{A}$ *are of the form*

  $$\begin{array}{l}\textbf{\textit{event}}\ \varepsilon_O(u_1,\cdots,u_m)\\ \textbf{\textit{when}}\ g\\ \textbf{\textit{do}}\ (\langle v_1,\cdots,v_q\rangle := \langle t_1,\cdots,t_q\rangle)\end{array}$$

*where $t_i, u_i \in C \cup V$ and $g$ is a boolean combination of equalities of the form $t = t'$ such that $t, t' \in C \cup V$.*

*We enforce that a scalarset SIA may only record the first and the last occurrence of an input parameter. We say that variable $v$* records the last occurrence *of input parameter $p$ if (0) $\Theta \Rightarrow v = \perp$ holds, (1) each input transition in which $p$ occurs contains an assignment $v := p$, (2) $v$ does not appear anywhere else in either the left-hand or right-hand side of assignments. We say that variable $v$* records the first occurrence *of input parameter $p$ if (0) $\Theta \Rightarrow v = \perp$ holds, (1) for each transition with guard $g$ in which $p$ occurs either $g \Rightarrow v = \perp$ holds and the transition contains an assignment $v := p$, or $g \Rightarrow v \neq \perp$ holds and $v$ is not mentioned in the assignments, (2) $v$ does not appear anywhere else in either the left-hand or right-hand side of assignments. We require that whenever a transition contains an assignment of the form $v := p$, $v$ either records the first or the last occurrence of $p$.*

For each event signature, we define a family of symbolic abstractions, parametrized by what we call an *abstraction table*. Our CEGAR procedure starts with the simplest of these abstractions (essentially the empty table). If this abstraction is sound (in the sense that $\mathcal{T} \parallel \mathcal{A}$ is behavior deterministic) learning will succeed. Otherwise, we refine the abstraction and add an entry to our table. Since there are only finitely many possible abstractions and we know that the abstraction that corresponds to the "full" table is sound, our CEGAR approach will always terminate successfully.

**Definition 7** (Abstraction table)**.** *Let $\Sigma$ be an event signature, let $P$ be the set of variables that occur in the terms in $\Sigma$ for input event primitives, and let $U$ be the set of variables that occur in the terms in $\Sigma$ for output event primitives. We associate to each input parameter $p \in P$ two unique variables $v_p^f$ and $v_p^l$, and define $V^f = \{v_p^f \mid p \in P\}$ and $V^l = \{v_p^l \mid p \in P\}$. An* abstraction table *for $\Sigma$ is a function*

$$F : (P \cup U) \times \{0, \ldots, N-1\} \to C \cup V^f \cup V^l \cup \{\perp\},$$

*where $N = 2 \cdot |P| + |C|$. We require that if an entry in the table is undefined, all subsequent entries in the same row are undefined as well: $F(p, j) = \perp \wedge j < k \Rightarrow F(p, k) = \perp$. Furthermore, each variable or constant occurs at most once in a row: $F(p, j) = F(p, k) \neq \perp \Rightarrow j = k$. Finally, the rows corresponding to the output parameters contain all variables and constants: $p \in U \Rightarrow F(p, j) \neq \perp$. We write: $f(p) = |\{j | F(p, j) \neq \perp\}|$ for the number of variables and constants defined for parameter $p$.*

**Definition 8** (Abstraction induced by table)**.** *Let $\Sigma_c = \langle E_I, E_O, T_c \rangle$ be a (concrete) event signature and let $F$ be an abstraction table for $\Sigma_c$, as above. We define $\mathcal{M}_F$ to be the symbolic abstraction $\langle \mathcal{S}, \Sigma_a, \Psi \rangle$, where*

- *$\mathcal{S}$ is the input and output enabled scalarset SIA $\langle \langle E_I \cup E_O, \emptyset, T_c \rangle, V, \Theta, \mathfrak{A} \rangle$, in which*

- $V = V^f \cup V^l$ *is the set of state variables,*
- $\Theta$ *given by* $\Theta(v) = \perp$*, for all* $v \in V$*,*
- *for each event term* $\varepsilon_I(p_1, \ldots, p_m) \in T\lceil E_I$*,* $\mathfrak{A}$ *contains a transition of the form*

$$\textbf{\textit{event}}\ \varepsilon_I(p_1, \cdots, p_n)\ \textbf{\textit{when}}\ (\langle v^f_{p_1}, \cdots, v^f_{p_n}\rangle = \perp)$$

$$\textbf{\textit{do}}\ (\langle v^f_{p_1}, \cdots, v^f_{p_n}\rangle := \langle v^l_{p_1}, \cdots, v^l_{p_n}\rangle := \langle p_1, \cdots, p_n\rangle)$$

*and a transition of the form*

$$\textbf{\textit{event}}\ \varepsilon_I(p_1, \cdots, p_n)\ \textbf{\textit{when}}\ (\langle v^f_{p_1}, \cdots, v^f_{p_n}\rangle \neq \perp)$$

$$\textbf{\textit{do}}\ (\langle v^l_{p_1}, \cdots, v^l_{p_n}\rangle := \langle p_1, \cdots, p_n\rangle)$$

- *for each event term* $\varepsilon_O(p_1, \ldots, p_m) \in T\lceil E_O$*,* $\mathfrak{A}$ *contains a transition of the form*

$$\textbf{\textit{event}}\ \varepsilon_O(p_1, \cdots, p_m)\ \textbf{\textit{when}}\ \textit{true}\ \textbf{\textit{do}}\ \textit{nop}$$

- $\Sigma_a = \langle E_X, E_Y, T_a\rangle$ *is an abstract event signature such that, for each concrete event primitive in* $E_I$ *and* $E_O$ *we have an abstract event primitive with the same arity in* $E_X$ *and* $E_Y$*, respectively. If* $\varepsilon(p_1, \cdots, p_m)$ *is a concrete event term and* $\varepsilon'(q_1, \cdots, q_m)$ *is the corresponding abstract event term then the type of* $q_i$ *equals* $\{-1, 0, \cdots, f(p_i) - 1\}$*.*

- *For each pair of a concrete event term* $\varepsilon(p_1, \cdots, p_m)$ *and matching abstract event term* $\varepsilon'(q_1, \cdots, q_m)$*,* $\Psi$ *contains a triple* $\langle \varepsilon(p_1, \cdots, p_m), \varepsilon'(q_1, \cdots, q_m), \varphi\rangle$*. For inputs,* $\varphi$ *expresses that, for each* $i$*, either* $q_i \geq 0$ *and* $p_i = F(p_i, q_i)$ *holds, or* $q_i = -1$ *and* $p_i$ *is different from all the variables and constants mentioned in the row for* $p_i$ *in* $F$*. For outputs,* $\varphi$ *expresses that, for each* $i$*,* $q_i$ *equals the smallest* $j$ *for which* $p_i = F(p_i, j)$ *holds, if there is such a* $j$*, and* $q_i$ *equals* $-1$ *otherwise.*

The reader may check that in the above definition $\Psi$ satisfies the four conditions from Definition 2 (use that the domain of the parameters is unbounded).

## 5.2 Abstraction Learning Algorithm

In our CEGAR algorithm, we begin with the abstraction generated by the table in which, for each $p \in P$, the corresponding row is empty. As a technical trick, we assume that all concrete traces of the SUT are started by a special event instance $\gamma(c_1, \cdots, c_{|C|})$, where $c_i$'s are constants of $C$. We define the *skeleton* of a trace to be the sequence of respective event terms of event instances of that trace. Two traces are said to be *conformable* if they have the same skeletons.

**Definition 9** (position)**.** *For a trace $\sigma$, we define a* position $\pi \in \mathbb{N} \times \mathbb{N}$ *to be an ordered pair $(\alpha, \beta)$ in which $\alpha$ is the index of an action in the trace and $\beta$ is the index of a parameter in that action. $\mathsf{Pos}(\sigma)$ is the finite set of all positions in trace $\sigma$. For two positions $\pi_1$ and $\pi_2$ of $\mathsf{Pos}(\sigma)$, we define $\leq_\sigma$, such that*

$$\pi_1 \leq_\sigma \pi_2 \Leftrightarrow \alpha_1 \leq \alpha_2 \wedge \alpha_1 = \alpha_2 \rightarrow \beta_1 \leq \beta_2.$$

*$\leq_\sigma$ is a total ordering on $\mathsf{Pos}(\sigma)$. We denote $\pi_1 <_\sigma \pi_2$ when $\pi_1 \leq_\sigma \pi_2$ and $\pi_1 \neq \pi_2$*

*For each position, we have a parameter that is the name of the parameter in the event term and we refer to it by $\mathsf{par}_\sigma(\pi)$. Furthermore, we have the value of that parameter in the position, notation $\mathsf{val}_\sigma(\pi)$. For the special action $\gamma$, $\mathsf{par}_\sigma$ returns the constant name and $\mathsf{val}_\sigma$ returns the constant value.*

**Definition 10** (visibility)**.** *A position $\pi$ is* f-visible *is $\sigma$, if it is the first occurrence of $\mathsf{par}(\pi)$ in $\sigma$. On the other hands, $\pi$ is* l-visible *from $\pi'$, if it is the last occurrence of $\mathsf{par}(\pi)$ in $\sigma$ before $\pi'$*

To find the right entries of abstraction table, we observe the traces where the current abstraction table present non-deterministic behavior to the learner. To observe non-determinism and to find new entries for abstraction table, we introduce edges on traces.

**Definition 11** (edge)**.** *An edge is an ordered pair $(\pi_1, \pi_2)$ of positions where $\pi_2$ is called the* head *and $\pi_1$ is called the* tail *of the edge. We impose the condition $\pi_2 <_\sigma \pi_1$ on the edges; in the other words, the edges are assumed to be backward.*

*For each trace we define two sets of edges:*

1. *Green Edges: $(\pi_1, \pi_2)$ is* green *if $\mathsf{val}(\pi_1) = \mathsf{val}(\pi_2)$ and either*

$$v^f_{\mathsf{par}(\pi_2)} \in F(\mathsf{par}(\pi_1)) \wedge \pi_2 \text{ is f-visible,}$$

*or*

$$v^l_{\mathsf{par}(\pi_2)} \in F(\mathsf{par}(\pi_1)) \wedge \pi_2 \text{ l-visible from } \pi_1;$$

*or $\mathsf{par}(\pi_2)$ is a constant of $C$ and $\mathsf{par}(\pi_2) \in F(\mathsf{par}(\pi_1))$.*

2. *Black Edges: $(\pi_1, \pi_2)$ is* black *if*

   - *$\pi_1$ has no outgoing green edge,*
   - *$\mathsf{val}(\pi_1) = \mathsf{par}(\pi_2)$, and*
   - *$\pi_2$ is either f-visible or l-visible from $\pi_1$.*

The implementation follows the two phases of LearnLib: the membership phase and the equivalence phase. During the membership phase, LearnLib poses membership queries to the SUT to construct an hypothesis. Once an hypothesis has been constructed, it generates some long test sequences to check if the hypothesis is correct. The concretization of abstract input actions is done

differently for the two types of queries. During membership queries the algorithm only selects fresh values for concrete actions. That is, when $F(\mathsf{par}(\pi)) = \perp$ for some position $\pi$ in the abstract trace, $\mathsf{val}(\pi)$ is set to a fresh value in the concrete trace. When $F(\mathsf{par}(\pi)) \neq \perp$, the value in the concrete trace is set to the corresponding value according to the abstraction, $\mathsf{val}(\pi) := \mathsf{val}(F(\mathsf{par}(\pi)))$.

Abstracting the concrete outputs from the SUT becomes trivial when all concrete values are distinct. The algorithm simply looks through the trace and finds a value equal to the output value and assigns the abstraction for the input value.

During the equivalence phase, instead of selecting fresh values when $F(\mathsf{par}(\pi)) = \perp$, random values are selected. The abstraction of concrete outputs is done using the hypothesis supplied from LearnLib. If the SUT generates an output which the hypothesis disagrees with, a counterexample has been found. Now there are 2 possibilities. It can be a counter example showing that the hypothesis learned by LearnLib is incorrect, or it can show that $\mathcal{T} \parallel \mathcal{A}$ is not behavior deterministic. To determine which case it is, the trace is converted into a membership query by converting all values in the trace into distinct values (except values present in green edges) and rerunning the trace on the SUT. Only if the resulting output has the same abstraction as the original counterexample, it is forwarded to LearnLib. If however the abstraction is different, the counterexample trace is a witness that the current abstraction is too coarse.

Once a counterexample is found the abstraction needs to be refined. This is done by identifying black edges which are only present in the trace, and trying to remove them by introducing fresh values. Black edges which can not be removed are added as new entries in the abstraction table.

The following lemma is crucial for termination of our algorithm. In the finest abstraction, in which the table is completely filled, there are no black edges and hence can be no counterexample, that is, $\mathcal{T} \parallel \mathcal{A}$ is behavior deterministic!

**Lemma 3.** *If two conformable traces have the same green edges and black edges, then their abstractions are identical.*

# 6 Experiments

We have implemented and applied our approach to infer models of realistic systems represented as Mealy machine style SIAs. For each system abstractions of the input and output have been learned automatically, which will be exemplified by means of the Session Initiation Protocol (SIP) as presented in [3]. Initially, no abstraction for the input is defined in the learner, which is translated to all parameter values being -1. As a result every parameter in every input action is treated in the same way and the mapper randomly selects a fresh concrete value. The learner proceeds by sending output queries in this way and constructs the abstract Mealy machine shown in Figure 2. When the hypothesis is checked for correctness, parameter values in test traces may be duplicated, which may lead to non-deterministic behavior. In the SIP experiment, the input sequence *IINVITE, IACK, IPRACK, IPRACK* leads in one case to an *O481* output and

in the other case to an *0200* output, see Figure 3. To resolve this problem, we need to refine the input abstraction. Therefore, we identify the green and black edges for both traces and try to remove black edges that are present in one trace, but not in the other. The algorithm first successfully removes black edge No. 1 by replacing the parameter value of the second input with a fresh value and observing the same output as before. However, removing black edge No. 2 changes the final outcome of the trace to an *O481* output. As a result, we need to refine the input abstraction by adding an equality check between the first parameter of the last *IINVITE* message and the first parameter of an *IPRACK* message to every *IPRACK* input. Apart from refining the input alphabet, every concrete output parameter value is abstracted to either a constant or a previous occurrence of a parameter. The abstract value is the index of the corresponding entry in the abstraction table. After every input abstraction refinement, the learning process needs to be restarted. We proceed until the learner finishes the inference process without getting interrupted by a non-deterministic output.

Table 1 gives an overview of the systems we learned with the number of input refinement steps, total output queries and total time needed. We have checked that all models inferred are bisimilar to their SUT. For this purpose we combined the learned model with the abstraction and used the CADP tool set[1] for equivalence checking.

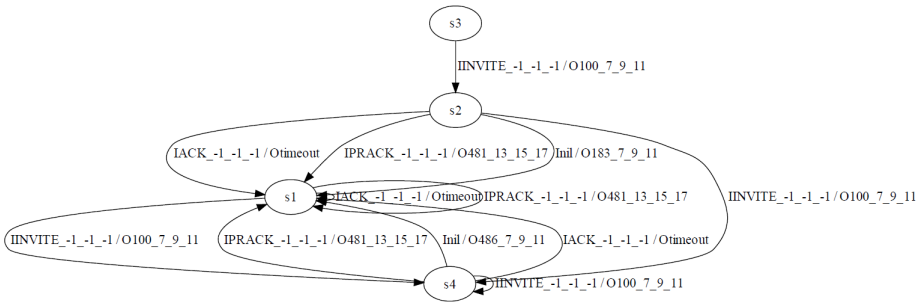


Figure 2: Hypothesis of SIP protocol

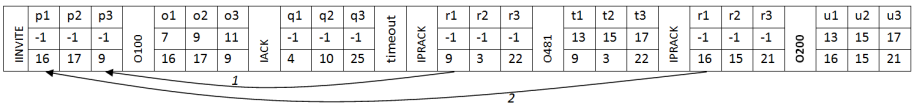


Figure 3: Non-determinism in SIP protocol

[1] http://www.inrialpes.fr/vasy/cadp/

| System under test | Input refinements | Total output queries | Total time |
|---|---|---|---|
| Alternating Bit Protocol - Sender | 1 | 193 | 18 seconds |
| Alternating Bit Protocol - Receiver | 2 | 145 | 33 seconds |
| Alternating Bit Protocol - Channel | 0 | 31 | 28 seconds |
| Biometric Passport | 3 | 2199 | 30 seconds |
| Session Initiation Protocol | 2 | 897 | 13 seconds |
| Farmer-Wolf-Goat-Cabbage Puzzle | 4 | 688 | 59 seconds |

Table 1: Learning statistics

# Paper E

# Model-Based Testing of Industrial Transformational Systems

Petur Olsen[*]

*Department of Computer Science*
*Centre for Embedded Software Systems*
*Aalborg University*
*Aalborg, Denmark*
petur@cs.aau.dk

Johan Foederer

*Test Automation*
*Océ-Technologies B.V.*
*Venlo, The Netherlands*
johan.foederer@oce.com

Jan Tretmans[**]

*Model-Based System Development*
*Radboud University*
*Nijmegen, The Netherlands*
tretmans@cs.ru.nl
*Embedded Systems Institute*
*Eindhoven, The Netherlands*

[*]Work performed while visiting Radboud University and Océ-Technologies B.V.

[**]This work has been supported by the EU FP7 under grant number ICT-214755: Quasimodo.

**Abstract** We present an approach for modeling and testing transformational systems in an industrial context. The systems are modeled as a set of boolean formulas. Each formula is called a clause and is an expression for an expected output value. To manage complexities of the models, we employ a modeling trick for handling dependencies, by using some output values from the system under test to verify other output values. To avoid circular dependencies, the clauses are arranged in a hierarchy, where each clause depends on the outputs of its children. This modeling trick enables us to model and test complex systems, using relatively simple models. Pairwise testing is used for test case generation. This manages the number of test cases for complex systems. The approach is developed based on a case study for testing printer controllers in professional printers at Océ. The model-based testing approach results in increased maintainability and gives better understanding of test cases and their produced output. Using pairwise testing resulted in measurable coverage, with a test set smaller than the manually created test set. To illustrate the applicability of the approach, we show how the approach can be used to model and test parts of a controller for ventilation in livestock stables.

# 1 Introduction

Océ is a leading company in designing and producing professional printers. As the complexity of these printers grows, both due to features added and due to the requirement to support several input formats and backwards compatibility, the task of effectively testing the printer controller becomes very difficult. In this paper we present a model-based approach to improve the testing of the controller software of Océ printers.

We consider the part of the controller which processes input job descriptions and sends commands to the hardware. The system considered is in its abstract form a simple function. It takes a set of parameter values as input and computes a set of output parameter values. Input parameters are specific settings to a print job (number or pages, duplex/simplex, etc.), and the output is the description, in terms of output parameters, of the actually printed job. The dependencies between inputs and outputs are not trivial, and as the number of input parameters is over 100 and the number of output parameters is 45, the size of the system makes testing a difficult task.

The controller is modeled using a set of constraint clauses on the input parameter values, in the form of boolean formulas. Each clause relates a set of input values to an expected output value. The approach that we take in this paper is similar to that of QuickCheck [30] and Gast [61], both of which are automatic testing tools for functional programming languages, that generate random test cases. Both tools are less suited for use at Océ, since, being based on functional programming languages, they are cumbersome to integrate into existing test frameworks, whereas randomness makes structured generation of test cases and coverage determination more challenging. This led us to implement an internal prototype in Python to handle the testing.

Others have tried similar approaches to testing real world applications, such as Lozano et al. [68], who model a financial trading system with constraint systems. This leads us to believe that the approach is applicable to other types of systems as well. To further evaluate this approach we analyze how it can be used to model parts of the controller software for a ventilation system for livestock stables.

While the final goal is to detect faults in the SUT, this has not been the main focus in this project. Rather the focus has been to take steps toward creating a maintainable, large-scale model-based testing environment. It was not our aim to compare numbers of bugs found in model-based and manual testing.

This paper presents the problem of testing printer controller software. We present an approach to modeling the controller as a set of boolean formulas, including a modeling trick to enable us to make relatively simple models for the complex system. We present the testing process and how the desired coverage can be achieved. Additionally we present some discussions on using model-based testing in an industrial setting, and which benefits this approach has given Océ. Finally, to illustrate the applicability of our approach, we show how it can be adopted to model and test part of the controller software for a ventilation system for livestock stables.

## 2 Problem Description

The problem is to test the controller of Océ printers. Océ produces professional printers, an example of which is shown in Figure 1.

- A is the input module with four input trays.
- B is the actual printer module.
- C is an output location called the High Capacity Stacker (HCS).
- D is an output location which supports stapling, called the Finisher.

This example is a small configuration of a printer. Several input modules can be attached and different output locations with different finishing options are supported.

The controller in these printers basically has two tasks: (*i*) handling the printing queues, and (*ii*) processing an input job description and sending the corresponding commands to the printing hardware. The part of the controller handling the printing queues can be seen as a reactive system, which continually monitors for job descriptions, sends them through the job processor to the printing hardware, and allows the user to perform actions on a user interface, for instance to cancel a job. This part can be modeled using some form of state machine. The part handling job processing, however, does not operate reactively. It accepts one job description at a time, and produces output for that job. Such a system can be seen in its abstract form as a simple, stateless function, accepting a set of input parameter values and returning a set of output

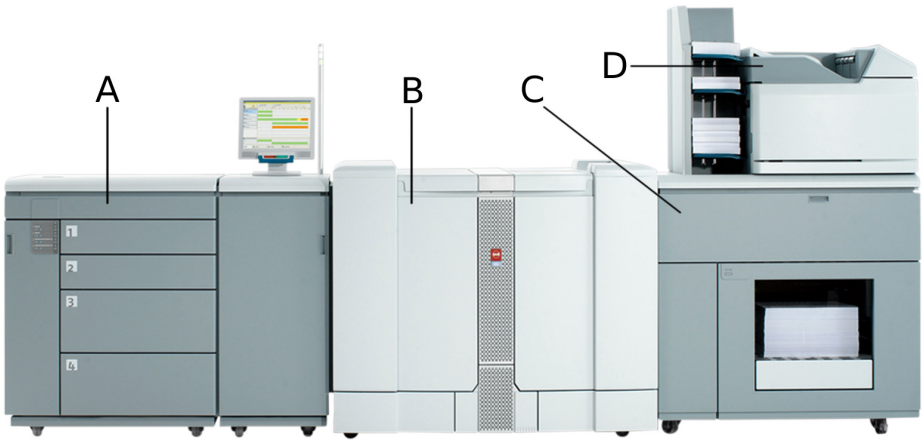


Figure 1: VarioPrint 6250. A) Input module with four trays. B) Printer module. C) High capacity stacker output location. D) Finisher output location.

parameter values. This is the part of the controller which this project focuses on, and which will be tested.

A *job description* consists of two parts: a document in a Printer Description Language (PDL) format and an optional *ticket* describing how to print the document. Several PDL and ticket formats are supported, each supporting different features and using different formats for expressing features. Some features are supported in both the PDL and the ticket, requiring the job processor to handle contradictions. Example input parameters include output location, stapling, and punching.

As output the job processor presents a set of parameter values for each sheet to be printed. These values are sent to the printer hardware which prints the job as specified. Example output parameters also include output location, stapling and punching, however the relationship between the inputs and outputs is not as simple as it might seem.

First, there are the contradictions. Stapling, for instance, can be specified both in the PDL and in the ticket, in which case the ticket will overrule the PDL. While stapling is enabled, an output location can be selected which does not support stapling, in which case the output location is overruled to one which does support stapling. However, there might not be any output locations attached to the printer which support stapling, in which case the stapling is disabled and the output location is as specified. Just for these two simple parameters we already have a lot of cases.

In addition to contradictions, there are different formats for specifying values. Specifying stapling in a ticket, for instance, has ten possible values: None, Top, TopLeft, Left, . . . , and Saddle. The stapling output from the job processor only has five: None, Portrait, Landscape, Booklet, and Saddle (the orientation of the paper and the output location determine where the specific staple is lo-

cated). Similarly other PDLs and ticket formats might have different formats for specifying stapling. Translating between these formats is not trivial.

On top of all these are the settings of the job processor. For instance the limit for the number of pages which the printer can staple can vary. Several other settings are available in the job processor.

It is clear that the job processor needs to handle all peculiarities in the input, as well as any settings of the printer. The job processor needs to support all configurations of printers, and needs to be able to print any job on any configuration, albeit possibly with some functionalities disabled. The configuration and settings of the printer can be seen as inputs to the job processor. Adding these to the PDL and ticket, and looking at all available parameters, the number of parameters for the job processor comes to well over 100. These facts make the job processor a very complex system, and testing such a system is not trivial.

## 2.1 Testing at Océ

The current testing process at Océ involves running a job description on a simulation of the hardware. When running the job processor on the simulated hardware, the output is presented in the form of a so-called APV file. This APV file contains all parameters for each printed sheet. Currently there are 45 parameters in the APV file.

The resulting APV file is analyzed manually. If deemed correct it is saved as a reference for future test runs. In subsequent automatic test runs the output can be checked against the reference and the result of the test can be determined.

There are several issues with this testing process. The first is maintainability when updating the job processor to support more parameters. This requires all test cases to be updated to support this parameter. Secondly, changing some requirements, which lead to failing test cases, requires the new APV file to be manually analyzed again. This manual analysis is very time consuming and error prone. It occurs that errors survive through the development process because of faulty analysis of the APV file.

The execution time of the test cases is also becoming an issue. Nightly runs, executing the complete set of test cases, have to finish in the morning, to present the results to the engineers. At the current number of test cases some of these runs do not complete in time. Due to expansions and new developments the number of test cases is expected to double, in the near future. This poses big requirements to the computer farm running the nightly tests, and requires expensive expansions. Therefore it is desirable to reduce the number of test cases, but the quality of the complete test set must not suffer.

Currently a *test case* is a Python script which sets up the printer, generates one or more job descriptions, and sends them to the controller in a specified order. These test cases are designed by test engineers who know the system intimately. The test cases are designed to find likely errors, and are very specifically designed, such that a failing test case gives some hint to where the error occurs. For instance, a test case might focus on stapling by generating several job descriptions with different stapling positions. If this test case fails the error

is most likely in the stapling module. This gives very specific test cases and to get good coverage it requires a lot of test cases. This leads to the desire to have structurally generated test cases which have some measure of coverage, while minimizing the number of test cases.

In the current framework there is no uniform way of defining test cases. This stems from the different formats for PDLs and tickets, and the fact that current test cases are directly coded at a low level in Python. Since these are often generated in batches in for-loops, it can be difficult for other testers and developers to understand exactly what a test case does. This leads to problems with understanding test cases, and once a test case fails, it can also be troublesome to understand precisely what the parameters of the failing job were.

This presents four areas where Océ wants to improve their testing process:

- maintainability,
- execution time,
- coverage, and
- understanding of test cases.

We will improve on these aspects, by implementing a model-based approach to testing.

# 3 Modeling the Controller

The job processor is in its abstract form a simple, stateless function. It takes a number of input parameter values and computes a number of output parameter values. We modeled the job processor as a collection of Boolean formulas, where each formula specifies the value for one output parameter through an implication with on the left-hand side a conjunction of input parameter constraints, and on the right-hand side an output parameter constraint:

$$i_1 = v_1 \wedge i_2 = v_2 \wedge \cdots \wedge i_n = v_n \Rightarrow u_p = v_p \tag{E.1}$$

This formula expresses that the expected output of parameter $u_p$ is value $v_p$ if input parameters $i_j$ have values $v_j$ for $1 \leq j \leq n$, respectively. Each of these formulas is called a *clause* in the model.

For integer parameters we also allow comparisons like $i_j \leq v_j$, e.g., to refer to equivalence classes of input parameters. As an example, a simplified clause of the staple position could look like this:

$$\begin{aligned}(Staple = TopLeft \wedge SheetCount \leq 100) \Rightarrow \\ StaplePos = Portrait\end{aligned} \tag{E.2}$$

This specifies that the output parameter for Staple Position *StaplePos* has value *Portrait* if the input parameter *Staple* is *TopLeft* and there are less than 101 sheets.

For integer output parameters we allow the expected output to be calculated by a function on the input parameters. For instance if the *Plexity* is set to *Duplex* (printing on both sides of the paper), *SheetCount* becomes half of the number of printed pages: $SheetCount = \lceil Pages/2 \rceil$.

The actual job processor model has many more parameters and also more possible values for the parameters, resulting in many more clauses. A complete job processor model consisting of such a collection of clauses must first be verified for completeness and consistency, i.e., checked whether the collection indeed specifies a function from input parameters to output parameters, but such a verification is orthogonal to testing.

Satisfaction of the model has been used as oracle for our testing process. This means that the model is instantiated with actual output parameter values of the job processor implementation, together with the corresponding input parameter values. (Section 4 will deal with choosing input values). If all clauses hold the test passes; if a clause does not hold then the test fails and the output parameter specified in the false clause is wrong.

## 3.1 Dependencies

As seen above the model for staple position depends on two input values, and the complete model is even bigger. If we have a look at a simplified clause of the output location *OutputLoc*:

$$\begin{aligned}(TicketOutputLoc = HCS \wedge Staple = TopLeft \wedge \\ SheetCount \leq 100) \Rightarrow \\ OutputLoc = Finisher\end{aligned} \tag{E.3}$$

then we can see that it depends on the input parameters *Staple* and *SheetCount*. This is because the output location should only be overridden if a staple was requested, and the printer is actually able to staple. This causes a chain of dependencies, where the output location clause must contain all – transitive – dependencies in its clause. These chains clutter the clauses and make modeling cumbersome, since there are a lot of these type of dependencies. To simplify the clauses we can observe that a part of (E.3) can be substituted with (E.2). Substituting $(Staple = TopLeft \wedge SheetCount \leq 100)$ for $StaplePos = Portrait$ we get the simpler clause:

$$\begin{aligned}(TicketOutputLoc = HCS \wedge StaplePos = Portrait) \Rightarrow \\ OutputLoc = Finisher\end{aligned}$$

We can see that the output location actually depends on the output parameter *StaplePos*. Formally, we allow $(i_j = v_j)$ from Equation E.1 to refer to input- and output parameters.

One potential problem arises with this approach. If there are circular dependencies, we can not trust the results. To avoid circular dependencies we arrange all clauses in a hierarchy, where the leaves have no dependencies and parents

depend on the parameters of their children. As long as this hierarchy is kept, it is safe to use some output parameters to verify other output parameter values. This approach simplifies the clauses significantly, and enables us to model these complex systems.

# 4 Testing

Testing a job processor implementation involves three steps:

- Selecting input values,
- executing the SUT with the input values, and
- verifying output from the SUT using the model.

Executing the SUT is done using the existing framework for automatic testing at Océ. Verifying the outputs is done using the model, as explained in the previous section. To select input values we look at the complete set of input parameters supported by the model, and the domains of these parameters. Instantiating each parameter constitutes a single test case. This can be done randomly, to generate a set of test cases, or it can be done structurally, based on some coverage criterion.

It has been shown in several projects [33, 34, 41, 27, 102] that most software errors occur at the interaction of a few factors, i.e. most errors are triggered by particular values for only a few input parameters, whereas the error is independent from the values of the other input parameter. Some projects report up to 70% of bugs found with two or fewer factors and 90% with three or fewer [33, 62], others show up to 97% of bugs found with only two factors [102]. This, combined with the fact that the number of test cases needs to be minimized, leads to combinatorial testing techniques, such as pairwise (or more generally n-wise) testing. The number of test cases in n-wise testing for fixed n grows logarithmically compared to exponential growth for testing all possible combinations.

The coverage of output parameters is not guaranteed by using the combinatorial testing technique. Pairwise testing does, however, tend to cover most of the unintended uses of the system, and many of the paths which lead to special cases, whereas manually generated test cases tend to focus on normal operation of the system. Once a test suite has been created, the output coverage can be analyzed, and the test suite can be updated to add any required coverage.

Test cases are generated based on a set of input parameters and their domains. A test case is an assignment of each input parameter to a single value from its domain. A *test specification* is a set of relations between input param-

eter name and the discrete domain of that parameter:

$$\begin{gathered}(P^1, \{v_1^1, v_2^1, \ldots, v_{n_1}^1\}) \\ (P^2, \{v_1^2, v_2^2, \ldots, v_{n_2}^2\}) \\ \vdots \\ (P^m, \{v_1^m, v_2^m, \ldots, v_{n_m}^m\}).\end{gathered}$$

Given such a test specification, algorithms can generate a set of test cases which cover all pairs of values. That is, for every $v_k^i$ and $v_l^j$ where $i \neq j$, $P^i = v_k^i$ and $P^j = v_l^j$ for at least one test case.

For instance if we have three input parameters: $TicketOutputLoc$, $Staple$, and $SheetCount$, the test specification could be:

$$\begin{gathered}(TicketOutputLoc, \{Finisher, HCS\}) \\ (Staple, \{None, Top, Left\}) \\ (SheetCount, \{\leq 100, 101\})\end{gathered}$$

Each pairwise combination of values, e.g. $TicketOutputLoc = Finisher$ and $Staple = Left$, must be present in at least one test case. In this case six test cases are needed. Generating complete coverage would require 12 test cases.

### 4.1 Diagnosis

The automatic test case generation based on a job specification can be used as a tool in diagnosis. Reducing the domain of one or more parameter in the job specification, can provide information about the fault. As an example consider the job specification above, and consider a fault has been found with one of the generated test cases. Reducing the domain of $Staple$ to $\{None\}$, and re-running test case generation and test case execution, can help locate the fault in the SUT. If, for instance, none of these new test cases finds the fault, it tells us that the fault only occurs when a staple is requested. This tells us that the error is either in the staple module, or occurs at the interaction between the staple module and some other part of the controller. Using this approach the test engineers can easily generate new sets of test cases to help locate bugs in the SUT.

## 5 Implementing the Test Tool

Other projects to improve the testing process, have previously been carried out at Océ. Experiences from these projects have shown that integrating tooling with existing frameworks can be difficult and cumbersome, and introducing new tools and frameworks requires some learning effort from the engineers. This led us to implement a prototype tool for this project by hand. The existing

framework for automated testing has a lot of tooling and libraries for testing the job processor, therefore it was decided to integrate the prototype into this framework. The existing framework is written in Python, so Python is the language of choice.

The clauses are implemented as if-then-else statements. To give a logical grouping of clauses, all clauses pertaining to the same output parameter are grouped into the same Python class. This way a class is said to *check* an output parameter by implementing all clauses for that parameter. To ease implementation several output parameters can be checked by the same class.

The entire set of actual input values and output values is passed to each class. This enables the class to access any values needed in the clauses to verify their respective output value. The classes access the values they need and then go through a series of if-then-else statements that implement the clauses in the formal approach. Once an expected value is found, it is checked against the actual output value, and an appropriate response is added to the return set.

The return set contains $OK$ or $Error$ responses from each class, and is returned back to the test system. Here it can be analyzed and all errors found by the model, can be returned to the engineer.

Once an output value has been verified, it is removed from the set of output values, which is passed to subsequent classes. This feature has two effects. First, it enables us to check whether all output values have been verified, by examining if the set is empty when all classes have been invoked. Second, it enforces the hierarchy required to detect circular dependencies as explained in Section 3.1. This is enforced since a circular dependency would result in a missing value in the latter class in the circle. This also means that classes with dependencies need to be invoked first, and classes with no dependencies are invoked last.

## 5.1 Test case generation

N-wise testing has currently been implemented using the tool `jenny`[1]. `jenny` is an executable which accepts as input, the domain size of each input parameter and generates a set witnessing n-wise coverage of all input parameter values. Currently `jenny` is always executed for pairwise coverage. This could be extended, if the coverage requirements increase. A wrapper has been written around `jenny`. A test specification is passed to the wrapper, which generates a `jenny` query for the domain sizes of the parameters. `jenny` returns a set of test cases which are translated back into the specific values in the test specification. As an example consider the test specification:

$$\begin{aligned}(TicketOutputLoc, \{Finisher, HCS\})\\ (Staple, \{None, Top, Left\})\\ (SheetCount, \{\leq 100, 101\})\end{aligned}$$

The wrapper generates a query for `jenny` with three parameters with domain size two, three, and two respectively. `jenny` generates six test cases:

[1] http://burtleburtle.net/bob/math/jenny.html

```
1a 2a 3b
1b 2c 3a
1b 2b 3b
1a 2b 3a
1b 2a 3a
1a 2c 3b
```

Each line represents a test case. The number represents the input parameter. The letter represents the value of the parameter. The test case `1b 2c 3a` is translated into ($TicketOutputLoc = HCS, Staple = Left, SheetCount =\leq 100$).

This way test cases are generated based on the test specification as created by the test engineer. In the case of diagnosis the engineer can reduce the domain sizes in the test specification, and rerun the wrapper to get a new set of test cases.

## 5.2 Run Time

The time required to execute a single test case is highly dependent on the number of pages printed in that test case; depending on the hardware running the simulator, printing a single page can take up to half a second. This fact means that test cases with a lower number of pages are preferred. In the model for testing the stapling module the equivalence classes for the number of sheets are:

- 1 (too few sheets),
- 2 (too few sheets for duplex),
- $3 - 100$,
- $>= 101$ (too many sheets).

Including all these in the pairwise test case generation would include many jobs with 100 pages or more. However, it can be observed that if the staple limit works for a single test case, it most likely works for all test cases. This observation leads us to implement the possibility to add manually generated jobs to the test set. By adding two jobs with 100 and 101 sheets printed, we reduce the equivalence classes for staple limit to three and by choosing a low value for the equivalence class $3 - 100$ the number of sheets printed can be kept low in all other jobs. This dramatically reduces the time required to execute the test suite, while still keeping acceptable coverage.

## 5.3 Invalid Test Cases

Since the PDL and ticket formats support different features, it is possible to generate invalid test cases. For instance, one ticket format supports accounting options, so we need to have input parameters for accounting. However, if we

generate a test case with accounting activated, while using a ticket format which does not support accounting, the test tool is unable to generate the job for the controller, since activating accounting in this ticket format is not possible. This is an invalid test case, since it can not be executed on the SUT.

These invalid combinations need to be removed from the pairwise coverage. This is because the pairs covered by an invalid test case are not executed on the system. Excluding simple combinations of parameter values is supported in the current version. More complex exclusions, such as employed by e.g. AETG [33], ATGT [29], or Godzilla [40] might be required in the future.

# 6 Status and Discussion

It was chosen to focus on modeling the stapling capabilities of the printers, as an initial step. The currently implemented models support four parameters: PDL format, page count, output location, and staple location. The four parameters have domain sizes two, three, three, and eight respectively. This set is pairwise covered in 27 test cases. To cover the upper page limits for stapling four test cases are manually added, bringing the total number of test cases generated from this project to 31.

It is difficult to say precisely how this coverage compares to the current set of test cases, as a lot of test cases touch stapling, while testing other areas of the controller as well. Of the manually generated test cases a total of 170 test cases use stapling. Looking at all the parameters and parameter values used in these 170 test cases, we observe 11 parameters with domains ranging from two to nine. Getting pairwise coverage of all these input parameters can be achieved with only 86 test cases. These parameters are not currently supported by the model, so the test cases can not be executed at this time. This shows us that once the models are extended, the number of test cases can be reduced. Determining if the fewer number of test cases will locate as many or more errors will require further work.

The choice of implementing a prototype in Python proved very useful. This also supports previous experience in similar projects. Integrating with the existing Océ framework was very easy. In the current version of the tool, implementing the models as Python classes is cumbersome and requires some copy-paste. With further development and refactoring, we expect to build a viable environment for developing models.

Using a model-based approach gives the engineers far better understanding of the test cases and their outcome. The new framework gives better overview of which parameter values are selected in each test case, and the models can be used to give a hint as to where an error is located in the code. The controllable test case generation can also be used for analysis, to find the precise interactions between parameters which cause the error.

The issues with maintenance are also improved, as making changes to the requirements only requires updating the model, then all test cases should work. Updating and implementing the models is not trivial, and errors in the model

could cause false positives and false negatives. However, since the same models are used by all test cases, it is more likely that errors in the model will be found.

Currently only pairwise coverage is supported. As the usage of this approach grows in Océ, it will be seen how effective this coverage is in locating faults. It might be the case that the coverage needs to be supplemented by manually generated test cases, or replaced by a different coverage measure. Currently no analysis of output coverage is done. This requires engineers to manually examine the test cases, and possibly supplement with additional cases.

Even though the focus in this project was not to detect faults in the SUT, one unknown fault has been located. The unknown fault has not been located by the old set of manual test cases. This also indicates that the coverage criteria might be good.

Based on the advantages of the model-based approach, Océ has decided to continue development of this prototype, and extend models to continue improving the testing process.

# 7 Modeling a Livestock Stable Controller

Since our approach shows promising results for modeling and testing printer controllers, and the literature shows similar approaches used in other types of systems, we wanted to examine if our approach could be applied in other companies. We have initiated contact with a company designing and producing ventilation systems for livestock stables, to examine how the approach applies there. The controller for these systems is split into several components, each of which monitors some input sensors and controls some output actuators. The inputs are continuous measurements of e.g. temperature. The outputs are either *on/off* values or a percentage value describing how much power should be given to, for instance, a ventilator. Calculation of the two types of output are done in a standard way.

For *on/off*-type of controllers there are two important parameters: $t$, and $T_\delta$. The value of $t$ describes when the output should be activated. To avoid oscillation between *on* and *off*, $T_\delta$ describes how far below $t$ the input has to fall before deactivating the output. It can be observed that between $t - T_\delta$ and $t$ this controller shows nondeterministic behavior. This nondeterminism can be seen as an internal state in the controller, storing its previous output value. For inputs between $t$ and $t - T_\delta$ the controller outputs the same value as previously. For inputs below $t - T_\delta$ the output is always *off*, and above $t$ the output is always *on*. Figure 2 illustrates the possible values.

For percentage-type of controllers the output value is a linear function of the input. The function is described by two parameters: $p$ and $P_\delta$. The value of $p$ describes when the output must start increasing. $P_\delta$ describes how far above $p$ the output must reach 100%. Below $p$ the output is always 0%, above $p + P_\delta$ the output is always 100%. In between the output grows linearly from 0% to 100%. Figure 3 illustrates the function.

Components can have several inputs based on the same patterns, to form

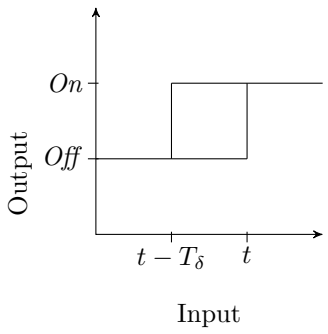


Figure 2: Graph for *on/off* values

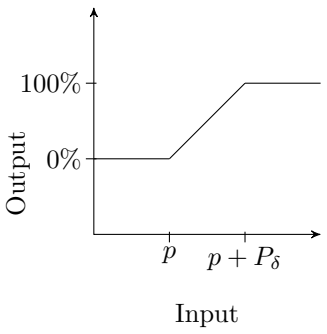


Figure 3: Graph for percentage values

more complex components. For instance, a component can have a temperature reading and a humidity reading as input, and can activate a ventilation fan as output. The ventilation fan should be activated when the temperature reaches above some value while the humidity is below some value. Both inputs will have $T_\delta$ values for when the fan should be deactivated again.

Currently test cases for the system are generated manually. A test engineer examines the parameters of the component and generates a set of inputs generating an acceptable coverage, and generates corresponding expected outputs. Subsequently the test cases are executed automatically and the expected outputs are compared to the actual outputs.

This type of system can be modeled within our framework, with a single modification. We need to handle the nondeterministic behavior. This can be done by representing the model as a hybrid automaton[9] by handling the state as an input. We make a fresh input parameter to represent the state of the model. The domain of this parameter is the state space of the model. This way the clauses in the model can depend on the state the system is in and act accordingly. However, in this simple setting this seems like too complex a solution. We only need a single state variable; a Boolean. We only need to know the value of this variable one time step backward. This can be easily be handled in the current setting of transformational systems. The problem with adding this functionality is that pairwise test case generation will not work since test cases become traces, where each step depends on the previous one. Some work needs to be done to find a good way to generate test cases for this type of system. The test case generation needs to generate valid test cases and provide coverage of the data in input parameters as well as coverage of the state space of the model.

With this modification a large set of manually generated test cases, can be automatically generated from simple models. Future test cases can easily be created by instantiating the model with the required values.

This shows that the approach is indeed applicable for different types of systems, even though it was developed for testing printer controllers. The diversity

of the SUTs shows that there are potentially several industrial areas where similar approaches could be applied to improve the testing process.

# 8 Conclusion

This paper has presented the initial steps towards a model-based testing framework for testing printer controllers at Océ. The approach has proved promising in improving the testing process and the quality of test cases. Advantages of the approach include: improved maintenance, reduced number of test cases, measurable coverage, and better understanding of the test results. While the approach seems promising at improving the testing process, further work is needed to state this for certain. While test cases can now be generated based on coverage requirements, it is unclear if the generated test set will locate as many errors as the old test cases. Also development of the model requires some significant effort, but it is expected to prove valuable in the long run. Based on the outcome of this project, Océ has decided to continue with the model-based approach. Finally, we illustrated that the approach is useful for modeling and testing controller software for ventilation systems in livestock stables. This diverse usefulness of our approach indicates that several other industrial areas could benefit from similar approaches.

The connection to the current way of working at Océ, and usability of the methods by Océ in their current environment, were among the main requirements and starting points of this project. This did not always lead to the most sophisticated, or theoretically new solutions, and sometimes even to ad-hoc solutions, e.g. to establish the connection to the existing Python tooling. Future work will include the use of more sophisticated approaches, such as using SAT-solvers or SMT tooling to solve, check, and manipulate Boolean formulas.

# Bibliography


[1] Property Specification Language: Reference Manual. Technical report, 2004. http://www.eda.org/vfv/docs/PSL-v1.1.pdf.

[2] Property Specification Language Tutorial. Technical report, 2004. http://www.project-veripage.com/psl_tutorial_1.php.

[3] F. Aarts, B. Jonsson, and J. Uijen. Generating models of infinite-state communication protocols using regular inference with abstraction. In A. Petrenko, J.C. Maldonado, and A. Simao, editors, *22nd IFIP International Conference on Testing Software and Systems, Natal, Brazil, November 8-10, Proceedings*, volume 6435 of *Lecture Notes in Computer Science*, pages 188–204. Springer, 2010.

[4] F. Aarts, J. Schmaltz, and F.W. Vaandrager. Inference and abstraction of the biometric passport. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation - 4th International Symposium on Leveraging Applications, ISoLA 2010, Heraklion, Crete, Greece, October 18-21, 2010, Proceedings, Part I*, volume 6415 of *Lecture Notes in Computer Science*, pages 673–686. Springer, 2010.

[5] Fides Aarts and Frits W. Vaandrager. Learning i/o automata. In Paul Gastin and François Laroussinie, editors, *CONCUR*, volume 6269 of *Lecture Notes in Computer Science*, pages 71–85. Springer, 2010.

[6] Parosh Aziz Abdulla and Richard Mayr. Minimal cost reachability/coverability in priced timed petri nets. In *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, FOSSACS '09, pages 348–363, Berlin, Heidelberg, 2009. Springer-Verlag.

[7] S.B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 27:509–516, 1978.

[8] Martin Alt, Christian Ferdinand, Florian Martin, and Reinhard Wilhelm. Cache Behavior Prediction by Abstract Interpretation. In *SAS '96: Proceedings of the Third International Symposium on Static Analysis*, pages 52–66, London, UK, 1995. Springer-Verlag.

[9] Rajeev Alur, Costas Courcoubetis, Thomas Henzinger, and Pei Ho. Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In Robert Grossman, Anil Nerode, Anders Ravn, and Hans Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 209–229. Springer Berlin / Heidelberg, 1993. 10.1007/3-540-57318-6_30.

[10] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, April 1994.

[11] Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. The benefits of relaxing punctuality. *J. ACM*, 43(1):116–146, January 1996.

[12] Rajeev Alur and Thomas A. Henzinger. Real-time logics: complexity and expressiveness. In *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 390 –401, jun 1990.

[13] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. *J. ACM*, 41(1):181–203, January 1994.

[14] Rajeev Alur, Gerard J. Holzmann, and Doron Peled. An analyzer for message sequence charts. In *SOFTWARE CONCEPTS AND TOOLS*, pages 304–313, 1996.

[15] Dana Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[16] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[17] T. Ball and S. Rajamani. The SLAM toolkit. In *CAV*, pages 260–264. Springer, 2001.

[18] Gerd Behrmann, Agnès Cougnard, Alexandre David, Emmanuel Fleury, Kim G. Larsen, and Didier Lime. Uppaal-tiga: time for playing games! In *Proceedings of the 19th international conference on Computer aided verification*, CAV'07, pages 121–125, Berlin, Heidelberg, 2007. Springer-Verlag.

[19] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[20] Gerd Behrmann, Kim Larsen, and Jacob Rasmussen. Priced timed automata: Algorithms and applications. In Frank de Boer, Marcello Bonsangue, Susanne Graf, and Willem-Paul de Roever, editors, *Formal Methods for Components and Objects*, volume 3657 of *Lecture Notes in*

*Computer Science*, pages 162–182. Springer Berlin / Heidelberg, 2005. 10.1007/11561163_8.

[21] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 87–124. Springer Berlin / Heidelberg, 2004.

[22] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal — a tool suite for automatic verification of real-time systems. In Rajeev Alur, Thomas Henzinger, and Eduardo Sontag, editors, *Hybrid Systems III*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer Berlin / Heidelberg, 1996. 10.1007/BFb0020949.

[23] T. Berg, O. Grinchtein, B. Jonsson, M. Leucker, H. Raffelt, and B. Steffen. On the correspondence between conformance testing and regular inference. In M. Cerioli, editor, *Fundamental Approaches to Software Engineering, 8th International Conference, FASE 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3442 of *Lecture Notes in Computer Science*, pages 175–189. Springer, 2005.

[24] J. A. Bergstra and J. W. Klop. Algebraic methods: theory, tools and applications. chapter ACP: a universal axiom system for process specification, pages 447–463. Springer-Verlag New York, Inc., New York, NY, USA, 1989.

[25] Thomas Bøgholm, Henrik Kragh-Hansen, Petur Olsen, Bent Thomsen, and Kim G. Larsen. Model-based schedulability analysis of safety critical hard real-time java programs. In *JTRES '08: Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems*, pages 106–114, New York, NY, USA, 2008. ACM.

[26] Guillaume Brat and Willem Visser. Combining static analysis and model checking for software analysis. In *Proc. ASE 2001*, pages 262–271. IEEE Computer Society, 2001.

[27] Kevin Burr and William Young. Combinatorial test techniques: Table-based automation, test generation and code coverage. In *Proceedings of the Intl. Conf. on Software Testing Analysis and Review*, pages 503–513. West, 1998.

[28] J. Byg, K.Y. Joergensen, and J. Srba. An efficient translation of timed-arc Petri nets to networks of timed automata. In *Proceedings of the 11th International Conference on Formal Engineering Methods (ICFEM'09)*, LNCS. Springer-Verlag, 2009.

[29] Andrea Calvagna and Angelo Gargantini. A logic-based approach to combinatorial testing with constraints. In Bernhard Beckert and Reiner

Hähnle, editors, *Tests and Proofs*, volume 4966 of *Lecture Notes in Computer Science*, pages 66–83. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-79124-9_6.

[30] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 268–279, New York, NY, USA, 2000. ACM.

[31] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71, 1981.

[32] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[33] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The aetg system: An approach to testing based on combinatorial design. *IEEE Trans. Softw. Eng.*, 23:437–444, July 1997.

[34] D.M. Cohen, S.R. Dalal, J. Parelius, and G.C. Patton. The combinatorial design approach to automatic test generation. *Software, IEEE*, 13(5):83 –88, September 1996.

[35] Andreas Engelbredt Dalsgaard, Mads Chr. Olesen, Martin Toft, René Rydhof Hansen, and Kim Guldstrand Larsen. Metamoc: Modular execution time analysis using model checking. In Björn Lisper, editor, *WCET*, volume 15 of *OASICS*, pages 113–123. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010.

[36] Werner Damm and David Harel. Lscs: Breathing life into message sequence charts. *Form. Methods Syst. Des.*, 19:45–80, July 2001.

[37] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 1990.

[38] A.M. Davis. *Software requirements: objects, functions, and states*. PTR Prentice Hall, 1993.

[39] L. de Alfaro and T.A. Henzinger. Interface automata. In V. Gruhn, editor, *Proceedings of the Joint 8th European Software Engineering Conference and 9th ACM SIGSOFT Symposium on the Foundation of Software Engineering (ESEC/FSE-01)*, volume 26 of *Software Engineering Notes*, pages 109–120, New York, September 2001. ACM Press.

[40] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 17(9):900–910, 1991.

[41] I. S. Dunietz, W. K. Ehrlich, B. D. Szablak, C. L. Mallows, and A. Iannino. Applying design of experiments to software testing: experience report. In *Proceedings of the 19th international conference on Software engineering*, ICSE '97, pages 205–215, New York, NY, USA, 1997. ACM.

[42] M.D. Ernst, J.H. Perkins, P.J. Guo, S. McCamant, C. Pacheco, M.S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1-3):35–45, 2007.

[43] Harald Fecher, Jens Schönborn, Marcel Kyas, and Willem-Paul de Roever. 29 new unclarities in the semantics of uml 2.0 state machines. In *Proceedings of the 7th international conference on Formal Methods and Software Engineering*, ICFEM'05, pages 52–65, Berlin, Heidelberg, 2005. Springer-Verlag.

[44] A. Finkel and P. Schnoebelen. Well-structured transition systems everywhere! *Theoretical Computer Science*, 256(1-2):63–92, 2001.

[45] Thom Frühwirth and Slim Abdennadher. Principles of constraint systems and constraint solvers, 2005.

[46] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Trans. Comput.*, 31(1):48–59, 1982.

[47] Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65, Paris, France, July 2001. Springer.

[48] Object Management Group. *Unified Modeling Language.* http://www.uml.org/, 1997.

[49] E. Harel, O. Lichtenstein, and A. Pnueli. Explicit clock temporal logic. In *Logic in Computer Science, 1990. LICS '90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 402 –413, jun 1990.

[50] Lise Tordrup Heeager and Peter Axel Nielsen. *Agile software development and its compatibility with a document-driven approach? A case study*, page 205. Monash University Press, 2009.

[51] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[52] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 58–70, New York, NY, USA, 2002. ACM.

[53] Anders Hessel, Kim Guldstrand Larsen, Marius Mikucionis, Brian Nielsen, Paul Pettersson, and Arne Skou. Testing real-time systems using uppaal. In *Formal Methods and Testing*, pages 77–117, 2008.

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

[55] F. Howar, B. Steffen, and M. Merten. Automata Learning with Automated Alphabet Abstraction Refinement. In *Verification, Model Checking, and Abstract Interpretation (VMCAI'11), January 23-25, 2011, Austin, Texas, USA*, 2011.

[56] H. Hungar, O. Niese, and B. Steffen. Domain-specific optimization in automata learning. In W.A. Hunt Jr. and F. Somenzi, editors, *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 315–327. Springer, 2003.

[57] C. Norris Ip and David L. Dill. Better verification through symmetry. *Form. Methods Syst. Des.*, 9:41–75, August 1996.

[58] B. Jonsson. Compositional specification and verification of distributed systems. *ACM Transactions on Programming Languages and Systems*, 16(2):259–303, March 1994.

[59] Bengt Jonsson and Joachim Parrow. Deciding bisimulation equivalences for a class of non-finite-state programs. *Inf. Comput.*, 107(2):272–302, 1993.

[60] Toni Jussila, Jori Dubrovin, Tommi Junttila, Timo Latvala Latvala, and Ivan Porres. Model checking dynamic and hierarchical uml state machines. In *Proceedings of the 3rd Workshop on Model Design and Validation (MoDeVa 2006)*, 2006.

[61] Pieter Koopman, Artem Alimarine, Jan Tretmans, and Rinus Plasmeijer. Gast: generic automated software testing. In *Proceedings of the 14th international conference on Implementation of functional languages*, IFL'02, pages 84–100, Berlin, Heidelberg, 2003. Springer-Verlag.

[62] Richard Kuhn and Michael Reilly. An investigation of the applicability of design of experiments to software testing. In *Proceeding of the 27th NASA/IEEE Software Engineering Workshop*. IEEE, 2002.

[63] Sabine Kuske. A formal semantics of uml state machines based on structured graph transformation. In *Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, «UML» '01, pages 241–256, London, UK, UK, 2001. Springer-Verlag.

[64] K.G. Larsen, M. Mikucionis, and B. Nielsen. Online testing of real-time systems using UPPAAL. In *Formal Approaches to Testing of Software*, Linz, Austria, September 21 2004. Lecture Notes in Computer Science.

[65] Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Model-checking for real-time systems. In *Fundamentals of Computation Theory*, pages 62–88, 1995.

[66] M. Leucker. Learning meets verification. In F.S. de Boer, M. M. Bonsangue, S. Graf, and W.P. de Roever, editors, *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*, volume 4709 of *Lecture Notes in Computer Science*, pages 127–151. Springer, 2006.

[67] C. Loiseaux, S. Graf, J. Sifakis, A. Boujjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6(1):11–44, 1995.

[68] Roberto Castañeda Lozano, Christian Schulte, and Lars Wahlberg. Testing continuous double auctions with a constraint-based oracle. In *Proceedings of the 16th international conference on Principles and practice of constraint programming*, CP'10, pages 613–627, Berlin, Heidelberg, 2010. Springer-Verlag.

[69] Maik Merten, Bernhard Steffen, Falk Howar, and Tiziana Margaria. Next generation learnlib. In *Proceedings of the 17th international conference on Tools and algorithms for the construction and analysis of systems: part of the joint European conferences on theory and practice of software*, TACAS'11/ETAPS'11, pages 220–223, Berlin, Heidelberg, 2011. Springer-Verlag.

[70] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[71] Robin Milner, Joachim Parrow Y, and David Walker Z. Modal logics for mobile processes. *Theoretical Computer Science*, 114:149–171, 1993.

[72] E. Newcomer and I. (chairs) Robinson. Web services atomic transaction (WS-atomic transaction) version 1.2, 2009. http://docs.oasis-open.org/ws-tx/wstx-wsat-1.2-spec.html.

[73] E. Newcomer and I. (chairs) Robinson. Web services business activity (WS-ws-businessactivity) version 1.2, 2009. http://docs.oasis-open.org/ws-tx/wstx-wsba-1.2-spec-os/wstx-wsba-1.2-spec-os.html.

[74] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[75] Flemming Nielson and Hanne Riis Nielson. Model checking *is* static analysis of modal logic. In C.-H. Luke Ong, editor, *FOSSACS*, volume 6014 of *Lecture Notes in Computer Science*, pages 191–205. Springer, 2010.

[76] Oliver Niese. *An Integrated Approach to Testing Complex Systems*. PhD thesis, University of Dortmund, 2003.

[77] Petur Olsen, Kim G. Larsen, Marius Mikucionis, and Arne Skou. Present and absent sets: Abstraction for data intensive systems suited for testing. In R. Huuck, G. Klein, and B. Schlich, editors, *Doctoral Symposium on Systems Software Verification (DS SSV'09)*, volume AIB-2009-14 of *Aachener Informatik Berichte*, pages 26–28, Aachen, Germany, 2009. RWTH Aachen University.

[78] Petur Olsen, Kim G. Larsen, and Arne Skou. Present and absent sets: Abstraction for testing of reactive systems with databases. *Electr. Notes Theor. Comput. Sci.*, 264(3):53–68, 2010.

[79] Carl Adam Petri and Wolfgang Reisig. Petri net. *Scholarpedia*, 3(4):6477, 2008.

[80] Amir Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57, 1977.

[81] Harald Raffelt, Bernhard Steffen, Therese Berg, and Tiziana Margaria. Learnlib: a framework for extrapolating behavioral models. *Int. J. Softw. Tools Technol. Transf.*, 11(5):393–407, October 2009.

[82] Lihua Ran, Curtis Dyreson, and Anneliese Andrews. Autodbt: A framework for automatic testing of web database applications. *Lecture Notes in Computer Science*, 3306/2004:181–192, 2004.

[83] Lihua Ran, Curtis Dyreson, Anneliese Andrews, Renée Bryce, and Christopher Mallery. Building test cases and oracles to automate the testing of web database applications. *Inf. Softw. Technol.*, 51(2):460–477, 2009.

[84] Wolfgang Reisig. Petri nets and algebraic specifications. *Theor. Comput. Sci.*, 80(1):1–34, March 1991.

[85] H. G. Rice. Classes of recursively enumerable sets and their decision problems. In *Transactions of the American Mathematical Society*, volume 74, pages 358–366, mar. 1953.

[86] A.W. Roscoe. Modelling and verifying key-exchange protocols using csp and fdr. *Computer Security Foundations Workshop, IEEE*, 0:98, 1995.

[87] W. W. Royce. Managing the development of large software systems. In *Proceedings of IEEE WESCON*, volume 26, pages 1–9, August 1970.

[88] Ingo Schinz, Tobe Toben, Christian Mrugalla, and Bernd Westphal. The rhapsody uml verification environment. In *Proc. SEFM 2004*, pages 174–183. IEEE, 2004.

[89] David Schmidt and Bernhard Steffen. Program analysis *as* model checking of abstract interpretations. In Giorgio Levi, editor, *Static Analysis*, volume 1503 of *Lecture Notes in Computer Science*, pages 351–380. Springer Berlin / Heidelberg, 1998. 10.1007/3-540-49727-7_22.

[90] David A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 38–48, New York, NY, USA, 1998. ACM.

[91] Steve Schneider. Verifying authentication protocols in csp. *Software Engineering, IEEE Transactions on*, 24(9):741–758, 1998.

[92] Bernhard Steffen. Data flow analysis as model checking. In Takayasu Ito and Albert Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 346–364. Springer Berlin / Heidelberg, 1991. 10.1007/3-540-54415-1_54.

[93] C. Thrane and U. Sorensen. Slicing for uppaal. In *Student Paper, 2008 Annual IEEE Conference*, pages 1 –5, feb. 2008.

[94] J. Tretmans. Test generation with inputs, outputs, and repetitive quiescence. *Software–Concepts and Tools*, 17:103–120, 1996.

[95] Jan Tretmans. *A Formal Approach to Conformance Testing*. PhD thesis, University of Twente, 1992.

[96] Jan Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29(1):49 – 79, 1996. ¡ce:title¿Protocol Testing¡/ce:title¿.

[97] Jan Tretmans. Testing concurrent systems: A formal approach. In *CONCUR*, pages 46–65, 1999.

[98] Jan Tretmans. Model based testing with labelled transition systems. In Robert Hierons, Jonathan Bowen, and Mark Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 1–38. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-78917-8_1.

[99] Jan Tretmans and Ed Brinksma. Torx: Automated model based testing. In *Proceedings of the First European Conference on Model-Driven Software Engineering*, page 12 pp., 2003.

[100] Uppaal-TRON. *Uppsala University and Aalborg University*. http://www.cs.aau.dk/ marius/tron/, 2006.

[101] Gregor von Bochmann and Jan Gecsei. A unified method for the specification and verification of protocols. In *IFIP Congress*, pages 229–234, 1977.

[102] Dolores R. Wallace and D. Richard Kuhn. Failure modes in medical device software: an analysis of 15 years of recall data. In *ACS/ IEEE International Conference on Computer Systems and Applications*, pages 301–311, 2001.

[103] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The Worst-Case Execution Time Problem - Overview of Methods and Survey of Tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.

[104] Sergio Yovine. Kronos: A verification tool for real-time systems. (kronos user's manual release 2.2). *International Journal on Software Tools for Technology Transfer*, 1:123–133, 1997.