CS-Report 01-014
Third Dutch Model Checking Day, Eindhoven, November 7, 2001-11-17
Proceedings

J.C.M. Baeten
D. Bosnacki
J.P.M. Voeten
L. Holenderski (Eds.)
Third Dutch Model Checking Day,
Eindhoven, November 7, 2001
Proceedings

by

J.C.M. Baeten, D. Bosnacki, J.P.M. Voeten and L. Holenderski (Eds.)

01/14

ISSN 0926-4515

All rights reserved
editors: prof.dr. J.C.M. Baeten
prof.dr. P.A.J. Hilbers

Reports are available at:
http://www.win.tue.nl/win/cs

Computer Science Reports 01/14
Eindhoven, October 2001
Preface

This report contains the preliminary proceedings of the third Dutch Model Checking Day, held on 7th November 2001 at the Technische Universiteit Eindhoven.

Model checking is an automatic technique for verifying hardware and software systems. The advance of the research in this area in the past few years has lead to a significant improvement of the model checking tools. Successful applications of model checking have been reported in the verification of a wide variety of systems, like complex sequential circuit designs and communication protocols. An important evidence of the great practical potential of model checking is the development of in-house model checking tools within the major companies from the information and telecommunication industry.

The objective of the Model Checking Day was to bring together researchers and practitioners from academia and industry who are interested in model checking. The presentations featured both practical and theoretical advances in the area. This includes new techniques and methodologies, as well as experience with their application in various areas, such as embedded systems, communication protocols, hardware components, production processes, etc.

Besides this, the Model Checking Day provided an opportunity to exchange experiences, and to have discussions about new ideas and the latest developments in the area.

This proceedings contains contributions related to the presentations on this day, details are given in the table of contents. The Model Checking Day received generous support from the Formal Methods Group of the Technische Universiteit Eindhoven and the research school IPA (Institute for Programming research and Algorithmics). At this point I would like to thank the members of the program committee Dragan Bošnački (TU/e Computer Science), Leszek Holenderski (Philips Research) and Jeroen Voeten (TU/e Electrical Engineering), and the secretary Elize Russell (TU/e Computer Science) for all their work.

Jos Baeten, chair program committee,
Formal Methods Group,
Division of Computer Science,
Technische Universiteit Eindhoven.
Contents

Preface ................................................................................. i
J. Baeten

Contents ............................................................................ iii

Abstracting C with abC .................................................... 5
D. Dams, W. Hesse and G. Holzmann

Model Checking of Reusable Hardware Components ............... 19
M. van Hulst

Analyzing Markov Chains by Model Checking ......................... 37
H. Hermanns

Parametric Model Checking of Fair TCTL ............................... 56
R.F. Lutje Spelberg and W.J. Toetenel

Black Box Checking .................................................................. 71
D. Peled, M.Y. Vardi and M. Yannakakis
Abstracting C with abC

Dennis Dams

joint work with William Hesse, Gerard Holzmann
Model Checking

= verifying temporal correctness properties by exhaustively searching the state-transition graph of a model of a system

“Testing looks for probable behaviors, model checking looks for possible behaviors.”
Challenges

- Requirement specification problem (not considered in this presentation)
- Model construction problem

For SW: model = abstraction of code

- source-to-source transformation

→ high level of automation possible

1. parsing
2. abstraction ⇐
3. simplification
4. conversion
Abstracting away variables

- Why: smaller state vector, hopefully smaller state space (encouraged by Gerard Holzmann’s experiences with *FeaVer*)
- What: removal of memory locations specified by identifiers (scalars, pointers, whole records and arrays)
- Basic idea: values of expressions may become unknown
Overapproximation

(Behavior of) abstract program should overapproximate concrete program in order to avoid false positives

- Have to introduce nondeterminism at points where rest of program depends on unknown expressions

(Assumptions: no out-of-bounds array indexing, limited pointer arithmetic)
Example: Introducing nondet.

```c
int x, y;

...  
if (x != 0) { ... }
...  
y = x;
```

➤ Want set of abstracted variables to be "almost closed" under dependencies
Transforming the program

- Only expressions (including assignments) are transformed (defs./decls. currently ignored)

⇒ Transform expressions so that "concrete side effects" are retained
Example: Transforming exprs.

\[
\begin{align*}
\text{if } (x \neq 0) \{ \ldots \} & \quad \rightarrow \quad \text{if } (\text{NONDET}) \{ \ldots \} \\
y = x & \quad \rightarrow \quad \text{drop} \\
y = x & \quad \rightarrow \quad y = \text{NONDET} \\
\ldots a[c++]\ldots & \quad \rightarrow \quad c++ \\
\text{or } (c++,\text{NONDET}) & \quad \text{or drop}
\end{align*}
\]
Tool architecture

c tree (parser)

ctree

(unknown)

label

cmdline

transform program

Pass 1

Pass 2

Pass 3

abC

datanode

utils

logging

type
A reminder about C

• An lvalue is an expression that can occur on the left-hand side of an assignment

• (assignment: \( E1 = E2 \))
Consider $E_1 = E_2$.
Which (additional) variables are to be abstracted?

In any (non-constant) expression that is an lvalue
and/or is of pointer type, there is an identifier from
which the expression's value is reachable (= constructed by applying operations as in the expression).
We call this the key identifier (if any).
(P.1, 2) Key ids: Definition

- keys(ID) = if is_lval(ID) (incl. pntr) → \{ID\}
  else ∅
- keys(E.ID) = keys(E)
- keys(E++) = if is_pntr(E) → keys(E)
  else ∅ (not an lvalue)
- keys((E1?E2:E3)) = keys(E2) ∪ keys(E3)
- keys(E1=E2) = if is_pntr(E2) → keys(E2)
  else ∅
(P.1, 2) Infer new abstractions

\[ E_1 = E_2 : \]

Key ids of \( E_1 \) must be abstract if
(1a) value of \( E_2 \) unknown, or
(1b) lvalue of \( E_1 \) unknown.

Key ids of \( E_2 \) must be abstract if
\( E_2 \) of pointer type and
(2a) lvalue of \( E_1 \) unknown, or
(2b) value of \( E_2 \) unknown.

Rules for \( +=, ++ \), etc. based on these.
(P.1, 2) Examples

```c
int x, y, a[N], i, *p, *q[N];
(1a) y = x
(1b) a[i] = 0
      ++a[i]
(2a) p = &y  (may be followed by *p = ...)
(2b) p = q[i]  (same as (1b))
```
Pass 3: Transforming expressions

Idea: Keep if known, otherwise filter out concrete side effects

For every operator:
If one of the subexpressions is unknown, then (recursively) handle each subexpression and put the results in a comma expression.
(P.3) Recursive def. of transform.

- \( \text{tr}(\, \text{ID} \, ) = \) if unknown(ID) \( \rightarrow \) NONDET
  else ID
- \( \text{tr}(\, \text{E1 OP E2} \, ) = \)
  if unknown(E1) or unknown(E2) \( \rightarrow \)
    (\( \text{tr}(\, \text{E1} \, ), \text{tr}(\, \text{E2} \, ), \text{NONDET} \))
  else E1 OP E2
- \( \text{tr}(\, \text{E1 = E2} \, ) = \)
  if unknown(E1) \( \rightarrow \)
    (\( \text{tr}(\, \text{E1} \, ), \text{tr}(\, \text{E2} \, ), \text{NONDET} \))
  else E1 = \( \text{tr}(\, \text{E2} \, ) \)
Output

- Transformed program
- Inferred abstractions

(demo?)
What next

• Display measure of dependency
• Scoping, defs/decls (initializers), remove unnecessary header stuff (tools?)
• Constant data detection
• Interprocedural abstraction
• Other abstractions than “all or nothing”
Model Checking
Reusable Hardware Components

Marten van Hulst

Electronic Design & Tools
Philips Research Labs Eindhoven, NL
November 7, 2001
Outline

• Reusable blocks in hardware design
• Examples: PI-Bus protocol, converters
• Assertional methods
• Conclusions
Hardware Verification

• 70% of total design effort is in verification
• Simulation is main workhorse
• Verification Gap is widening
• New, effective methods are sought
Model Checking vs. Simulation

- Exhaustive search
- Needs formal spec
- Handles small models
- Labor intensive
  - Model reductions and abstractions
  - Spec development
- Full coverage infeasible
- Needs testbench/vectors
- Works on bigger models
- Quicker initial result but gets harder towards full coverage
Where/How to apply MC

- Relatively small designs that have complex behavior
  - Control logic
  - State Machines
- Designs for which specs can be reused
  - Bus compliance of reusable IP (Intellectual Property)
Why focus on IP blocks

- Reused multiple times
  - Bugs have more impact if not found
  - Robustness is an issue
- Separate property development and usage
  - Development can be done centrally
  - Smoother introduction into design flow
PI BUS

- Peripheral Interconnect (on-chip) bus
- Standard for linking components
  - Facilitates reuse
- Widely used within Philips
  - Toolkit available for simulation
PI BUS (2)
PI BUS (3) -- Verification

• Wrapper file used for each type of component: master, slave and controller
  – allows separate verification -- smaller size
  – identifies ‘holes’ in interface specification
• 30+ properties specified
• several errata found in state machines
PI BUS (4) -- Examples

• At most one master may be granted
• At most one slave may be selected
• No masters are granted while bus is locked
• Only granted master may lock the bus
• Selected slave must acknowledge
Device Transaction Level (DTL)

Yesterday(?):
IP cores with bus interface

1st step:
IP cores with interface(s),
Adapters to bus

2nd step:
classical busses faded out,
new communication structure can be incorporated with same IP
DTL (2) -- Adapters

- Provide glue from DTL to various buses
- dual clock domain
  - handshake logic:
    - data width conversion
    - buffering and packetization
DTL (3) -- Highlights

- Reset behavior verified for *all* clock ratios
- Scalable FIFO-verification environment
- Implementation-specific properties
  - FIFO buffer management
  - splitting transactions
MC: Current Situation

- Model Checking only used in special cases
- Pay-off considered insufficient
- Not ready for embedding in design flow

What can we do about this?
Assertions in Simulation

- Shorten debug time
- Useful on both block and system-level
- Enabler for (semi-)formal techniques
- plethora of language proposals
  - e, 0-in, Solidify HPL, Sugar, Forspec, OVL,...
Example Assertions

- **ASSERT_ALWAYS** (clk, reset_n, expr)

- **ASSERT_NO_OVERFLOW**
  
  (clk, reset_n, min, max, expr)
Assertions in Verification

• Standard language needed (Accellera)
  – allows mix and match of tools
• High degree of automation is key
• Transformation of properties into constraints and back (on block interfaces)
Conclusion and Prospects

• Model Checking currently used in special cases only

• Assertion-based methods are on the rise
  – easy integration in design process
  – opportunity for integration with formal methods
  – **Challenge**: use MC-techniques to leverage assertions
Construction and Verification of Performance and Reliability Models

Holger Hermanns

University of Twente, Faculty of Computer Science, Formal Methods and Tools Group,
P.O. Box 217, 7500 AE Enschede, the Netherlands
hermanns@cs.utwente.nl

Abstract

Over the last two decades formal methods have been extended towards performance and reliability evaluation. This paper tries to provide a rather intuitive explanation of the basic concepts and features in this area. Instead of striving for mathematical rigour, the intention is to give an illustrative introduction to the basics of stochastic models, to stochastic modelling using process algebra, and to model checking as a technique to analyse stochastic models.

1 Introduction

Modern industrial systems, such as communication networks, transport systems, or manufacturing systems, are more and more operating in a stochastic context: communication lines can break, buffers can overflow, a lorry with material for a just-in-time production line might get stuck in a traffic jam. Each of these phenomena is stochastic by nature, its absence or presence can only be predicted up to some probability. Since these stochastic phenomena have impact on the system under consideration, it is nowadays commonly agreed that the systems themselves exhibit stochastic behaviour. As a consequence, performance and reliability studies of industrial systems have to take into account that rigid assessments ("It is impossible that the system fails") only hold under unrealistic assumptions.

The construction and analysis of models suited for performance and reliability studies of real-world phenomena is a difficult task. To a large extent this problem is attacked using human intelligence and experience. Due to increasing size and complexity of systems, this tendency seems even growing: performance as well as reliability modelling becomes a task dedicated to specialists, in particular for systems exhibiting a high degree of irregularity. Traditional performance models such as queueing networks lack hierarchical composition and abstraction means, significantly hampering the modelling of systems that are developed nowadays. Some notable results and concepts have been developed [8, 36, 37, 38], but they remain isolated from the system design cycle, due to the lack of a well-founded theory of hierarchy, composition and abstraction.

On the other hand, for describing the plain functional behaviour of systems various specification formalisms have been developed that are strongly focussed on modelling systems in a compositional, hierarchical manner. A prominent example of such specification formalisms is process algebra [19]. Developed on a strong mathematical basis, process algebra has emerged as an important framework to achieve compositionality. Process algebra
provides a formal apparatus for reasoning about structure and behaviour of systems in a
compositional way.

During the last decade, stochastic process algebra has emerged as a promising way
to carry out compositional performance and reliability modelling, mostly on the basis of
continuous time Markov chains (CTMCs). Following the same philosophy as ordinary
process algebra, the stochastic behaviour of a system is described as the composition of
the stochastic behaviours of its components.

To analyse properties of formally specified models model checking is a very successful
technique to establish the correctness of the model, relative to a given set of temporal logic
properties the model is supposed to satisfy [12, 13]. Using efficient encoding techniques,
model checking has been applied to industrial size designs involving more than $10^{30}$ states.

It appears valuable to apply efficient model checking techniques also to performance
and reliability properties of industrial systems. Since performance and reliability models
are stochastic in nature, the properties of interest are stochastic as well, and have to be
described in an appropriate extension of a temporal logic. The model checking algorithm
then involves the calculation (or approximation) of probabilities of certain properties to
hold.

This paper tries to provide a rather intuitive explanation of the basic concepts and
features of stochastic models, of stochastic modelling using process algebra, and of model
checking as a technique to analyse stochastic models. For the sake of being illustrative the
paper tends to treat various fine points much more simplistic than the advanced reader
probably desires.

The paper is organised as follows. Section 2 introduces the basic concepts of stochastic
models. Section 3 exemplifies the use of process algebra for modelling stochastic phe­
nomena by means of a real-world example, and Section 4 describes the model checking
approach to analyse stochastic models. Section 5 concludes the paper. This paper is a
revised version of an invited contribution to the 5th International ERCIM Workshop on

2 Stochastic models

A stochastic model is basically a means to describe the evolution of a real-world phe­
nomenon as time\(^1\) passes, with a particular emphasis on phenomena with stochastic timing
characteristics. In other words, repeated observations of the same phenomenon can have
varying timing characteristics, but their variation exhibits a specific kind of randomness.

As an example, consider a gambler that throws a die every minute. Observing the
gambler, one might wish to study a phenomenon, such as the time it takes to throw a
$\square$. Starting the observation at some arbitrary minute, one counts the minutes till the
die shows a $\square$. Obviously, repeated observations will usually lead to different results, at
least if gambling with a fair die. Nevertheless, the variation among these observations
exhibits a specific kind of randomness: The time needed to throw a $\square$ is known to follow
the so-called geometric probability distribution.

**Probability distribution.** A probability distribution is a function that assigns a prob­
ability (a real value between 0 and 1) to each element of some given set. For instance,
the geometric probability distribution $P$ assigns probabilities to natural numbers. For the

\(^1\)It is a bit narrow minded to consider the time domain as the only possible domain of variability.
Spatial Markov processes, for instance, are used to describe the evolution of a phenomenon as its position
in some appropriate space changes, as opposed to the time.
gambler, these numbers enumerate the minutes he is already gambling (remind that he throws the die once per minute). For some $t$, $P(t)$ is the probability to see the first $D$ after $t$ minutes, and is given by:

$$P(t) = \Pr\{\text{see a } D \text{ within first } t \text{ minutes}\} = 1 - \left(\frac{5}{6}\right)^t, \quad t \in \{0, 1, 2, \ldots\},$$

or complementary,

$$1 - P(t) = \Pr\{\text{still no } D \text{ after } t \text{ minutes}\} = \left(\frac{5}{6}\right)^t, \quad t \in \{0, 1, 2, \ldots\}.$$  

For instance, the probability of not having seen a $D$ after $t = 2$ minutes, i.e., after throwing the die twice, equals $25/36$.

To make the example a bit more interesting, assume that the gambler is throwing the die somewhere outside his office. Before leaving his office he has put a note on the door, as depicted in Figure 1. In fact, his intention is to return to his office as soon as the die shows a $D$. Now let us assume that someone arrives at his door, finding the door closed. How long will he have to wait for the gambler? Probably just a minute, but probably (more likely) more than a minute, probably (unlikely) more than ten minutes. Since this experiment is governed by the above geometric distribution, the probability of having to wait more than a minute is $5/6$, the probability of waiting more than ten minutes is $(5/6)^{10}$. Figure 2 depicts these probabilities for the first 15 minutes.

Figure 2: A geometric probability distribution\textsuperscript{2}: Will the gambler still be absent at time $t$?

\textsuperscript{2}The reader familiar with the standard pictorial representation might be surprised that the plot in
Markov chain. Having explained the gambler's behaviour, we are now in the position to specify a stochastic model of his behaviour. It is depicted in Figure 3. As many other (formal or semi-formal) models, the model is a graph, consisting of states and transitions. There are two states in this model. One state represents the absence of the gambler, one represents his presence in the office. The model contains three transitions representing possible events that might induce a change of state. One transition indicates that every minute the absent gambler has a 1-out-of-6 chance to return to his office. Another transition indicates that with probability 5/6 the absent gambler will miss the office, and hence has to stay absent for at least another minute. In case he is back in his office, the third transition indicates that he stays there (ad infinitum). The small arrow on top of the left state indicates the initial state, i.e. the state occupied at time zero.

Figure 3: A discrete-time Markov chain describing the gambler’s behaviour

The stochastic model of the gambler’s behaviour is a very simple one. It is a Markov chain, named after A.A. Markov who studied models of this kind in the beginning of the last century [34]. More specific, it is a discrete-time Markov chain (DTMC), since state changes are only possible at discrete points in time: The gambler can return to his office precisely every minute only. DTMCs restrict the possible time points for state changes to a discrete subset of dense real time. As in our example, these time points are often (but not necessarily) equidistant.

Markov chain analysis. For a given stochastic model, such as a Markov chain, there is usually a variety of interesting properties that one might want to study. Two substantially different classes of properties can be distinguished. Transient analysis investigates the evolution of the model up to a given point in time. On the contrary, steady-state analysis focuses on the long-run average behaviour. It requires that on the long-run initial start-up effects (the transient phase) do not have a measurable impact.

A trivial steady-state property for the gambler is that with probability zero he will be absent on the long-run. As an example for a transient property, we have already indicated that the probability of still being absent after 10 minutes is \((5/6)^{10}\). A variant of steady-state analysis gives us that on average it takes the gambler six minutes to throw a dice. So, the sign on the office door is essentially right, the gambler will be back in six minutes, on average. (This can also be directly derived from the fact that the expected (or average) value of a geometric distribution is \(1/p\) where \(p = 1/6\) is the chance of a successful event.)

Analysis techniques. In practice, three fundamentally different techniques are used to analyse stochastic models. They differ with respect to accuracy, applicability and computational requirements. Here, we only give a concise subjective summary on differences and similarities, and refer to Jain’s textbook [33] for a more elaborate discussion.

Figure 2 (and Figure 5) tends to zero for \(t \to \infty\), instead of tending towards 1. What is depicted is not the probability distribution \(P(t)\) (nor its probability density), but its complement \(1 - P(t)\). This is done for didactic reasons.
Simulation. The stochastic model is mimicked by a simulator throwing dice and producing statistics of simulated time spent in states. The fraction of simulated time spent in a particular state is used as an estimate for the state probability. This technique is generally applicable, in particular it is suitable also for non-Markov stochastic models. However, it should be noticed that good accuracy tends to require long simulation runs, and hence limits applicability in practice: To increase the accuracy of the simulation by a factor of \(n\), one needs to increase the length of the simulation runs (and hence the run-time of the simulation) by a factor of \(n^2\).

Numerical solution. The transient or steady-state behaviour of a stochastic model is obtained by an exact or approximate numerical algorithm where model parameters are instantiated with numerical values. This approach gives accurate results in general, up to numerical precision. Typically the solution time increases logarithmically with an increase in accuracy. On the other hand, its applicability is restricted to finite Markov chains (with a few exceptions, see e.g. [22, 32]). Furthermore the number of states of the model is a limiting factor, because of computational and especially storage requirements. A very readable textbook on numerical solution methods is [41].

Analytical solution. The transient or steady state property of interest is expressed as a closed formula over the parameters of the model. This is the most simple, accurate and elegant technique. However, analytical solutions are available only for highly restricted classes of stochastic models.

Absence of memory. Markov chains are widely used as stochastic models of real-world phenomena. This is mainly because they possess a distinguishing feature that simplifies both modelling and analysis. They obey the so called memory-less property: The future evolution of a Markov chain model is independent of the past, it only depends on the state currently occupied. This property is best explained in terms of the absent gambler. The probability that the gambler returns to his office after one minute from now is 1/6, independent of the fact that someone might be waiting for him in front of his door for ten minutes (or years) already. This is a direct consequence of the fact that a fair die has no memory; the die does not change if it has not shown a \(D\) for ages. This should not be mixed with the fact that the probability of actually having to wait for ten minutes is low, \((5/6)^{10}\). Under the assumption that this unlikely case becomes reality, it still needs another six minutes waiting time on average, as the sign on the door indicates.

Discrete vs. continuous time. Discrete-time Markov chains are convenient to describe the stochastic evolution of sequential systems. In each state, the outgoing transitions define how the probability mass will be spread at the next time instant. Since DTMCs evolve in a discrete time domain, the flow of probability is not continuous, instead it possesses jumps, and remains unchanged in the time interval between two relevant time points, such as between \(t = 2\) and \(t = 3\). This is relatively convenient for sequential systems. But it is not convenient in a concurrent probabilistic setting, for both theoretical as well as pragmatic reasons.\(^3\)

As an example, imagine that the gambler's office door is checked by some customer. In case he finds the door closed he probabilistically decides to check again after either 24 or 48 seconds. Note that the basic time unit of this DTMC is 24 seconds. For instance, one might want to study the probability that the customer finds an open door after 72 seconds.

\(^3\)DTMCs are appropriate for some types of concurrent systems possessing globally synchronising clocks, such as ATM-switches or certain manufacturing systems.
Without specifying the model in all detail, we are already in the position to understand the problem: In order to develop a concurrent probabilistic model of both gambler and customer, we have to relate events that may happen at every 24 seconds to events that happen every 60 seconds. One solution is to change the basic time unit of both models to 12 seconds, the greatest common divisor of their basic time units. In other words, the gambler's model is blown up to record in 4 additional states that while being absent, four times 12 seconds pass till he may throw the die in the last twelve seconds of the minute (cf. Figure 4). After a similar change in the customer's sub-model, one can combine both models (by essentially taking the cross-product of states and the products of transition probabilities). To determine the concurrent stochastic behaviour at the next point in time (i.e. after 12 seconds) one synchronously updates the respective states in the two sub-models, because state changes now occur exactly at the same time. The probability for such a joined transition is given by the product of the transition probabilities in the sub-models.

This strategy has two practical limitations, at least. First, it tends to induce a tremendous blow-up of the size of the model, caused by the number of auxiliary states needed in general. Second, it fails if there is no greatest common divisor, for instance if the customer shows up every 11 seconds, or if time points are not equidistant. As a consequence, virtually all stochastic models of concurrent systems are developed in a continuous time domain, including models of modern computer systems (even though each component of such a system can be considered as working in discrete time, changing state according to fixed frequency clock ticks).

Continuous-time Markov chains. Continuous-time Markov chains (CTMCs) are Markov chains interpreted over continuous time, in contrast to DTMCs. They are widely used to model the stochastic behaviour of concurrent real-world phenomena, due to their mathematical simplicity, paired with modelling convenience.

How does the continuous-time variant of the gambler look like? In a continuous time setting, the absent gambler is able to return to his office at arbitrary time points. Still we may assume that he has a 1-out-of-6 chance to return within the first minute, and so on. Under these assumptions, we get the following probability distribution:

\[ Pr\{\text{still no \( \square \) after } t \text{ minutes}\} = (5/6)^t, \quad t \in \mathbb{R}^+. \]

What is this? It perfectly resembles the geometric distribution appearing in the discrete time case, but it is different. The difference is that the domain of this function is the

\footnote{Note that this change encodes some kind of memory in an otherwise memory-less model: A sequence of states is used to keep track of the time already spent in the original state.}
non-negative real line, instead of the natural numbers. In other words, the above function assigns a probability to all time points one may think of, instead of only to each minute. Hence, there is now a non-zero probability of returning within the first second already, namely $1 - (5/6)^{1/60}$. Instead of being a geometric distribution, this function belongs to the class of so-called (negative) exponential probability distributions, because $(5/6)^t$ can be rewritten to $e^{-\lambda t}$, with $\lambda = \ln 6 - \ln 5 \approx 0.18232$. The value $\lambda$ is a parameter of the distribution, usually called 'rate'. For $t < 15$, the probabilities determined by this exponential probability distribution are depicted (by the dark plot) in Figure 5. The expected value of an exponential distribution (i.e. the average duration) is $1/\lambda$, the reciprocal value of the rate. So, the (continuously gambling) gambler returns after $5.48$ minutes on average, not after six minutes.\(^5\)

Figure 5: A negative exponential probability distribution with $\lambda = \ln 6 - \ln 5$: Will the gambler still be absent at time $t$?

A continuous-time Markov chain model of this absent gambler is depicted in Figure 6. It consists of two states, and one transition. The transition represents that the gambler can return to his office with rate $\lambda$. The gambler stays absent as long as needed to throw a $0$. According to the value of $\lambda$ the probability mass flows from state to state as time passes, that is, a fraction of $1 - e^\lambda = 1/6$ of the probability mass flows from the left state to the right state per minute.\(^5\)

Figure 6: A continuous-time Markov chain describing the gambler’s behaviour

Though the above example shows one of the simplest CTMCs one can think of, it exhibits all relevant ingredients: states and transitions, the latter labelled with rates of exponential distributions. It is worth to note that – in correspondence to geometric

---

\(^5\)Remark that since the probability mass is flowing continuously, a sixth of the mass leaks prior to the first minute tick. Hence, to some extent the probability mass flows earlier than in the discrete-time case, where a sixth of the probability mass jumps a bit later, at each minute tick. As a consequence, the average time needed for the continuously gambling gambler is slightly smaller than 6 minutes. To obtain an average duration of 6 minutes, one has to adjust $\lambda$ to 1/6.

\(^6\)Since the gambler continuously tries to return to his office, there is no need to record by an explicit (looping) transition that he might fail for some (continuous) time. For CTMCs, this fact is implicit, while in the DTMC scenario it is not.
distributions – exponential distributions are memory-less: The future evolution of a CTMC model is independent of the past, it only depends on the state currently occupied. In terms of the gambler, the probability that the absent gambler returns to his office within the next minute is 1/6, independent of the fact he might have been absent for ages already.

Figure 5 allows us to illustrate the memory-less property in a pictorial way [1]. Consider the case that the gambler is still gambling after minute 10. We obtain the probability that he will still be gambling at time $10 + t$ by stretching the tail of the distribution (from time 10 to $\infty$) upwards in such a way that it reaches probability 1 for minute 10, i.e. $t = 0$. As a matter of fact, this stretching returns precisely the original distribution, as indicated by the light-grey plot in Figure 5, except that it is shifted by 10 minutes. The same graphical illustration holds for the geometric distribution, but for no other discrete or continuous distribution, because the exponential (resp. geometric) distribution is the only memory-less continuous (resp. discrete) distribution.

From a pragmatic point of view, the memory-less property is rather convenient. It simplifies analysis, but it also simplifies modelling. In particular, it fits well to concurrent stochastic phenomena: If two sub-models, both described in terms of CTMCs, are to be considered concurrently, one can simply interleave their evolution: If one sub-model changes from one state to another, the other sub-model is not affected. The fact that the latter has been staying in some state for some time (the time it took the former sub-model to change state) does not need to be recorded somehow, because it does not alter the future behaviour of the latter sub-model, due to the memory-less property.

Anyway, it should be clearly stated that absence of memory is an assumption that is by far not always justified when modelling real-world phenomena. On the other hand, exponential distributions are the most appropriate guesses if only expected values are known. This is because among all distributions with a given expected value, the exponential distribution maximises entropy [23, 39]. Actually, in performance engineering practise one hardly has measurement data at hand that allows one to determine more than an expected value.

3 Formal specification of continuous-time Markov chains

In this section we illustrate the use of formal methods to model a specific aspect of a real-world example as a CTMC. Several formal notations exist that map on CTMCs, among them stochastic Petri nets and stochastic process algebra. Here we restrict ourselves to illustrate the use of process algebra; an introduction to the Petri net based approach can be found for instance in [1]. As opposed to standard Petri nets, process algebra allows one to compose models out of smaller sub-models, by means of general composition operators such as parallel composition and choice [19], and also more specific constructs, such as exception handling [21]. We will make use of these operators to model a simplified view on the performance and reliability of the Hubble space telescope.

The Hubble Space Telescope. The Hubble space telescope (HST) [40] is an orbiting astronomical observatory operating from the near-infrared into the ultraviolet (cf. Figure 7). Launched in 1990 and scheduled to operate through 2010, the HST carries a variety of instruments producing imaging, spectrographic, astrometric, and photometric data.

7It is possible to incorporate a notion of memory into the model, similar to what we have used to realise synchronisation of DTMCs (cf. footnote 4). In this way, general non-exponential probability distributions (so-called phase-type distributions) can be represented. The price to pay for this is usually a blow up of the model.
Figure 7: The Hubble space telescope [40].

The HST was first conceived in the 1940's. It was designed and built in the 1970's and 1980's, aiming at a life span of 15 years with on-orbit servicing taking place on 3 year intervals. The HST programme is a cooperation of the National Aeronautics and Space Administration (NASA) and the European Space Agency (ESA).\footnote{Originally, the HST was designed to be returned to earth via the space shuttle every 5 years with on-orbit servicing every 2.5 years as well. This concept was later dropped as it was felt there was a too great risk of contamination and structural load to make the concept sound.}

Since the telescope was launched in April 1990, three servicing missions were carried out: in December 1993, in February 1997, and in December 1999. During the last mission the stabilising unit of the HST was repaired. This was necessary, since severe problems with the reliability of the gyroscopes contained therein had forced the HST to turn into a sleep mode.

The gyroscopes are part of HST pointing system. They provide a frame of reference to determine where it is pointing and how that pointing changes as the telescope moves across the sky. They report any small movements of the spacecraft to the HST pointing and control system. The computers then command the spinning reaction wheels to keep the spacecraft stable or moving at the desired rate in order to avoid that the telescope pointing device staggers. This is of particular importance to avoid that pictures taken by the telescope are blurred. The gyroscopes work by comparing the HST motion relative to the axes of the spinning masses inside the gyroscopes.

The HST has a total of six gyroscopes, grouped into three fine guidance sensors. They are arranged in such a way that any three gyroscopes can keep the HST operating with full accuracy. Two fine guidance sensors had been replaced already during the first servicing mission in 1993. Till the end of the second servicing mission in 1997, all six gyroscopes were working normally, but then one after the other failed. Starting from January 1999 the HST had been operating with only 3 functional gyroscopes. As a consequence of a fourth gyroscope failure on November 13, 1999, HST turned itself into a sleep mode and the science program was suspended. Without operational gyroscope the telescope would have run the risk to crash. In December 1999, a space shuttle mission was sent to the HST to replace (among others) the complete stabilising unit. This mission was successful.
In order to judge whether the problems of the HST could have been expected beforehand, one might want to study the reliability of the stabilising unit by means of an abstract stochastic model. Here we construct a simple Markov chain model of the gyroscopes, and of their controller. The model is a toy example, developed to give a flavour of Markov chain modelling with process algebra. The model is developed in the algebra of interactive Markov chains (IMC) [24, 27], an orthogonal extension of both CTMCs and basic process algebra.

**Basic processes.** Each gyroscope might fail after an exponentially distributed amount of time (it is known that exponential distributions fit relatively well to failures of technical equipment). The failure rate $\lambda$ is the same for all gyroscopes. A GYRO specification is as follows:

$$\text{GYRO} = (\lambda). \text{FAIL. STOP}$$

This specification corresponds to a graphical representation depicted in Figure 8. Apart from a transition labelled $\lambda$ representing the delay prior to failure, there is a second kind of transition, indicated by a dotted arrow labelled FAIL. In abstract terms, this transition represents the potential of interaction, i.e. of synchronising with a partner transition (labelled with the same name) in a different sub-model. The potential of interaction between sub-models is one of the well-known features offered by a process algebraic approach [7, 35].

![Figure 8: A simple interactive Markov chain describing the gyroscope's behaviour](image)

**Parallel composition.** Six of these gyroscopes coexist independently in the stabilising unit, together with a controller that keeps track of the status of each gyroscope, by means of synchronisation on FAIL. This is realised using the operator $[||\text{FAIL}||$ for synchronisation, and $||$ to denote independent parallelism (among the gyroscopes):

$$\text{STABILISER} = \text{CONTROLLER}$$

$$[||\text{FAIL}||]$$

$$(\text{GYRO}||\text{GYRO}||\text{GYRO}||\text{GYRO}||\text{GYRO}||\text{GYRO})$$

The controller counts the number of failures, and mechanically turns the telescope into sleep mode in case four gyroscopes have failed. To turn into sleep mode requires some time. For the moment we just assume an exponential distribution with rate $\mu$. We will explain shortly how to deal with other distributions. After turning on the sleep mode, the controller notifies the base station by means of a SLEEP signal. In the meantime, further gyroscope failures might occur. If the last gyroscope fails, a CRASH is assumed to be inevitable. The graphical representation of the controller is depicted in Figure 9.

$$\text{CONTROLLER} = \text{FAIL. FAIL. FAIL. FAIL.}$$

$$((\mu). \text{SLEEP. STOP} ||| \text{FAIL. FAIL. CRASH. STOP}))$$
To complete the picture, we consider the stabilising unit of the HST in the context of the base station. The base station listens to the SLEEP notification and reacts accordingly: launch a space shuttle mission to repair – and then restart – the telescope.

**BASE = SLEEP. PREPARE. LAUNCH. REPAIR. RESTART. BASE**

**Exception handling.** The complete specification consists of the STABILISER and the BASE station synchronising on SLEEP. Two events may alter the functioning of the system. If a CRASH occurs, the whole system is extinguished, but if the shuttle mission manages to repair the stabilising unit in time, the whole system will be restarted anew.

\[
\text{HST} = \begin{cases} 
\text{try} & \{ \text{STABILISER} \mid (\text{SLEEP}) \mid \text{BASE} \} \\
\text{catch} & \text{RESTART} \{ \text{HST} \} \\
\text{catch} & \text{CRASH} \{ \text{STOP} \} 
\end{cases}
\]

**Time constraints.** Of course, preparing the shuttle mission takes time, and one might wish to incorporate the expected (random) delay in the model. To do so, we can use a constraint-oriented style, as advocated in [42]. This style allows one to add constraints on the timing of certain sequences of interactions, such as between PREPARE and LAUNCH by means of a dedicated operator, defined in [27]. For instance,

\[
\text{on PREPARE delay LAUNCH by } (\nu). \text{STOP in HST}
\]

adds an exponentially distributed delay with rate \( \nu \) between PREPARE and LAUNCH. Semantically speaking, this will have the same effect as specifying BASE = SLEEP. PREPARE. (\( \nu \)). LAUNCH. REPAIR. RESTART. BASE, but it is much more modular and flexible, in particular because it can be used to impose very general time constraints, instead of only exponentially distributed ones, see [27]. In short, one can insert an arbitrary (phase-type distributed) delay between PREPARE and LAUNCH, by replacing \( (\nu). \text{STOP} \) in the above expression by some appropriate term (in fact, an encoding of the distribution as a CTMC).

For the sake of the presentation we do not add further time constraints, even though a realistic model would at least impose some nontrivial delay between LAUNCH and REPAIR, (as well as a non-exponential delay to set up the SLEEP mode.)
Extracting the Markov chain. The complete HST specification gives rise to a stochastic model, a CTMC depicted in Figure 10. It is obtained from the specification by applying the formal semantics of the process algebra, and compressing the model by means of an appropriate weak bisimulation afterwards. The states are labelled from left to right with the number of gyroscopes that are currently operational, except if the system is sleeping, or crashed.

Remark that in this CTMC the failure rate $\lambda$ appears weighted with different multiplying factors. The intuitive reason is that if six gyroscopes are operational, the time to the first failure is six times smaller than if only one gyroscope is left. This increased failure rate for multiple identical components is correctly derived by the formal approach outlined above. The necessary compression algorithm is implemented in [9].

4 Performance and reliability via model checking

In this section we illustrate the use of model checking to analyse performance and reliability properties of CTMC models. We discuss the main ingredients of this approach, and apply model checking to the simple Hubble space telescope example of Section 3.

Temporal logic. The model checking approach relies on the use of temporal logic for specifying properties one is interested in. For this purpose temporal logic provides means to specify undesired (or - dually - desired) evolutions. Typical specifications of properties are 'something undesired never happens' or 'eventually a desired state is reached'. A temporal logic specification is usually considered in the context of a given model (provided by some process algebraic specification, for instance). The mechanic verification whether a model satisfies a temporal logic specification is called model checking. It is worth to mention that basic temporal logic does not allow one to reason about delays and time points (although the name might suggest the converse). It is 'temporal' in the sense that it allows one to refer to the ordering of events as the model evolves in time.

Temporal logics for Markov chains. In the context of Markov chain models, the temporal logic approach turns into a probabilistic temporal one. It is not sufficient to decide whether 'eventually a desired state is reached'. Instead the probability of

---

9The semantics of this exception handling is defined in [17].
10As explained in [27], constructing the Markov chain requires to internalise all possible interactions beforehand. This is necessary but not always sufficient to extract a CTMC, since interactive Markov chains are strictly more expressive than CTMCs (because of the absence on non-determinism in CTMCs).
eventually reaching a desired state is much more interesting. For the gambler example in Figure 3 the standard interpretation of 'eventually the gambler will be present' would return false, because it is in principle possible to stay absent ad infinitum. However, this evolution is extremely unlikely, it has probability zero. So, a quantitative interpretation of temporal logic is needed, quantifying the likelihood of satisfying a given property. This allows one to specify properties such as 'a desired state is eventually reached with at least probability 0.95'.

Moreover, since the evolution of a Markov chain model in time is measurable (in the true sense of the word), it is possible to reason about time instances within the temporal logic. Timed properties such as 'with at most probability 0.2 the gambler will still be absent after 10 minutes' are possible.

**Continuous stochastic logic.** The continuous stochastic logic (CSL), first proposed in [2] and further refined in [3, 5] provides means to reason about continuous-time Markov chain models. It is a branching time logic based on CTL [11] with dedicated means to specify time intervals, and to quantify probability. As explained in Section 2, there are two substantially different classes of properties of a CTMC: transient and steady-state properties. Therefore, CSL provides two complementary means to quantify the probability mass: a steady-state operator $S$, to quantify the long-run likelihood, and a transient probability operator $P$.

For instance, a steady-state property $S_{\leq p}(\Phi)$ is true if the long-run likelihood of property $\Phi$ is at most $p$.\footnote{Instead of '≤' one may use arbitrary comparison operators, or specify intervals of probabilities instead.} $\Phi$ can be a basic property (usually called atomic proposition) valid (or invalid) in some state. It can also be an arbitrary nested property of the logic. The transient probability operator is used to quantify the likelihood of evolving in a specified way, from a given state and a given time point on. For example $P_{\geq p}(X \Phi)$ is true in a state if the probability of moving (in one step) to a state where $\Phi$ holds is at most $p$. Apart from $X$ there can be various other arguments for the operator $P$, such as

- $\Diamond \Phi$ quantifies the probability for evolving in such a way that eventually a $\Phi$-state is reached.
- $\Diamond [0,t] \Phi$ characterizes the amount of probability for reaching a $\Phi$-state within $t$ time units.
- $\Phi_1 U \Phi_2$ characterizes the amount of probability for evolving only along $\Phi_1$-states until a $\Phi_2$-state is reached.
- $\Phi_1 U [t_1,t_2] \Phi_2$ quantifies the probability for evolving only along $\Phi_1$-states until a $\Phi_2$-state is reached, under the additional constraint that $\Phi_1$ holds at least up to time $t_1$, and $\Phi_2$ holds at time $t_2$ the latest.

**Model checking CSL.** Model checking a CTMC with respect to a given CSL property involves various algorithms. Since the details are not of vital importance for a proper understanding of the approach – at least relative to the logical means to specify properties – we only give a concise overview of the ingredients.

As in other model checking strategies, a couple of graph algorithms are used. In addition, algorithms to quantify the probability mass of satisfying the above criteria are needed. In principle, these probabilities could be derived using simulation, numerical solution, or sometimes via analytical solutions. Since numerical solution of CTMCs is well-studied and generally applicable, it seems wise to use numerical solution methods
to model check CSL properties [5]. In this way, model checking involves matrix-vector multiplications (for $X$), solutions of linear systems of equations (for $\mathcal{O}$, $\mathcal{U}$ and for $\mathcal{S}$), and solutions of systems of Volterra integral equations (for $\mathcal{U}_{\text{vol}}$). Linear systems of equations can be solved iteratively by standard numerical methods [41]. Systems of integral equations can be solved either by piecewise integration after discretisation, or they can be reduced to standard transient analysis [3]. A prototypical model checker for CSL, $\text{EI-MC}^2$, is available [28]. We shall make use of $\text{EI-MC}^2$ to investigate properties of the Hubble space telescope.

**Properties of the telescope model.** CSL provides a rich framework to study performance and reliability properties of the HST. Here we consider a few illustrative cases. In order to allow the calculation of numerical values, we first need to fix the model parameters $\lambda$, $\mu$, and $\nu$ of the CTMC in Figure 10. Assuming a basic time unit of one year, we set $\lambda = 0.1$, i.e., we assume that each gyroscope has an average lifetime of 10 years. (Remind that $1/\lambda$ gives the average duration of an exponential distribution with rate $\lambda$.)

To turn on the sleep mode may require a hundredth of a year (a bit more than three days and a half) on average, hence we set $\mu = 100$. Further assuming that preparing the repair mission will take about two months, we set $\nu = 6$. Unless otherwise stated we consider the validity of CSL properties in the initial state, i.e. the state labelled $\delta$ in Figure 10. The state labels appearing in this figure serve as atomic state propositions for the logic.

First, let us look into long-run averages. An interesting property, often called availability, is the probability that the system will be available – i.e. neither crashed nor sleeping – on the long-run average. In CSL we assure an availability higher than $p$ by specifying

$$S > p (\neg (\text{sleep} \lor \text{crash})).$$

None of the states of the HST satisfies this property (whatever the value of $p$ may be). This should not be surprising, because the telescope is not constructed for the long run. In fact, the availability of the telescope is zero, because on the long run, the modelled telescope will crash, all the probability mass will eventually be accumulated in the crash-state (cf. Figure 10).12

While checking standard availability does not make much sense for the HST, the instantaneous availability is of interest. Instantaneous availability is a typical transient property, it is the probability that the system is operational at a given time point $t$. This time point could for instance be given by the need to observe a rare astronomical event.

Assuming that an interesting comet passes the telescope in five years, we specify

$$P_{\geq 0.95} (\leq (\text{sleep} \lor \text{crash}))$$

in order to assure that with at least probability 0.95 the telescope is neither sleeping nor crashed then. (Note that the time interval $[t, t]$ denotes just a single time point.) We compute a probability of more than 0.98 of satisfying this property, hence it is satisfied.

In the same vain, we may wonder about the probability to obtain blurred data at that time from the telescope, because less than three gyroscopes are operational, but sleep mode is not yet turned on. This is a very unlikely situation, and one might accept at most a probability of $10^{-6}$. One way of characterising the relevant states is to isolate those

---

12Generally speaking, steady-state properties provide very useful insight in the model, in particular for the widespread class of models where the probability mass can flow forever without gradually leaking into some sink (so to speak), or where more than one sink exists. Each of these sinks may in general consist of a set of mutually reachable states.
(non-sleep) state that (with positive probability) can turn on the sleep mode in the next step. This gives us

\[ P_{\leq 10^{-5}}(\Diamond^{[5,5]} \neg \text{sleep} \land P_{>0}(X \text{ sleep})) , \]

a property that is not satisfied, because the probability of being in the specified states after 5 years is about \(10^{-5}\).

Another quantity of interest is the time until first sleep, i.e., the time span before the (fully operational) telescope has to be put into sleep mode for the first time. In reality, this happened within 2.7 years: All gyroscopes were operational at the end of the second servicing mission in early 1997, and the sleep mode was turned on in November 1999. We require a less than 10% chance of such a first sleep within 2.7 years by specifying the property

\[ P_{<0.1}(\neg \text{sleep} U^{[0,2.7]} \text{ sleep}). \]

It turns out that this property is valid; \(E \vdash MC^2\) computes that the probability of a first sleep within 2.7 years amounts to about 0.03. A related question is whether it was likely not to witness any gyroscope failure within the four years between the first (1993) and the second servicing mission (1997). We answer this question by checking whether the probability to leave state 6 within 4 years is between, say, 0.3 and 0.7. (Notice that leaving state 6 corresponds to a gyroscope failure).

\[ P_{[0.3,0.7]}(\Diamond^{[0,4]} \rightarrow 6) \]

In fact, this property is invalid, because the probability of a gyroscope failure within 4 years is approximately 0.9, thus exceeding the upper bound 0.7.

As a last example property, be reminded that the HST is planned to stay on orbit through 2010. Hence, it seems worth to study whether a crash before reaching the year 2010 can hardly be expected. To do so, we check a property saying that there is at most a 1% chance that the system will crash within the next 10 years (given that the system was reset to state 6 in late 1999):

\[ P_{<0.01}(\Diamond^{[0,10]} \text{ crash}) . \]

This property is satisfied, the probability of crashing within 10 years is calculated by \(E \vdash MC^2\) to be 0.00036. Remind that the model is a toy example, and that its timing parameters are not claimed to reflect reality.

5 Concluding remarks

In this paper, we have tried to give an illustrative introduction to the basics of stochastic models, to stochastic modelling using process algebra, and to model checking as a technique to analyse stochastic models.

A few questions have not been addressed to a satisfactory extent. In particular we have skipped the discussion how to label states of a CTMC generated from a process algebra in such a way that these labels can be used in temporal logic property specifications. One solution to this problem is to move from a state-based logic towards a transition-based formalism [29]. As a further direction of research, we have extended CSL to so-called Markov reward models, where states are parametrized with costs. This extension allows one to specify a broad set of performability properties [4].

Another important issue for industrial strength formal analysis is the availability of tool support. Currently, prototypical tool support is available for both the stochastic
modelling and the analysis phase: A couple of prototypes exist that allow a process algebraic modelling of CTMCs [6, 10, 26]. So far, performance models with up to $10^7$ states have been modelled and analysed compositionally [27]. A prototypical model checker for Markov chains, E$\inf$MC$^2$, is available [28], it has been used to check the above CSL properties of the Hubble space telescope. More effort is nevertheless needed to enhance modelling and analysis convenience. In addition, it seems favourable to link stochastic features to existing modelling and analysis tools that provide an open and extensible architecture. We are currently making efforts to incorporate stochastic modelling and analysis features into the CADP toolset [9, 20, 18].

Markov chain models have been the clear focus of this paper. Their memory-less property considerably simplifies both modelling and analysis, but the property also implies that some real-world phenomena can only roughly be approximated with Markov chains. Hence, there is a need to extend the framework sketched in this paper beyond Markov models. The work of D’Argenio et al. [15, 16, 17] develops a process algebra to specify non-Markov performance and reliability models in an elegant way. So, the benefits of a process algebraic formalism extend to performance and reliability modelling in general. Anyhow, the analysis of such models needs further investigations. First results in this direction, however, indicate that numerical solution methods are impractical in general [30]. We are currently developing an open analysis environment for such specifications, linking to UPPAAAL [31] for real-time model checking and to MOBIUS [14] for discrete event simulation and numerical analysis.

Acknowledgements. Pedro R. D’Argenio, Boudewijn Haverkort and Joost-Pieter Katoen have provided very valuable comments on an earlier version of this paper.

References


54


Real-Time Model Checking based on Splitting

R.F. Lutje Spelberg and W.J. Toetenel

1 Thales Naval Nederland, Hengelo, The Netherlands, ronald.lutjespelberg@nl.thalesgroup.com
2 Delft University of Technology, Faculty of Information Technology and Systems, Delft, The Netherlands, w.j.toetenel@its.tudelft.nl

Abstract. This paper discusses a new approach to model checking of real-time systems. One of the novel aspects of our approach is the fact that an unconventional approach is chosen to deal with representing symbolic state spaces. Its key feature is that it does not use a canonical representation for representing symbolic nodes, but an alternative representation based on splitting trees. Unlike often used canonical representations, this representation is tailored for the type of exploration algorithm that we apply, namely an algorithm based on partition refinement. We describe this approach in terms of the verification problem of reachability of real-time systems. Based on this approach we have built model checkers for more complex verification problems, like parametric real-time temporal logic verification. From experiments with these tools, we found that our splitting-based model checking approach results in very efficient verification tools.

1 Introduction and overview

The main problem to be solved when building a model checker is dealing with the size of state spaces that are explored. To do so, often a symbolic model checking approach is adopted, especially if real-time systems are involved. The key characteristic of a symbolic model checker is that it explores the state space of system in terms of symbolically represented sets of states, rather than single states. Constructing a symbolic model checking algorithm involves finding solutions for the following three problems:

1. An algorithm for exploring the symbolic state space;
2. A representation for the symbolic state space that is efficient in storage and allows efficient manipulation;
3. An approach to evaluate properties over the symbolic state space.

Most published exploration algorithms are based on either backward/forward reachability analysis [2, 12], compositional techniques [11], or partition refinement techniques [1]. Several approaches based on partition refinement have been described, but seem not to have lead to tools that match the performance of the reachability analysis-based tools. However, we have shown elsewhere that partition refinement is also a powerful approach to real-time model checking [13]. We will show that the key to applying partition refinement is the usage of a
specific approach to represent symbolic state spaces. Traditionally, in symbolic model checking, canonical representations have been used. In particular, in non-real-time model checking, Binary Decision Diagrams have proven to be very successful [14], while for real-time systems, Difference Bound Matrices [6] have often been used. However, the application of a partition refinement technique enables an unconventional approach in representing state sets. In this paper, our interest is in the application of this new state space representation, called splitting trees, in combination with a partition refinement algorithm.

We present a symbolic model checking approach truly founded on splitting. It is a combination of an exploration algorithm that based on iterative splitting and a symbolic state space representation based on splitting trees. We will show that splitting trees are better fit to handle symbolic state space representation for partition refinement techniques, resulting in model checking tools that are at least as efficient as the well-known reachability analysis-based tools. Although splitting trees have been used in some verification approaches [19, 16], they where to our knowledge never applied as means to represent symbolic state spaces.

As for the third issue in building symbolic model checker, we will discuss our approach in terms of a simple model checking problem, namely that of reachability analysis. However, we have also addressed more complex verification problems. In fact, we have extended the approach to parametric model checking of fair TCTL specifications [17]. A tool was built that implements parametric fair-TCTL model checking based on this approach. However, the validity of the results presented here is rather independent of the type of verification problem. Defining our approach in terms of reachability analysis leads to simpler and more compact descriptions.

Section 2 briefly discusses the modeling language we will use for this and introduces our model checking approach. Section 3 is focused on the symbolic representation of the state space, the splitting trees. Section 4 presents practical results, and section 5 presents some conclusions.

2 Modeling systems and a model checking algorithm

2.1 Modeling systems

The definition of our modeling language abstracts from several aspects of a concrete modeling language. For example, the modeling of parallel processes, location invariants and urgency are not included. These aspects are not relevant when discussing our approach to real-time model checking. However, the tools that we build based on the approach discussed in this paper do use a far more expressive modeling language, namely XTG (extended timed graphs).

The modeling language we use belongs to the family of timed automata languages (e.g. [9,12]), which are often used in model checking approaches for real-time systems. To model the data aspects we use an abstract model in which we do not specify the exact nature of the data. We only assume that there are clock variables, since these are essential for modeling real-time systems.
The data model for our language is abstractly defined by the following domains: Given a set of variables $V$, a subset of clock variables $V_c \subseteq V$, we assume the following domains:
- $V$: a set of variables, having a subset $V_c \subseteq V$ of clock variables;
- $\text{Val}$: possible values;
- $\text{Bexpr}_V$: boolean expressions over the set of variables $V$;
- $\text{Updv}_V$: updates over the set of variables $V$;
- $\text{Env}_V: V \rightarrow \text{Val}$ valuations over $V$.

and the following semantical evaluation functions

\[
\begin{align*}
V_v[] &: \text{Bexpr}_V \rightarrow (\text{Env}_V \rightarrow \text{Val}) \\
U_v[+] &: \text{Updv}_V \rightarrow (\text{Env}_V \rightarrow \text{Env}_V)
\end{align*}
\]

which evaluate value expressions and updates over $V$. We use $\rho$ to range over valuations. If $V$ is a set of variables, with $V_c \subseteq V$ a subset of clock variables, then $\rho[+\delta]$ is written to denote that each each clock in $V$ is increased by $\delta$.

A real-time system is modeled as a tuple $(V, \rho_0, L, l_0, E)$, where
- $V$ defines a set of variables where $V_c \subseteq V$ defines the subset of clock variables
- $\rho_0: V \rightarrow \text{Val}$ defines initial values for the set of variables
- $L$ is a set of location identifiers, where $l_0 \in L$ identifies the initial location
- $E \subseteq L \times \text{Bexpr}_V \times \text{Updv}_V \times L$ is a set of edges. An edge is a tuple $(l, g, u, l')$, having a sources location $l$ a guard $g$ an update $u$ and a destination $l'$.

A timed structure is a tuple $(S, s_0, T)$, where $S$ is a set of states, $s_0$ is the initial state, and $T \subseteq S \times (\{\mu\} \cup \mathbb{R}^{\geq0})$. The operational semantics of an XTG $(V, \rho_0, L, l_0, E)$ is a tuple $(L \times \text{Env}_V, (l_0, \rho_0), T)$ where $T$ is defined as follows ($\rightarrow$ is used to denote transitions in $T$):

\[
\begin{align*}
(l, \rho) \xrightarrow{} (l', \rho') & \iff \exists (l, g, u, l') \in E. ~ V_v[g](\rho) = \text{true} \text{ and } \rho' = U_v[u](\rho) \\
(l, \rho) \xrightarrow{} (l, \rho') & \iff \rho' = \rho[+\delta] \text{ for any } \delta \geq 0
\end{align*}
\]

The first rule in the definition of $T$ specifies that a discrete transition can be taken if an outgoing edge exists, if that edge is enabled in the current valuation, and if the destination state reflects the update. The label $\mu$ is used to indicate discrete transitions. The second rule states that a $\delta$ time transition can be taken by increasing all clocks with a nonnegative value $\delta$.

### 2.2 Region graphs

Our symbolic state spaces are represented by so-called region graphs. A region graph represents a partition of a state space, for which all states in a class originate from the same control state of the system. We are only interested in region graphs that define partitions that allow us to decide the verification problem at hand. To this end will define the concept of stability. If a region
graph is stable, this means that all regions are equivalence classes with respect to our verification problem. In fact, the equivalence relation we are looking for is the greatest time-abstracting bisimulation relation as defined in [18].

To define region graphs, we need the notions of zones and regions. A zone is a set of valuations, while a region is the combination of a single location and a zone. A region can thus be seen as representing a set of states that share the same location. Thus, given an XTG with locations \( L \) and variables \( V \), a zone is an element of \( \mathcal{P}(\text{Env}_V) \), and a region is an element of \( L \times \mathcal{P}(\text{Env}_V) \).

To indicate zones, we will often use the letter \( Z \), optionally with primes and different subscripts. In the same way \( r \) is used to indicate regions. If \( r = (l, Z) \), we write \((l', \rho) \in r \) if \( l = l' \) and \( \rho \in Z \), indicating that a state is included in a region. Analogously, we write \((l, Z) \subseteq (l', Z')\) to indicate that \( l = l' \) and \( Z \subseteq Z' \).

Furthermore, we say that a zone \( Z \) respects a constraint \( c \) if either all valuations in \( Z \) satisfy \( c \) or all valuations in \( Z \) dissatisfy \( c \).

A region graph is induced by a partition of the states of a timed structure underlying a system definition. The single constraint that is put on such partitions is that all states in a particular class have to correspond to the same location. As a consequence, the vertices of a region graph can be represented as regions. Let \( (S, s_0, T) \) be a timed structure corresponding to a system \( (V, \rho_0, L, l_0, E) \). A region graph corresponding to this timed structure is a graph \( (N, E) \), where

- \( N \subseteq L \times \mathcal{P}(\text{Env}_V) \), where \( N \) defines a partition of \( S \),
- \( E \subseteq N \times N \) is given by the following definition \((r, r') \in E \) is written as

\[
\begin{align*}
\text{Edges in a region graph can be categorized as either discrete edges (those corresponding to edges of the system), and time edges (those corresponding to the elapsing of time).}
\end{align*}
\]

Region graphs can be simplified without losing any of the information it conveys. This simplification consists of leaving out time edges that are redundant from an analysis point of view. Suppose there are three regions \( r_1, r_2 \) and \( r_3 \), and that for some \( \delta_1 \) and \( \delta_2 \), there exists time edges \( r_1 \rightarrow r_2 \) and \( r_2 \rightarrow r_3 \). In that case necessarily also an edge \( r_1 \rightarrow r_3 \) exists. The latter edge can be left out, because it is 'covered' by the other two edges. Furthermore, reflexive edges can be also left out because each region will have one. Therefore their existence can also be left implicit.

A simplified region graph is a tuple \( (N, E) \), such that \( N \) is defined as above and \( E \) is defined such that for any \( r, r' \in N \):

\[
\begin{align*}
\text{It is easy to show that from any simplified region graph, the corresponding (not simplified) region graph can be derived. Since a simplified region graph is coarser than the region graph it corresponds to, using simplified region graphs causes the model checking algorithm to become more efficient.}
\end{align*}
\]
2.3 The Partition Refinement algorithm

This section presents a description of a partition refinement algorithm that checks whether or not some target location $ld$ can be reached from the initial state $s_0$. The algorithm is loosely based on minimal model generation algorithm of [4]. It builds the reachable part of the partition of the state space induced by a greatest time-abstracting bisimulation that refines an initial partition. On-the-fly it checks if the verification problem can be decided.

The algorithm starts from an initial partition and iteratively refines the partition by splitting unstable reachable classes, until all reachable classes are stable. In that case, the set of reachable classes coincides with the reachable subset of classes of the minimal partition. The initial partition is chosen in such a way that it separates states that originate from different locations, i.e. it is defined by the control graph of the system. As a consequence, the partitions occurring in the algorithm can be represented by region graphs, since its classes can be represented by regions. The algorithm operates on-the-fly, which means that the partition is built while the state space is being explored.

Fig. 1 presents a simplified pseudo-code description of algorithm for reachability analysis. To keep the description of the algorithm simple, the bookkeeping of region graph edges is not made explicit. They are assumed to be added when appropriate. The following structures are used by the algorithm.

- $R_i$, the region of the initial partition that contains the initial state.
- $R_d$ is the region of the initial partition that is defined by the target location $ld$ of the reachability analysis.
- $S$ is the set of nodes of which the edges have not yet been fully explored.
- Reachable($r$) indicates whether or not a region $r$ is reachable.
- Subnodes($r$) denotes the set subnodes of $r$.

Starting from the initial region graph node $R_i$, a stable region graph is constructed in an iteration of three steps: (1) exploring a new edge (and possibly a new location), (2) stabilizing the region graph, and (3) use the new information to possibly decide values of nodes. The state space is thus explored by iteratively incorporating new XTG edges. When incorporating a new edge, the algorithm will often encounter locations that were not yet visited. For such locations an initial region graph node is created. These newly created nodes are immediately split to create subnodes that respect the guards of their outgoing edges.

These splittings may cause stable predecessors of this node to become unstable, because states corresponding to these predecessor nodes may now have successors in different nodes. The stabilize function takes care of this. It splits a region that is unstable with respect to an edge to another region into smaller regions that again stable. The stabilize operation is recursive because stabilizing a node may again cause predecessor nodes to become unstable. In this manner splits are propagated backwards through the graph. The key operation in the stabilize function is propagate. Suppose there are two nodes $r$ and $r'$, and an edge from $r$ to $r'$. Furthermore suppose that $r'$ is split into smaller subnodes.
1. reachable($R_d$) = true; $S = \{R_i\}$
2. while $S \neq \emptyset$ do
3. \hspace{1em} let $r \in S$
4. \hspace{1em} if $r$ is a split node then
5. \hspace{2em} $S = S \cup \{r' \in \text{subnodes}(r) \mid \text{reachable}(r')\} \setminus \{r\}$
6. \hspace{1em} else
7. \hspace{2em} if there exists an enabled, unexplored edge $e$ from $r$ then
8. \hspace{3em} if the destination location $l$ of $e$ was not yet visited then
9. \hspace{4em} create new node $nr$ for $l$; reachable($nr$) = false; $S = S \cup \{nr\}$
10. \hspace{3em} for all outgoing edges $e'$ of the location $l$ do
11. \hspace{4em} split($nr$, guard), where guard denotes the guard of $e'$
12. \hspace{2em} stabilize($r$, $nr$)
13. \hspace{2em} if there exists a region $r \subseteq R_d$ s.t. reachable($r$) then
14. \hspace{2em} return true
15. \hspace{1em} else
16. \hspace{2em} $S = S \setminus \{r\}$
17. \hspace{2em} return false
18. stabilize($r$, $r'$):
19. \hspace{1em} propagate($r$, $r'$)
20. \hspace{2em} for all nodes $rr$ having an edge to $r$ do
21. \hspace{3em} stabilize($rr$, $r$)
22. \hspace{2em} for all $sr \in \text{subnodes}(r)$ do
23. \hspace{3em} if $sr \subseteq R_i$ or there exists a node $rr$ with an edge to $sr$ s.t. reachable($rr$) then
24. \hspace{4em} reachable($sr$) = true

Fig. 1. simplified reachability analysis algorithm

Then propagate splits $r$ in to subnodes that are stable with respect to the edges to subnodes of $r'$. One could say it 'propagates' the splitting of $r'$ back to $r$.

The backward propagation technique is combined with a reachability analysis mechanism, to avoid unnecessary splitting of unreachable nodes. The stabilize operation keeps track of the reachability properties and avoids splitting unreachable nodes.

It is obvious that the algorithm relies on two operations:

- *propagate* ($r$, $r'$): split all subnodes of $r$ into nodes that are stable with respect to edges to $r'$.

- *split* ($r$, $c$): split all subnodes of $r$ into nodes that respect $c$

These are the two operations that our symbolic state space will be expected to provide. This is also true for applications to more complex verification problems. Indeed, for more complex verification problems, the algorithm is very similar. The main difference is that there is a more complex mechanism for on-the-fly evaluating whether or not the property can be decided on the current partition. Thus, by constructing an state space representation that allows efficient implementation of the two operations, we open the way for a range of verification solutions based on partition refinement.
3 The symbolic state space

A partition refinement technique is only useful if partitions can be represented symbolically, i.e. as symbolic state spaces. We will describe how symbolic state spaces resulting from a partition refinement algorithm can be represented efficiently. Traditionally, in symbolic model checking canonical representations have been used. However, the time and data resources needed for maintaining such canonical representations have great impact on the performance of the model checking techniques. Most model checking approaches require several operations on regions, which justifies the burden of using canonical representations, because all these operations are performed relatively efficient on such representations. When applying a partition refinement technique, this is not a natural approach because the required operations on the state space are very limited and specific. We already say that only two basic operations are required, namely split and propagate. In this case, there is no advantage in using canonical structures like DBM’s. As we will show, these operations can be implemented surprisingly simple on state spaces represented by a dedicated, more efficient representation.

Our model checking technique operates on set of states that can be seen as tuples \((l, Z)\), where \(l\) is a location and \(Z\) is a set of valuations for the system’s variables. Essentially, what is needed is a representation for zones. Zones occurring in symbolic algorithms are defined as sets of constraints. Canonical representations like Difference Bound Matrices, keep a canonical model for each zone. We took a different approach, which is inspired by the fact that zones are generated by a partition refinement algorithm. In the partition refinement approach zones are the consequence of repeated splitting. It therefore seems natural to represent zones as leaves of a so-called splitting tree. A splitting tree is a binary tree that records a history of splittings. Internal nodes of the tree are labeled with constraints, which represent the splitting of a zone into two sub-zones, one satisfying the constraint and one dissatisfying the constraint. Leaf nodes define zones that are relevant for the symbolic state space. The set of constraints defining the zone associated with a region is obtained by collecting all constraints while traversing the tree upwards from the node, negating constraints when appropriate. Fig. 2 shows an example splitting tree dividing the value space into six zones.

Fig. 2. example splitting tree
This leads to a forest structure in which for every location, a splitting tree is kept which holds for that location. The set of leaf nodes of the tree associated with a location represents the complete set of zones that are relevant for that location. Thus, leaf nodes correspond to region graph nodes of the region graph, where the location is identified by the particular tree in the forest and the zone is defined by the splitting history. Splitting trees can be used to model zones generated by constraints on any type of data. Here, we will focus on zones over the domain of reals, since these are of primary importance because these are the result of clock variables and real variables in the system. We demand that constraints (on reals) in a splitting tree can only be single linear equations. By doing so it is ensured that the leaf nodes of the splitting trees always represent convex zones.

There are several advantages of using splitting trees to represent a symbolic state space. As will be shown below, it allows efficient implementation of the two basic operations needed by the partition refinement algorithm. Furthermore, the creation of new regions is a computationally simple operation, which is a clear difference from canonical representations, where each region has to be brought into a canonical form.

Splitting trees were introduced in [19] also also applied in [16], but these approaches use splitting trees solely for the purpose of propagation, not for state space representation. Splitting trees have some similarity with the ‘decision diagram’ approaches introduced in real-time model checking, for example Difference Decision Diagrams (DDD’s) [15] and Clock Difference Diagrams (CDD’s)[3]. These approaches extend the Binary Decision Diagram (BDD) techniques for the representation of zones. Difference Decision Diagrams (DDD’s) are quite similar to our splitting trees, the difference being that DDD are normalised to a form that allows fast implementation of the necessary operations required for reachability analysis based model checking. For splitting trees no normalisation is done, because partition refinement requires different operations on regions. The only time-consuming operation in our model checking approach is reduction, which can be implemented relatively efficiently on unstructured sets of constraints, avoiding the toll of expensive normalisation operations. However, it would be interesting to investigate the application of some of the reduction techniques applied in the decision diagram approaches to simplify splitting trees. Furthermore, we use splitting trees as a general representation for constraints on data, while the above mentioned approaches aim at representation of zones generated by difference constraints. In particular, the current splitting tree application can deal with linear constraints and constraints on other data types (integers, enumerated types). Finally, note that our splitting tree approach is specific for partition refinement algorithms. It would be useless for reachability-based approaches.

3.1 Splitting and reduction

Given a node \( n \) and a constraint \( c \), the split operation minimally splits \( n \) into subnodes that respect \( c \). For regions represented by the splitting tree represen-
tation, the splitting of a region is in fact a trivial operation. Given a region represented by a leaf node of some splitting tree, splitting this region means that the node becomes an internal node labeled with the new constraint and that two new child nodes are added. The latter nodes represent the regions that result from the splitting.

However, what is needed is a mechanism to avoid the useless splitting of nodes. If a zone respects a constraint, then splitting it with the constraint is not useful. The mechanism of avoiding such useless splittings, which we refer to as reduction, is applied whenever a region split is considered. If reduction would not be applied, the algorithm would in most cases keep on splitting eternally. Given a zone \( Z \) defined by a set of constraints \( C \) and a new constraint \( c \), the reduction procedure integrates two tests, one checking whether or not \( \bigwedge C \Rightarrow c \), and another one checking \( \bigwedge C \Rightarrow \neg c \). If either condition is true, then no splitting needs to performed. One reason for restricting ourselves to convex nodes lies in the fact that it allows us to efficiently solve the reduction problem.

The reduction algorithm is a critical part of our model checking approach. In our model checking implementations it proved to be the most time consuming operation by far. Furthermore, reduction is the single operation that limits the type of constraints that can be used to define regions. This limitation is directly related to the type of verification problems that can be handled by the model checker, in particular the complexity of the guards and updates occurring in an XTG model. Thus, the reduction algorithm is a central aspect of our model checking approach, since it largely determines the efficiency of a model checker as well as the complexity of the models it can handle. As one would expect, a higher level of complexity tends to come with a lower efficiency. To further explore this relation, we distinguish three categories of constraints:

1. simple constraints: \( x - y \sim c \) and \( s \sim c \) where \( x \) and \( y \) are variables and \( c \) is an integer constant.

2. linear constraints: \( ax + by + \cdots \sim c \), where \( a \), \( b \) and \( c \) are integer constants, \( x \) and \( y \) are variables.

3. non-linear constraints

For the first type of constraints, efficient representations can be found. Efficient model checkers like Uppaal [12] and Kronos [5] rely on the limitation to simple constraints. For linear constraints things are getting considerably more complex, only few model checkers [2] implementing it. Finally, for non-linear constraints, model checking becomes mostly infeasible, although for some limited problems, solutions have been found [7].

We investigated reduction approaches for the first two types of constraints. Reduction on regions defined by simple constraints can be performed by a variation of a shortest path algorithm. A convex zone defined by a set of simple constraints can conceptually be represented by a graph in which the vertices represent variables and edges represent constraints on these variables: the presence of an edge \( x \rightarrow y \) means that \( x - y < 3 \) is one of the constraints defining the zone. An additional node '0' is introduced to represent constraints on single
variables: an edge $x \rightarrow y$ represents the fact that $x < 3$. Given a new constraint $x - y < c$, the reduction procedure checks if there is a path from $x$ to $y$ with an accumulated weight less than $c$ (thus $\land C \Rightarrow x - y < c$), or a path from $y$ to $x$ with an accumulated weight less than $-c$ ($\land C \Rightarrow -(x - y < c)$). This approach is easily extended to also deal with non-strict constraints.

For linear constraints the reduction problem can be formulated as a linear programming problem. Again let $C$ be the set of constraints that define a zone, and $c$ a new constraint, then the reduction operation can be done by checking

1. whether or not $\land (C \cup c)$ is feasible, and
2. whether or not $\land (C \cup \neg c)$ is feasible.

If for both are feasible, the new constraint $c$ represents a true split. Both problems can obviously be solved by standard linear programming algorithms. We built our own dedicated linear programming package, because an LP solver was needed that efficiently solves large amounts of relatively small LP problems. This requirement does not match very well to existing LP solvers, since these tend to focus on efficiently solving huge problems.

### 3.2 Propagation

Thanks to the fact that the splitting history of zones is recorded in splitting trees, we can implement the propagation operation in an efficient manner, based upon ideas presented in [19]. Suppose that there are two regions $r$ and $r'$ and an edge from $r$ to $r'$, and that $r$ is stable with respect to this edge. If $r'$ becomes split into $n$ smaller subnodes $s_1, ..., s_n$, then $r$ is very likely to become unstable, because usually not all states represented by $r$ will have edges to exactly the same subnodes of $r'$. If we would only use some canonical representation to store regions, we would have to compute separately for each subregion $s_i$ of $r'$, the subregion of $r$ that is stable with respect to $s_i$. However, by using the splitting history of $r'$ (recorded in the subtree 'below' $r'$), we can perform this splitting by propagating the splitting history of $r'$ back to $r$, one split at the time. Propagation can now be defined in terms of the following basic operation. Assume two nodes $r$ and $r'$ in a region graph and an edge $r \rightarrow r'$ between them and that $r$ is stable with respect to the edge. Suppose that $r'$ is split by a single constraint $c'$. Then propagating the splitting of $r'$ back to $r$ means that one has
to find the single constraint \( c \), that splits \( r \) into two subnodes that are stable with respect to the edges to the subnodes of \( r' \).

To perform the propagation operation over discrete edges, the update that is associated with the edge has to be taken into account. Fig. 3 illustrates this. It shows a situation in which there is a discrete edge between two regions (in the diagram represented by their zones) originating from a region graph edge which has an update \( x := y + 2 \) (\( x \) and \( y \) both non-clock variables). If the destination region is split by a constraint \( x < 3 \), this causes the source region to become unstable, because some of its states will have transitions to the region represented by the zone satisfying \( x < 3 \), while others will have transitions to the region represented by the zone satisfying \( x <= 3 \). Therefore the source region is split in such a way that the resulting regions again represent states with transitions to the same destination regions. It can be easily seen that this can be achieved by splitting with the constraint \( y < 3 \).

Alternatively, if a node is unstable with respect to a time edge, a different kind of propagation is to be performed. In this case the passing of time has to be taken into account. See the left diagram in fig. 4. Shown there is a zone defined by a single constraint \( c \geq 3 \) that is split by a constraint \( d < 5 \), \( c \) and \( d \) both being clocks. This causes the region satisfying \( c < 3 \) to become unstable, since not all states in this region have a time edge to the states in the region represented by the zone \( c \geq 3 \land d < 5 \). This is fixed by splitting this zone with the constraint \( d - c < 2 \). A special case of this situation exists, which is illustrated by the second diagram of fig. 4. If a node is split by a clock constraint, this may cause instability of one to the two resulting subnodes with respect to a time edge to the other subnode. In that case an additional split is needed, as is shown in the diagram. Fig. 4 also illustrates the effect of using simplified region graphs. It this figure only the edges relevant for simplified region graphs are shown. In case the zones would be part of a non-simplified region graph, we would also have to draw an edge between \( c < 3 \land d - c > 2 \) and \( c > 3 \land d > 5 \).
To stabilize a node $r$ with respect to an edge to a split node $r'$, we first split $r$ such that its subnodes are stable with respect to the edges to the two direct subnodes of $r'$. Suppose that the direct subnodes of $r'$ are $r_1$ and $r_2$. Then $r$ is to be split in subnodes $r_1$ and $r_2$ such that there are stable edges $r_1 \rightarrow r'_1$ and $r_2 \rightarrow r'_2$. If $r'_1$ (or $r'_2$) is also a split node, then the stabilization process continues in the same way in order to stabilize $r_1$ (or $r_2$). In some cases, splitting $r$ will not be needed because $r$ is already stable with respect to $r_1$ and $r_2$. In that case either $r \rightarrow r'_1$ or $r \rightarrow r'_2$ is present. Note that the detection of these cases is the purpose of the earlier discussed reduction operation.

4 Practical results

We have extended the approach described in section 3 towards the parametric verification of fair TCTL property on systems specified in a modeling language called XTG. XTG is an extension of timed automata which allows the expression value passing, urgency. Furthermore, constraints occurring in guards and invariants can be linear equations. Fair TCTL is an extension of TCTL [9] in the spirit of [8]. Parametric verification allows one to specify parameters both in system and parameters, the result of a parametric verification being a set of constraints on the parameters defining the set of parameter valuations that cause the property to be satisfied. Instead of operating on a region graph, the extended algorithm operates on a region product graph. A region product graph is a product structure of a graph representation of the property and a region graph. The decomposition of properties into graph representations is based on fixed point characterisations of TCTL formulae. The product structure approach is partly based approaches followed in [16] and [8].

The type of constraints that may occur in the exploration determines the type of verification problems that can be handled by the model checker. If a system specification contains only simple constraints on clocks then only simple constraints will occur in the exploration. Likewise, if non-simple, linear constraints occur in a system specification, linear constraints will occur in the exploration. Furthermore, if besides clocks also real variables are allowed in specifications, and these variables occur together with clocks in the specification, also non-simple constraints will occur. As a consequence, most parametric real-time verification problems will need a model checker that is based on linear constraints, since parameters will often be compared with clocks. Finally, being able to deal with linear constraints also allows the verification of a limited set of hybrid systems.

We have build two similar model checking tools, one with the reduction routine for simple constraints and one with the reduction routine for linear constraints. The first one (called PMC) is able to efficiently deal with XTG specifications in which only simple constraints are used and in which discrete variables cannot be mixed with clock variables. The second one (called LPMC) is able to deal with systems specified using linear equations in updates and constraints. It allows full parametric verification, and the verification of limited set of hybrid
systems. Besides real variables, both tools allow the definition of integer and enumerated variables. Both tools can generate diagnostic traces.

To evaluate the performance of the tools, we compared them with leading tools in the field. We compared PMC with Uppaal [12], a well-known model checker that allows the same amount of expressiveness as PMC. Table 4 compares PMC with Uppaal on two benchmark examples from the makers of Uppaal. For Uppaal, we did two series of experiments, one with all optimization strategies turned on, except for convex hull approximation (options -aDS), and one with all optimization strategies turned on (options -aADS). The experiments were performed on a 450 MHz Pentium III with 128 Mb internal memory.

The results show that the performance of PMC is roughly comparable to that of Uppaal without approximation. If Uppaal’s approximation is turned on, it performs far more better than our tool. These results clearly show that a partition refinement approach to model checking leads to performances that are no worse than those of reachability-analysis based tools, at least for exact verification approaches. Obviously, the interesting question that arises is if partition refinement approaches are also compatible with approximate verification approaches. HyTech [2] is a tool that is comparable to LPMC with respect to expressiveness of the modeling language. Table 4 compares their performances.

5 Conclusions and further work

We presented a new model checking approach based on partition refinement. This approach includes a novel technique for representing symbolic state spaces called splitting trees. Unlike canonical representations like DBM’s, splitting trees are not generally applicable in real-time symbolic model checking. The splitting tree representation is dedicated to partition refinement approaches, and therefore
The idea of using a splitting tree is independent of the kind of data that is represented. This makes the incorporation of other data types in the input language of our model checker relatively easy. It is not hard to extend our model checker to also deal with other data types. All that is required is the definition of propagation and reduction procedures for the data type.

The basic idea behind our approach is quite simple. In [19] it was already shown that splitting trees are very useful for implementing the propagation operation. Instead of using a standard canonical state space representation to represent the regions, we extended the usage of the splitting trees to also act as the data structure that defines the region. This works because of the fact that in a partition refinement approach, besides propagation, only one other operation is required, which involves the checking of feasibility of regions. This operation can be implemented relatively efficient on regions represented by sets of constraints.

Based on the ideas presented in this article we build two model checkers that have proved to be strong both in terms of performance as well as expressiveness of the input languages. Especially LPMC, allows a very expressive modelling language, since it enables the expression of any linear equation in guards and updates. On the other hand, PMC is relatively efficient, although limited to simpler models. The performance figures show that partition refinement, compared to other approaches, does not lead to less efficient model checking - as long the appropriate state space representation is used.

We still see much room for improvement. As was mentioned before, the reduction algorithm is the central part of our model checking approach. We are currently further investigating approximate reduction strategies through combination of our approach with the approach implemented in Halbwachs NBAC tool [10]. A nice characteristic of approximate reduction strategies is that it shifts the balance from data space resources to time resources. In our experience, time resources are the limiting factor for many verification problems. In particular, we are looking for reduction strategies that are less expensive than solving an LP problem. Finally it could also be interesting to investigate whether approximate

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>time</td>
<td>memory</td>
<td>time</td>
</tr>
<tr>
<td>fischer-2</td>
<td>0.21</td>
<td>1892</td>
<td>0.22</td>
</tr>
<tr>
<td>fischer-3</td>
<td>1.09</td>
<td>2188</td>
<td>1.32</td>
</tr>
<tr>
<td>fischer-4</td>
<td>8.57</td>
<td>12668</td>
<td>25.54</td>
</tr>
<tr>
<td>fischer-5</td>
<td>71.36</td>
<td>99784</td>
<td>-</td>
</tr>
<tr>
<td>fischer-6</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>CSMA/CD-3</td>
<td>0.34</td>
<td>1892</td>
<td>0.77</td>
</tr>
<tr>
<td>CSMA/CD-4</td>
<td>0.66</td>
<td>1892</td>
<td>5.28</td>
</tr>
<tr>
<td>CSMA/CD-5</td>
<td>1.86</td>
<td>3242</td>
<td>60.83</td>
</tr>
<tr>
<td>CSMA/CD-6</td>
<td>5.26</td>
<td>9624</td>
<td>-</td>
</tr>
<tr>
<td>CSMA/CD-7</td>
<td>18.04</td>
<td>25780</td>
<td>-</td>
</tr>
</tbody>
</table>

Fig. 6. comparing the performance of LPMC and HyTech
reduction strategies can also be used to deal with non-linear constraints, an area in which model checking has not yet proved to be very successful.

References


70
Black Box Checking

Doron Peled
The University of Texas at Austin
Department of Electrical and Computer Engineering
Austin, TX 78712, USA

Moshe Y. Vardi* Rice University
Department of Computer Science
Houston, TX 77005, USA

Mihalis Yannakakis
Avaya Laboratories
233 Mount Airy Road
Basking Ridge, NJ 07920, USA

October 23, 2001

Abstract

Two main approaches are used for increasing the quality of systems: in model checking, one checks properties of a known design of a system; in testing, one usually checks whether a given implementation, whose internal structure is often unknown, conforms with an abstract design. We are interested in the combination of these techniques. Namely, we would like to be able to test whether an implementation with unknown structure satisfies some given properties. We propose and formalize this problem of black box checking and suggest several algorithms. Since the input to black box checking is not given initially, as is the case in the classical model of computation, but is learned through experiments, we propose a computational model based on games with incomplete information. We use this model to analyze the complexity of the problem. We also address the more practical question of finding an approach that can detect errors in the implementation before completing an exhaustive search.

1 Introduction

Model checking [5] and testing [16] are two complementary approaches for enhancing the reliability of systems. Model checking usually deals with checking whether the design of a finite state system satisfies some properties (e.g., mutual exclusion or responsiveness). On the other hand, testing is usually applied to the actual system, often without having access to, or knowledge of its internal structure. It checks whether a system (or an implementation) conforms with some design (i.e., informally, has the same behaviors).

Even if access to the internal structure of the tested system is possible, it is not always a good idea to use it when performing tests, as this may lead to a bias in the testing process. Furthermore, the
whole system may be very large (e.g., millions of lines of code), while we are interested only in specific aspects of it. Extracting the part of the code that is relevant from the whole system, especially in the case of large legacy systems, is most probably infeasible (and is itself subject to errors). Suppose one is interested in checking specific properties of some system such as a communication switch or protocol. Model checking would be appropriate for checking properties of a model of the system, but not checking the system itself. On the other hand, testing methods often compare the system with some abstract design.

One motivation for the current work is the case where acceptance tests need to be performed by a user who does not have access to the design, nor to the internal structure of the checked system. Our aim is thus to combine the two approaches, hence checking automatically properties of finite state systems whose structure is unknown. Of course, a completely hidden structure cannot be effectively checked. Thus, the following properties are assumed:

- A bound \( n \) on the number of states of the checked system is known.
- The tester can always reset the system to its (unique) initial state.
- The input alphabet \( \Sigma \) of the checked system is known.
- An experiment consists of repeatedly applying an input from \( \Sigma \) or a reset to the current state. An indication of whether the input was possible (enabled) from the current state is available.
- If an input \( \alpha \) was possible from the current state, the system makes the move. Otherwise, it stays in the current state. No backtracking is available (but the tester can simulate backtracking by resetting and repeating the successful prefix of the experiment).
- The checked system is deterministic in the sense that from each state it can move with any given input to at most one successor state.

We do not assume that the size of the system is known precisely: \( n \) is only an upper bound on the number of states. In particular, we would like to study the effect of the possibility that the bound on the number of states \( n \) may be much bigger than the actual number of states. This is the case when the number of states is only estimated. In practice, the system that is being checked may be large and have multiple functions, while the property may concern a specific aspect of the system. Although the system as a whole may be quite big, large parts of it may be irrelevant to the property, and the system may be equivalent to a much smaller finite state machine as far as checking the property is concerned. In this case, \( n \) should be taken to be an estimate on the logical complexity (the control structure) of the system with respect to the property at hand. Our methods can be used also if no bound \( n \) is available, by running the algorithms to the extent that the available time and space resources allow; the guarantees in this case depend on the time spent.

Following the automata-theoretic approach to model-checking [12, 24], the (negation of the) checked property is directly given as, or translated into, a finite automaton on infinite words, usually a Büchi automaton [3]. Then, both the system and (the complement of) the checked property are represented using automata. An example of a system that is based on such principles is Spin [8, 10], where the specification is given by an automaton called a never claim that recognizes the bad (or disallowed) computations. In order to check whether the system under consideration satisfies
the checked property, we intersect the automaton representing the system with the automaton representing the disallowed computations. Any sequence in the intersection is a counterexample for the checked property, while the absence of any counterexample means that the property is satisfied.

The problem we study here is a variant of the above model checking problem. We are given the automaton that represents the computations not allowed by the checked property. But the internal structure of the checked system is not revealed, and only some experiments, as described above, are allowed on it. Still, we want to check whether the system satisfies the given property. We call this problem **black box checking**. To simplify the discussion, we will not deal here with machines with output. Their treatment and the results are similar to the ones presented here. We will present on-the-fly algorithms that are aimed at quickly detecting errors in a checked system.

The choice of an appropriate computational model is central to the issue of black box checking. Unlike standard decision problems, the input is not given here at the beginning of the computation, but is learned through a sequence of experiments. We propose a computational model based on games with incomplete information, and use this model to analyze the complexity of the problem. There are two notions of time complexity for testing problems: the *testing* time complexity, i.e., the time spent testing the black box, corresponding to the length of the experiment performed, and the *computational* time complexity, i.e., the time spent in computing the test sequence (experiment) and interpreting the result.

Our methods combine techniques from model checking, conformance testing and learning theory. All three areas have been actively pursued for a number of years and there is an extensive body of literature. Model checking has been a vibrant area of research for more than 15 years with the development of the theory and a number of software tools. Most tools check properties of finite state models expressed in some formal notation. One tool that is directed at the checking of software systems without a model is VeriSoft [7]: it is aimed at checking state invariants (assertions) of communicating processes, using partial order reduction methods for space exploration. For a recent book on model checking see [5].

The study of testing black box automata was initiated in Moore's classical paper from 1956 [15], where he defined and studied several problems including the machine identification problem (infer the state transition diagram of an unknown black box automaton). He also posed the fault detection or conformance testing problem (checking that the black box conforms to a specified design automaton). This problem has been studied in the subsequent years by many researchers, obtaining good bounds on the lengths of the tests needed, as well as efficient algorithms that check for conformance for different types of automata (machines with a distinguishing sequence or with reset, or in general without) [4, 9, 25, 26]. In the last 15 years, there has been a lot of work on conformance testing in the protocols community, with a large number of papers, many of them based on the black box automaton testing models and methods. Early surveys of the work in the 50's and 60's can be found in e.g., [11, 23], and surveys of the more recent results and related work on protocol testing can be found in [13, 21]. Finally, there is substantial work in the learning community on the problem of learning finite automata (i.e. machine identification) with the help of a teacher. Efficient algorithms for learning different types of automata in this framework have been developed in [2, 14, 20].
2 Preliminaries

Automata Theoretic Model-Checking

A Büchi automaton is a quintuple $(S, S_0, \Sigma, \delta, F)$, where $S$ is a finite set of states, $S_0 \subseteq S$ are the initial states, $\Sigma$ is the finite alphabet, $\delta \subseteq S \times \Sigma \times S$ is the transition relation, and $F \subseteq S$ are the accepting states. A run over a word $a_1a_2\ldots \in \Sigma^\omega$ is an infinite sequence of states $s_1s_2s_3\ldots$ with $s_1 \in S_0$, such that for each $i > 0$, $(s_i, \alpha_i, s_{i+1}) \in \delta$. A run is accepting if at least one accepting state occurs in it infinitely many times. A word is accepted by a Büchi automaton exactly when there exists a run accepting it. The language $L(A)$ of a Büchi automaton $A$ is the set of words that it accepts. Two automata are equivalent when they accept the same language.

An implementation automaton $B = (S_B, S_0^B, \Sigma, \delta_B, F_B)$ has several restrictions: $S_0^B$ is a singleton $\{l\}$, and $F_B = S_B$ (namely, all the states are accepting). We assume that the number of states $|S_B|$ is bounded by some value $n$.

We can view an implementation machine in our model as a Mealy machine: at each state $v$ and for each input $a$, the machine outputs 0 if the transition is not enabled, and then remains in the same state, and 1 if it is enabled. Furthermore, we assume that the implementation automaton is deterministic, i.e., if $(s, a, t) \in \delta_B$ and $(s, a, t') \in \delta_B$, then $t = t'$.

For a specification automaton $P = (S_P, S_0^P, \Sigma, \delta_P, F_P)$, we will denote the number of states $|S_P|$ by $m$. Let the size of the alphabet $\Sigma$, common to the implementation and the specification, be $p$.

As we mentioned in the introduction, we can easily extend the framework of this paper, and the results to implementation machines with arbitrary output (i.e., Mealy machines), and specification machines that describe the legal input-output behaviors.

The intersection (or product) $B \times P$ is $(S_B \times S_P, S_0^B \times S_0^P, \Sigma, \delta', S_B \times F_P)$, where

$$\delta' = \{((s, s'), \alpha, (t, t')) | (s, \alpha, t) \in \delta_B \wedge (s', \alpha, t') \in \delta_P\}.$$

Thus, the intersection contains (initial) states that are pairs of (initial, respectively) states of the individual automata. Note that this version of the intersection is a simple case of Büchi automata intersection, given that $F_B = S_B$. The transition relation relates such pairs following the two transition relations. The accepting states are pairs whose second component is an accepting state of $P$. We have that $L(B \times P) = L(B) \cap L(P)$.

A reset is an additional symbol of $S_B$, not included in $\Sigma$, allowing a move from any state to the initial state. An experiment is a finite sequence $\alpha_1\alpha_2\ldots \alpha_{k-1} \in (\Sigma \cup \{\text{reset}\})^*$, such that there exists a sequence of states $s_1s_2\ldots s_k$ of $S_B$, with $s_1 \in S_0^B$, and for each $1 \leq j < k$, either

1. $\alpha_j = \text{reset}$ and $s_{j+1} = \epsilon$ (a reset move), or
2. $(s_j, \alpha_j, s_{j+1}) \in \delta_B$ (an automaton move), or
3. there is no $t \in S_B$ such that $(s_j, \alpha_j, t) \in \delta_B$ and $s_{j+1} = s_j$ (a disabled move).
Games of Incomplete Information

The computation model for experiments on black box automata is not the standard one, in which the input is known from the beginning of the computation. Here, part of the input is hidden, and its structure is studied through experiments.

The relevant computational model is related to games of incomplete information [1, 18], where an 3-player plays against a deterministic environment, representing a degenerate version of a V-player. (This should be contrasted with games against an adversarial environment, used, for example, in program synthesis [19].) Each such game consists of a nondeterministic machine with finitely many configurations\(^1\) \(C\), containing the following disjoint subsets: \(C_i\) are the initial configurations, \(W^+\) and \(W^-\) are the positive and negative winning configurations for the 3-player, respectively.

Intuitively, since we want to check properties of systems, \(W^+\) corresponds to finding an error, and \(W^-\) corresponds to concluding that there is no error.

Let \(L_3\) and \(L_\gamma\) be sets of labels for the 3-player and the environment, respectively. Then the sets of moves are \(M_3 \subseteq C \times L_3 \times C\), and \(M_\gamma \subseteq C \leftrightarrow L_\gamma \times C\), respectively. The 3-player can have a choice of moves, thus \(M_3\) is a relation, connecting the current configuration with all possible pairs of move-labels and resulted successor configurations. The moves of the environment \(M_\gamma\) are defined as a function from the current configuration into the unique transition label and successor configuration. No move can originate in a winning configuration. Moreover, any two different moves from the same configuration must have different labels. The two players make moves in alternation, starting with the 3-player, who makes the first move from an initial configuration. A play is a sequence from \((CL_3CL_\gamma)^*C\), where each adjacent triple over \(C(L_3 \cup L_\gamma)C\) conforms with a move of one of the players. A play is winning if it ends with a winning configuration in \(W^+ \cup W^-\). There is no initial configurations starting both a play that ends with a configurations in \(W^+\) and a play that ends with a configurations in \(W^-\).

The incomplete information is stated by the partition of the configurations \(C\) into equivalence classes called information sets. The 3-player cannot distinguish between configurations \(c_1\) and \(c_2\) that are in the same information set, denoted \(c_1 \approx c_2\). Therefore, the move function \(M_3\) must allow moves with the same labels for all the configurations that are in the same equivalence class. Furthermore, if \(c_1 \approx c_2\) then \(c_1 \in W^+\ (c_1 \in W^-\) respectively) if and only if \(c_2 \in W^+\ (c_2 \in W^-\) respectively).

A deterministic strategy for the 3-player is a function \(st_3 : C \times (L_\gamma \cup \{\text{init}\}) \rightarrow M_3\), such that

1. If the 3-player will keep playing \(st_3(c, l)\) when it is his turn from configuration \(c\) and after the environment has played a move labeled with \(l\), the sequence will end with a configuration in \(W^+ \cup W^-\).

2. If \(c \approx c'\), then the labels on \(st_3(c, l)\) and \(st_3(c', l)\) are the same.

The additional value \(\text{init}\) is paired with initial configurations from \(C_i\) (since there is no previous label for this configuration). A path played according to a strategy is an alternating sequence of

---

\(^1\)Since the games that will be described later involve choosing an automaton and performing experiments on it, we choose to distinguish between a configuration of a game and a state of an automaton.
configurations and labels, starting with an initial configuration. A winning path ends with a winning configuration. We define the deterministic time complexity as the length of the longest winning path in a deterministic strategy that ends with a configuration in $W^+ \cup W^-$.  

We also define a nondeterministic strategy $nst_3 : \mathcal{C} \times (L_{\gamma} \cup \text{init}) \rightarrow M_3$ for the $3$-player. Let $c \in C_i$ be an arbitrary configuration such that there exists a play from $c$ that ends with a configuration in $W^+$. Every play starting with $c$ in which the $3$-player keeps playing his turn according to the $nst_3$ strategy will end with a configuration in $W^+$. The second constraint that was imposed on the deterministic strategy does not have a counterpart in the definition of the nondeterministic strategy. The intuition is that in the nondeterministic case, an $3$-player that is playing according to a nondeterministic strategy can make guesses that can distinguish between configurations that are in the same information set. We define the nondeterministic time complexity of $a$ as the length of the longest winning path in a nondeterministic strategy that ends with a configuration in $W^+$.  

**Combination Lock Automata**

The following family of automata [15, 25] plays a major role in proving lower bounds on experiments with black box automata: a combination lock automaton [15] is a finite automaton such that there exists some complete order of the states $s_1, s_2, \ldots, s_n$ with $s_1$ the initial state, and where the state $s_n$ has no enabled transition. For each state $s_i$, $i < n$, there is a transition labeled with some $\beta_i \in \Sigma$ to $s_{i+1}$. For all other letters $\gamma \in \Sigma \setminus \{\beta_i\}$, there is a transition labeled with $\gamma$ from $s_i$ back to the initial state. Such an automaton is said to be the combination automaton for $\beta_1 \beta_2 \ldots \beta_{n-1}$. Figure 1 depicts a combination lock automaton for $n = 5$.  

A sequence leading to a state without a successor (or even to a state where not all the letters are enabled) in a combination lock automaton must have a suffix of length $n - 1$ that is $\beta_1 \beta_2 \ldots \beta_{n-1}$. (This is a necessary, but not a sufficient condition; for example, the automaton in Figure 1 does not reach a deadlock state as a result of the sequence $\beta_1 \beta_2 \beta_1 \beta_2 \beta_3 \beta_4$ when $\beta_1 \neq \beta_3$, since the second $\beta_1$ only causes it to return to its initial state.) Every path that does not contain the consecutive sequence $\beta_1 \beta_2 \ldots \beta_{n-1}$ is allowed (enabled) by the automaton.
3 Black Box Deadlock Detection

In this section we first describe a simpler problem related to black box checking, which will be presented in the next section. Given a deterministic finite state system $B$, with no more than $n$ states, we want to check whether this machine deadlocks, namely reaches a state from which no input is possible.

In this problem, part of the model is unknown and is learned via experiments, which motivates modeling the problem as a game with incomplete information. We will also demonstrate that the deterministic and nondeterministic complexity do not have the same connections as in the standard model of computation that is based on Turing Machines.

For each implementation automaton with $n$ or less states, there exists a single initial configuration. Each configuration in $C$ in a play contains the same automaton as in the initial configuration, the current state of this automaton, as controlled by the moves of the 3-player, and some information about the sequence of moves played so far. The current state of the automaton in an initial configuration is its initial state. The moves $M_{\text{e}}$ of the environment are labeled by success or fail. The label indicates whether the environment was successful or not in executing the transition corresponding to the label of the last move of the 3-player from the state of the implementation automaton. The moves of the 3-player are the possible input symbols, or a reset followed by a symbol.

Projecting the labels of the moves of the 3-player from a play $E$, we obtain an experiment over the implementation automaton in the initial configuration of $E$. If the configurations $c_1$ and $c_2$ are reachable using the prefixes of two plays $\xi_1$ and $\xi_2$ that correspond to the same experiment, then $c_1 \approx c_2$. The winning set $W^+$ contains only configurations that include an automaton that has a deadlock. Similarly, the winning set $W^-$ contains only configurations that include an automaton without a deadlock.

A Nondeterministic Strategy

The 3-player guesses in each move a label, forming a sequence of length smaller than $n$ that brings the state of the selected machine from its initial state to some deadlock state. He then checks that this state has no enabled transitions, by trying out all input labels without success.

Complexity: Nondeterministic time $O(n + p)$. The only information that is needed to be kept in each configuration is a counter from 0 to $n - 1$ and a counter on the number of labels checked from the final state.

A Deterministic Strategy

The 3-player checks systematically the possible sequences, up to length $n - 1$, starting from the initial state. (Of course, there is no need to check sequences that include prefixes that have led to a failure.)

Complexity: Deterministic (testing and computational) time $O(p^n)$. 

77
Theorem 1 The deterministic testing time complexity for black box deadlock detection is $\Omega(p^{n-1})$.

Proof. Suppose that the initial configuration includes an implementation automaton $B$ with $n$ states that allows any input from any state. Consider a play $\xi$, played using a deterministic strategy for the 3-player. Assume that $\xi$ has less than $p^{n-1}$ moves of the 3-player, and terminates with a winning configuration $c_w$ in $W^-$. Then at least one sequence $\beta_1 \beta_2 \ldots \beta_{n-1}$ does not appear consecutively in the experiment associated with $\xi$. If instead of the above automaton $B$, the environment would have chosen a combination lock automaton for $\beta_1 \beta_2 \ldots \beta_{n-1}$, the deterministic strategy would have resulted in a prefix of a play that has the same labels as $\xi$. Now we would have reached a configuration $c'$ such that $c_w \approx c'$. Further, $c_w$ is associated with an automaton without deadlocks, while $c'$ is associated with an automaton with deadlocks. This contradicts the assumption that $c_w \in W^-$. 

In the standard complexity model, it is not known whether one can obtain a polynomial deterministic algorithm from a nondeterministic polynomial algorithm. Here, the (tight) lower deterministic bound is exponentially larger than the nondeterministic complexity. This justifies the use of games with incomplete information as an alternative for the common computational model of Turing machines.

4 Checking Properties of Black Box Finite State Machines

We address now the problem of black box checking. Namely,

\textit{given a specification B"{u}chi automaton $P$ with $m$ states, and a black box implementation automaton $B$ with no more than $n$ states, over a mutual alphabet $\Sigma$ with $p$ letters, check if there is a sequence accepted by both $P$ and $B$.}

Recall that the automaton $P$ accepts the bad computations, i.e., those that are not allowed. Thus, if the property is given originally e.g., using a linear temporal logic (LTL) [17] property $\varphi$, then $P$ is the automaton corresponding to $\neg \varphi$. For an efficient translation from LTL to automata, see e.g., [6]. The following simple theorem demonstrates that the current problem is at least exponential in time in the size of the automaton $B$.

Theorem 2 The deterministic testing time complexity of black box checking is $\Omega(p^{n-1})$.

Proof. Similar to the proof of Theorem 1, we construct variants of a combination lock automata. The deadlock state is replaced now with a self loop labeled by $\gamma$. The symbol $\gamma$ is disabled in the initial state. This removes at most half of the possible combinations (in the case where $p = 2$), so the complexity changes only by a constant factor. The property automaton $P$ consists of two states: $t_0$ which is an initial state, and $t_1$ which is an accepting state. There is a self loop from $t_0$ to itself on each label from $\Sigma$, and from $t_1$ to itself on $\gamma$. There is also an edge labeled by $\gamma$ from $t_0$ to $t_1$. Thus, the intersection is nonempty exactly if a state can be reached in the black box automaton, where $\gamma^w$ can be executed. The only such state of a combination lock black box is the state at the end of the path prescribed by the combination.
4.1 An Off-Line Strategy

A straightforward way to perform black box checking is to infer first the structure of the black box system, and then to apply model checking techniques to its newly revealed structure. The machine identification problem is a well studied problem. Typically, it is applied to automata that produce output, either at the states (Moore machines) or at the transitions (Mealy machines). As we mentioned, an implementation machine in our model can be viewed as a Mealy machine, where output 0 on a transition means that it is not enabled and output 1 means that it is enabled.²

It is well known that if two machines with \( n \) states are not equivalent, then there is an input of length at most \( 2n - 1 \) that distinguishes them. This implies that any machine with at most \( n \) states is completely characterized by its output on all input strings of length \( 2n - 1 \). That is, a black box is uniquely determined by applying all such \( p^{2n-1} \) input strings. A \( p \)-ary tree of depth \( 2n - 1 \) can be constructed from the responses of the black box, and it can be minimized to produce the minimal machine \( M \) consistent with these outputs [23]. Then we can use model checking to check whether \( M \) satisfies the given property \( P \). The length of the test sequence (or in terms of games with incomplete information, the length of the corresponding play), which gives us the testing time complexity, is \( O(np^{2n-1}) \). If implemented in the straightforward way, the space complexity is also exponential (to record the tree of all the input strings and their output). However, the construction and minimization can be done instead together incrementally within polynomial space. The computational time for the model checking is comparatively small, \( O(pmn) \) where \( m \) is the size of the property automaton \( P \) (which is typically very small).

The complexity of this method is not that far off the lower bound, and in the worst case one may indeed need to identify in effect the black box machine in order to check a property. However, intuitively it is clear that in many cases this method can be wasteful in that it does not take advantage of the property to avoid doing a complete identification. For example, suppose that the property is that some error indication label \( \gamma \) never occurs. The property automaton \( P \), representing the bad computations is in this case a simple 2 state automaton. Obviously in this example, there is no reason to wait until we reconstruct the full black box automaton before we check the property. The sensible thing to do would be to check the assertion (i.e., try to see if \( \gamma \) is enabled in the current state) as we go along during the test, and if it gets violated at some point, then an error has been found and the check is complete.

In general, it would be obviously beneficial to use the property automaton on the fly to detect errors as early as possible and prune the test. Notably, if the estimate \( n \) on the number of states is much higher than the actual number of different states, or if it is accurate, but there is still a 'small' counterexample, i.e. there is a small set of states that exhibits the faulty behavior, we would like to be able to find the error without searching the whole space, if possible. This is not always as easy, especially in the case of properties that depend on the infinite behavior of the system, that is, in cases where the property automaton is a genuine Büchi automaton. We will investigate such methods in the following sections.

²By this convention, the output provides some partial information also on the next state, namely if the output is 0 then we know that the state does not change. This is not important for what follows (i.e. all the methods apply to any Mealy machine) but it can be used to do obvious optimizations on the tests. For example, if we apply input \( a \) and it is not enabled, then it is pointless to try again \( a \) until an enabled transition has been performed.
4.2 An On-the-fly Strategy

We will present now a strategy that is guided by the checked property and can terminate before exploring the entire structure of the black box automaton.

A Nondeterministic Strategy

As before, we start with a nondeterministic version, in order to demonstrate the principle behind the on-the-fly black box checking.

According to the strategy, the 3-player guesses a path $\sigma$ of the automaton $P$, starting from an initial state, that can be partitioned into two subpaths $\sigma_1$ and $\sigma_2$, each of which is of length smaller or equal to $mn$. Both subpaths end with the same accepting state $t$ of $P$. Furthermore, the black box automaton is tested to allow executing the transitions of $\sigma_1, \sigma_2^{n+1}$ after a reset.

The first path $\sigma_1$, when input to the automaton $P$, needs to terminate with some accepting state $t$. The second path $\sigma_2$, when starting in state $t$ of $P$, needs to terminate with $t$ as well. For such a pair, we apply the second path $n$ more times. That is, we try to execute the path $\sigma_1, (\sigma_2)^n$. If we succeed, this means that there is a cycle in the intersection through a state with a $t$ (which is accepting) as the $P$ component, since there are at most $n$ ways to pair up $t$ with a state of $B$. In this case, there is an infinite accepting path in the intersection.

Correctness: Consider the unknown end state of the intersection automaton for each iteration of $\sigma_2$ in the experiment $\sigma_1, \sigma_2^{n+1}$. Then at least one component state $s$ of $B$ must occur twice with the same accepting component state $t$ of $P$ (as there are no more than $n$ states in the intersection that have the same component $t$). Thus, the path $\sigma$ must include a cycle through an accepting state, which guarantees that an infinite accepting run exists in the intersection. Conversely, it is easy to see that if the intersection of $B$ and $P$ is nonempty, such a guess exists.

Complexity: Nondeterministic time $O(n^2m)$.

A Deterministic Strategy

As in the nondeterministic case, the strategy finds a path consisting of the components $\sigma_1$ and $\sigma_2^{n+1}$, where $|\sigma_1|$ and $|\sigma_2|$ is smaller or equal to $mn$. This is done by running through the different possible paths in a systematic way.

Complexity: Deterministic time $O(n^2p^{2mn}m)$. This is because there are $p^{2mn}$ choices of such paths. Each is of length bounded by $mn$, and we repeat it $n+1$ times.

The following comments should be noted:

- The complexity of this strategy grows exponentially with the number of states $m$ of $P$. However, $m$ is typically small, or even fixed, when talking about a fixed property.

- For properties that can be specified by automata on finite strings (i.e., depend essentially on finite computations, e.g., safety properties), we need to search only for the first string $\sigma_1$ and the complexity is $O(np^{mn}m)$. 

10
• When searching for the strings $a_1, a_2$, we need only consider strings that can be extended to accepting strings of the property automaton. Furthermore, we can start by limiting the length of the subpaths that we explore and gradually increase that length as we proceed in the search. In this way, if the actual size of the automaton $B$ is much smaller than $n$, and an error occurs, it can be found much earlier than in the exhaustive strategy, as required above.

4.3 A Strategy Based on Learning and Testing

We show now that the factor $m$ in the exponent can actually be removed. We provide below a strategy with complexity whose exponential term is $O(p^n)$. Furthermore, if the black box has an error, the time complexity will be exponential only in the actual size of the minimized version of the black box automaton.

The Conformance Testing Procedure

In our strategy, we will use as a procedure an algorithm for conformance testing of a given finite automaton $M$ with a black box automaton $B$ by Vasilevskii and Chow [4, 25]. The (testing and computational) time complexity of this algorithm is $O(k^2 n p^{n-k+1})$, where $n$ is the assumed bound on the size of the black box automaton, $k \leq n$ is the size of the given automaton $M$, and $p$ is the alphabet size. Intuitively, the algorithm has to check the states and transition relation of the black box automaton, and that no error that follows a 'combination lock' occurs from any one of its nodes.

We give a brief description of the Vasilevskii-Chow algorithm. Assume without loss of generality that the given automaton $M$ is minimized and has $k$ states. Construct (1) a spanning tree $T$ of the automaton $M$ rooted at its initial state, and (2) a 'characterizing' set $W$ of input strings that pairwise distinguish the states of $M$. That is, a set of strings with the property that for any two distinct states $s$, $t$ of $M$, there is a string $w \in W$ which produces different output starting from states $s$ and $t$ (recall that outputs 0 and 1 correspond to disabled and enabled transitions). There is always a set $W$ with at most $k$ strings of length at most $k$, which can be constructed efficiently by the classical automata minimization algorithms. The Vasilevskii-Chow algorithm applies to the black box automaton $B$ all the strings of the form $0.\beta$, where $\beta$ ranges over the input strings corresponding to the paths of $T$ starting at the root, $\beta$ ranges over all input strings of length $n-k+1$, and $\gamma$ ranges over all strings of $W$. If $B$ produces the same outputs as $M$ on all these test strings, and $B$ has at most $n$ states, then $B$ is equivalent to $M$. There are $k$ choices for string $\alpha$, at most $2p^{n-k+1}$ choices for $\beta$ and at most $k$ choices for $\gamma$. The length of each test string is $O(n)$, so the total length of the test is $O(k^2 n p^{n-k+1})$.

In general, the different test strings can be applied in arbitrary order as far as correctness is concerned. However, for efficiency purposes it is advantageous to apply them in increasing order of the substrings $\beta$, i.e. first the empty string for $\beta$ combined with all $\alpha$ and $\gamma$, then all strings $\beta$ of length 1 (i.e. all input symbols), then length 2 etc. In this manner, if the actual size of the black box $B$ is $l \leq n$, and $B$ does not conform with $M$, then a discrepancy will be discovered for some $\beta$ of length at most $l-k+1$, i.e. with a test of length at most $O(k^2 l p^{l-k+1})$. (If $l < k$, then a discrepancy will be found with the length of $\beta$ equal to 0, i.e., for some string $\alpha \gamma$, thus with test complexity $O(k^3)$.)
The Automata Learning Algorithm

Another procedure that we use in our strategy is an algorithm for learning an automaton $B$ with reset using membership tests and questions to an oracle (a teacher) by Angluin [2]. In this learning algorithm, the teacher answers equivalence tests to a proposed machine and provides a counterexample in case of inequivalence. We will replace the teacher with experiments on the black box automaton. Starting from a trivial automaton, Angluin’s algorithm generates successively larger candidate automata $M_i$, for $i = 1, 2, \ldots$ (the number of states in each conjectured automaton is monotonically increasing). It asks the teacher for equivalence. If equivalence does not hold, it uses a counterexample provided by the teacher, queries some more strings, and then generates the next conjectured automaton with more states, until it reaches the correct number of states. At this point the conjectured automaton is the correct one.

The basic data structure of Angluin’s algorithm consists of two finite sets of finite strings $V$ and $W$ over the automaton alphabet $E$, and a table $I$. The set $V$ is prefix closed (and contains thus in particular the empty string $\epsilon$). The rows of the table $I$ are the strings in $V \cup V \epsilon$, while the columns are the strings in $W$. Let $I(v, w) = 1$ when the sequence of transitions $vw$ is enabled in $B$ (starting from $B$’s initial state), and 0 otherwise. The entry $I(v, w)$ can be computed by performing the experiment $vw$ on the automaton $B$.

We define an equivalence relation $\equiv \mod(W)$ over strings in $E^*$ as follows: $v_1 \equiv v_2 \mod(W)$ if and only if for each $w \in W$, the sequence of transitions $v_1 w$ is enabled in $B$ if and only if the sequence $v_2 w$ is enabled in $B$. Thus, two strings $v_1$ and $v_2$ corresponding to two rows of the table $I$ are equivalent $\mod(W)$ if the rows are the same. It is easy to check that this is indeed an equivalence relation.

Denote by $[v]$ the equivalence class that includes $v$. A table $I$ is closed if for each $v, a \in E \setminus \{\epsilon\}$, either $I(v, a) = 0$, or there is some $v' \in V$ such that $va \equiv v' \mod(W)$. A table is consistent if for each $v_1, v_2 \in V$ such that $v_1 \equiv v_2 \mod(W)$, either $I(v_1, \epsilon) = I(v_2, \epsilon) = 0$, or for each $a \in \Sigma$, we have that $v_1 a \equiv v_2 a \mod(W)$. Notice that if the table is not consistent, then there are $v_1, v_2 \in V$, $a \in \Sigma$ and $w \in W$, such that $v_1 \equiv v_2 \mod(W)$, and $v_1 aw \in \mathcal{L}(B)$ but $v_2 aw \notin \mathcal{L}(B)$ or vice-versa. This means that $f(v_1 a, w) \neq f(v_2 a, w)$, and thus we can add $aw$ to $W$ in order to separate $v_1$ from $v_2$.

Given a closed and consistent table $I$ over the sets $V$ and $W$, we construct a proposed automaton $M_f = (S, \{s_0\}, \Sigma, \delta, F)$ as follows:

- The set of states $S$ is $\{[v] : v \in V, f(v, \epsilon) \neq 0\}$.
- The initial state $s_0$ is $[\epsilon]$ (where $\epsilon$ is the empty string).
- The transition relation $\delta$ is defined as follows: for $v \in V, a \in \Sigma$, the transition from $[v]$ on input $a$ is enabled iff $f(v, a) = 1$ and in this case $\delta([v], a) = [va]$.
- $F = S$.

3By this definition, the entries of the table are 0-1. Alternatively we could have defined $f(v, w)$ to be the output sequence produced by $B$ during the application of the string $w$ starting from the state reached after $v$, i.e. a bit string indicating whether each transition of $w$ is enabled at the corresponding state. This may result in a smaller set $W$, though the entries of the table will be longer.
The facts that the table $f$ is closed and consistent guarantee that the transition relation is well defined. In particular, the transition relation is independent of which state $v$ of the equivalence class $[v]$ we choose; if $v, v'$ are two equivalent states in $V$, then for all $a \in \Sigma$ we have that $[va]$ coincides with $[v'a]$ (by consistency) and is equal to $[u]$ for some $u \in V$ (by closure).

There are two basic steps used in the learning algorithms for extending the table $f$:

- $\text{add.rows}(v)$: Add $v$ to $V$. Update the table by adding a row $va$ for each $a \in \Sigma$ (if not already present), and by setting $f(va, w)$ for each $w \in W$ according to the result of the experiment $vaw$.

- $\text{add.column}(w)$: Add $w$ to $W$. Update the table $f$ by adding the column $w$, i.e., set $f(v, w)$ for each $v \in V \cup V.\Sigma$, according to the experiment $vw$.

We define a subroutine $\text{ANGLUIN}$ that accepts a data structure as above consisting of two sets of strings and a table (corresponding to a previous guess of the automaton $B$), and a counter example $\sigma$, on which $B$ and the current guess do not agree. Without loss of generality we can assume that the only disagreement occurs at the last symbol of $\sigma$ (otherwise, we can truncate $\sigma$). The algorithm updates the sets $V$, $W$ and the table to a new consistent and closed table, and returns the corresponding automaton, denoted $\text{automaton}(V, W, f)$. The subroutine adds to the table rows corresponding to all the prefixes of the counter example $\sigma$, plus all the prefixes extended by a single letter from $\Sigma$. Then the subroutine adds rows and columns to the table $f$ until it is both closed and consistent. The subroutine appears in Figure 2.

The Black Box Checking Algorithm

For the purpose of black box checking, we modify Angluin's algorithm as follows. Our modification can use two kinds of counterexamples, provided by the teacher:

1. A simple counterexample of the form $\sigma \in \Sigma^*$, meaning that $\sigma$ belongs to one of the checked automata, but not the other.

2. A pair of words $\sigma_1, \sigma_2 \in \Sigma^*$ such that $\sigma_1 \sigma_2^*$ belongs to one of the checked automata, but not the other.

In either case, we truncate the counterexample string to the first disagreement between the two automata (the black box and the conjectured automaton).

We construct a sequence of automata $M_1, M_2, \ldots$ that attempt to converge to the black box automaton $B$. Membership queries are just experiments on the black box $B$. For equivalence queries, suppose we have a conjectured automaton $M_i$ for the black box. First, we check if $M_i$ generates a word accepted also by the specification automaton $P$, namely whether $\mathcal{L}(M_i) \cap \mathcal{L}(P) \neq \emptyset$. If the intersection is not empty, it must contain an ultimately periodic word of the form $\sigma_1 \sigma_2^*$ [22]. We input $\text{reset} \sigma_1 \sigma_2^{n+1}$ to the black box $B$. If this experiment succeeds, then there is an error as $\mathcal{L}(B) \cap \mathcal{L}(P)$ contains $\sigma_1 \sigma_2^*$ and thus is not empty. If it fails, then this gives a counterexample to the equivalence of $B$ with $M_i$.  

13

83
subroutine ANGLUIN((V,W,f), σ)
    if f is empty then
        let V := {ε}; W = {ε};
        add_rows(ε);
    else
        for each prefix v′ of σ that is not in V do
            add_rows(v′);
        while (V,W,f) is inconsistent or not closed do
            if (V,W,f) is inconsistent then
                find v₁,v₂ ∈ V, a ∈ Σ, w ∈ W, such that
                v₁ ≡ v₂ mod(W) and f(v₁a,w) ≠ f(v₂a,w);
                add_column(aw);
            else /* (V,W,f) is not closed */
                find v ∈ V, a ∈ Σ,
                such that va ∉ [u] for any u ∈ V;
                add_rows(va);
            end while
        return automaton(V,W,f)
    end ANGLUIN

Figure 2: The Angluin Algorithm

The special case of 'safety' properties, i.e. properties that depend only on finite computations, is a bit simpler. In this case P is an automaton on finite words, and we can find a witness finite string σ ∈ L(Mᵢ) ∩ L(P); we input reset σ to the black box as above to check if B allows it, which signifies an error, or does not allow it, which provides a counterexample to the equivalence of B and Mᵢ. If the experiment fails, we use the resulted counterexample σ or σ₁σ₂⁺¹ (truncated to the first discrepancy) in Angluin's algorithm to generate the next candidate automaton with more states.

If Mᵢ does not generate any word accepted by P, we check whether Mᵢ conforms with B. Let k be the number of states of Mᵢ. We start the conformance test between Mᵢ and B assuming B has k states and apply the Vasilevskii-Chow (VC) algorithm. If the conformance test fails, we use the counterexample in Angluin's learning algorithm to generate Mᵢ₊₁. If the conformance test succeeds, we repeat it with k + 1, k + 2, ... , n. Note that, by our earlier description of the VC algorithm, we do not have to start in each round the algorithm from scratch, but simply extend it, by testing the strings of the form αβγ for increasing lengths of the middle substring β. If the bound n is reached without a discrepancy, we declare that the black box satisfies the checked property.

This strategy is described in Figure 3. The procedure TEST takes a black box automaton B along with a state bound n, and a property given in terms of an automaton P that specifies the erroneous computations. It determines whether B satisfies the property, provided that it has at most n states. The procedure call VC(M,k) activates the Vasilevskii-Chow algorithm for conformance testing the current conjectured automaton M with the black box automaton B, assuming that B has no more than k states. VC returns (true,−) if the conformance test succeeds. If it fails, it returns
proc TEST(B:black box; P:prop. automaton; n:state bound)
    V := W := Ø; f := empty table;
    M := ANGLUIN(empty, -);
    while (M has at most n states) do
        X := M × P;
        if \( L(X) = \emptyset \) then
            k := number of states of M;
            repeat
                (conforms, o) := VC(M, k);
                k := k + 1;
                until k > n or ¬conforms;
                if conforms then WIN(+);
            else
                let \( \sigma_1, \sigma_2 \) be strings such that \( \sigma_1 \sigma_2^\varepsilon \in L(X) \);
                if B allows reset \( \sigma_1 \sigma_2^\varepsilon \) then WIN(-);
                else \( \sigma := \) maximum prefix of \( \sigma_1 \sigma_2^\varepsilon \) allowed by B;
                M := ANGLUIN((V, W, f), \sigma);
            end while
        end if
        return 'the black box has more than n states';
    end while
end TEST

Figure 3: A strategy using learning and testing
columns of the table) to distinguish the states. When we call the VC procedure in TEST, one
choice is to use the set $V$ of strings to access the states (one string of $V$ for each equivalence
class) instead of constructing a tree $T$, and to use the current set $W$ of strings constructed from the calls
to Angluin’s procedure (corresponding to the columns of the table $f$). The advantage of doing this
is that by construction of the conjectured automaton $M$, we know that $VC(M,k)$ will succeed for $k$
equal to the number of states of $M$ (all the relevant experiments were run in the construction of the
current table $f$), and hence we can skip that call and continue from there. A possible disadvantage
is that the strings in $V$ and $W$ may be longer than necessary.

The opposite choice is to replace the table $f$ with a spanning tree $T$ from the initial state (for
example a shortest path spanning tree), and a new characterizing set $W$, with the objective of
making it have as few and short strings as we can find. Furthermore, if this is not the first call to
VC, then we can simply extend the tree $T$ from the previous call to reach the new states and extend
the characterizing set similarly. If the conformance test then succeeds for $k$ (the number of states
of $M$), and fails at a higher bound, we then have the option in the next Angluin call to ignore the
old data structure and use instead the paths of $T$ as the new set $V$ and the characterizing set as
the column set $W$; the entries of the table $f$ are already known from the test executed during the
$VC(M,k)$ call. This may lead to shorter subsequent test sequences in the Angluin procedure.

Another variant of the algorithm is the following: Suppose that at some iteration the model
checking determines that the conjectured automaton $M$ does not satisfy the property and that it
produces an example string that is not accepted by the black box $B$. If the string $\sigma$ is too long, instead
of feeding it to the $ANGLUIN$ procedure, we may first try to find a much shorter counterexample
to the equivalence of $B$ and the conjectured automaton $M$ by calling the VC procedure up to some
bound on the number of states of $B$. If the VC procedure succeeds in finding a counterexample, and
the first such counterexample that it finds is $a(3''\gamma$, where $a$ is a string corresponding to a path of the
spanning tree $T$, $\beta$ is a string of length $r \leq d$, and $\gamma \in W$, then it can be shown that all strings of
the form $a\beta$$'$, where $\beta'$ is a prefix of $\beta$, lead to states that are inequivalent to each other and to all
the other states that were previously found (i.e. the states of $B$ reached by the paths of the tree $T$.)
Thus, calling the $ANGLUIN$ algorithm with the counterexample $a\beta$ will result in at least $r$ more
states.

Complexity of the Strategy

We discuss now the testing complexity, i.e., the length of the test. The computational complexity
is similar. Suppose that the minimum equivalent automaton of the black box $B$ has $l \leq n$ states.
Suppose first that the black box $B$ has an error. Then this error will be discovered by the time
the conjectured automaton $M$ reaches size $l$, that is, at some iteration we will have a conjectured
automaton $M$ with at most $l$ states (possibly the minimum automaton equivalent to $B$, but also
possibly an inequivalent one with fewer states), and the model checking algorithm will produce a
string in the intersection of $M$ and $P$, which is allowed by $B$. Clearly the VC procedure is never
called with a state bound greater than $l$. If the calls to VC are performed incrementally as explained
earlier, i.e. by extending the same tree $T$ and characterizing set $W$, reusing the previously performed
test sequences, then the total time spent by all the VC calls is $O(l^2p')$.

We analyze next the cost of the $ANGLUIN$ calls. For simplicity and for the purposes of worst
case complexity, we assume that the VC and ANGLUIN algorithms use a common set of access and distinguishing strings. Further, we employ the variant mentioned above, in which if \( B \) does not allow a string produced by the model checking, then we do not call ANGLUIN directly with this counterexample string \( \sigma \), but instead run the VC algorithm to find another short counterexample. In practice, we would do this only if the counterexample from model checking is too long (in the worst case it can have length \( l^2m \)). As mentioned above, if the VC algorithm finds a counterexample string of the form \( \alpha\beta \), where \( \alpha \) is a path of the tree \( T \) and \( \beta \) has length \( r \), then all the \( r \) strings \( \alpha\beta' \) for the prefixes \( \beta' \) of \( \beta \) lead to new inequivalent states, and we can extend the spanning tree \( T \) from the state reached by the path \( \alpha \) by hanging a path of new states labeled by \( \beta \). When we call the ANGLUIN procedure with the counterexample string \( \alpha\beta \), we would first include in \( V \) the string \( \alpha\beta \) and its prefixes; note that \( \alpha \) and its prefixes are already present (because \( \alpha \) is a path of the access tree \( T \)). The other strings \( \alpha\beta' \) will give new inequivalent states. The same is true of any other strings that are added to \( V \) by ANGLUIN as a consequence of the table not being closed. Note that each of the latter strings has the form \( \nu \sigma \) where \( \nu \in V \) and \( \sigma \in \Sigma \); we extend the tree \( T \) as we add \( \nu \sigma \) to \( V \). From these remarks, it follows that the set \( V \) has at all times at most \( l \) elements, all of length at most \( l \), and thus the ANGLUIN calls perform at most \( l^2p \) tests, each of length \( O(1) \), for total cost \( O(l^3p) \). Clearly, this is dominated by the bound of the VC calls.

Besides the VC and the ANGLUIN calls, the remaining tests on the black box apply the strings from the model checking to determine whether \( B \) can execute them. If the property is characterized by an automaton on finite strings with \( m \) states, then the counterexamples provided by the model checking algorithm have length at most \( ml \). There are at most \( l \) such strings, for total length \( l^2m \). Typically, \( m \) (the size of the property automaton) is a small constant, \( m << l \), and therefore \( l^2m \) is dominated by \( l^2p \) even for \( p = 2 \). But in general, even if this is not the case (i.e. if \( m \) is large), for the purpose of bounding the worst case complexity, we can do the following trick to amortize all, except the last test from the model checking: to test if \( B \) can execute a string \( \sigma \) (provided by the model checking), apply successively prefixes of \( \sigma \), doubling the length each time, interleaved with tests from the VC algorithm that use approximately equal time. Then the time for these tests is dominated by the VC calls, except for the last test when the conjectured automaton may be the correct automaton. The cost of the last test is \( O(lm) \). Thus, for properties that depend on finite executions, the total complexity of TEST in case of a faulty black box is \( O(l^3p + lm) \). Note that this does not depend on the a priori bound \( n \).

If the property is characterized by an automaton on infinite strings, then the model checking will give strings \( \sigma_1 \) and \( \sigma_2 \) of length at most \( lm \). If \( B \) does not allow \( \sigma_1\sigma_2 \), then the first failure will occur within the first \( l \) repetitions of \( \sigma_2 \), because the black box must be at inequivalent states after \( \sigma_1 \) and after each repetition of \( \sigma_2 \). Thus, all these tests have length \( l^2m \), except for the last one which \( B \) can execute completely (showing that \( B \) is faulty), which has length \( lmn \). Again, regarding the tests before the last one, their cost is dominated by the VC calls - either because \( m \) typically is a small constant, or otherwise apply the trick described above. Consequently, in this case the total testing complexity of the algorithm is \( O(l^3p + lmn) \). In this case the complexity depends on the bound \( n \), but only linearly.

If the black box satisfies the property, then the strategy will generate the minimum equivalent automaton \( M \) in time \( O(l^3p') \), but then it will have to spend \( O(l^2np^{n-l+1}) \) more time to verify that the black box is indeed equivalent to \( M \) (by conformance checking \( M \) with \( B \)). It should be noted that unless the bound \( n \) is a good estimate of the actual size \( l \) (more precisely, if \( n > 2l \), the
complexity is dominated by this last check that confirms the conformance of B with the conjectured automaton.

**Theorem 3** Black box checking, for a black box automaton B with l states, where l is unknown but is smaller than some bound n, and a property automaton P with m states can be done in testing time

- \(O(l^3 p^l + l m n)\), when there is an error, i.e., the intersection of B and P is nonempty (time \(O(l^3 p^l + l m)\) if P is an automaton on finite strings), and
- \(O(l^3 p^l + l^2 n p^{n-1} + 1)\), when there is no error.

If we do not have a bound n on the number of states of the automaton B, we can run the algorithm as long as time permits. Consider first a property characterized by an automaton on finite strings (such as deadlock freedom and other safety properties). If we ever encounter an error, i.e. find a string \(\sigma\) accepted by both the black box and the property automaton P, then it is a true error and we can stop the test. If there is an error and B has size \(l\) (or the smallest counterexample has length \(l\)), we are sure to find the error within time \(O(l^3 p^l + l m)\). Conversely, if after the allocated time no error has been found, then this means that either the black box is correct, or else the smallest possible counterexample and the size of the black box must exceed a certain bound, which depends on the time spent on the test.

Suppose that we have a Büchi automaton that depends on infinite behaviors. Suppose further that at some point the conjectured black box automaton \(M_i\) has a nonempty intersection with the property automaton \(P\), and let \(\sigma_1 \sigma_2^*\) be a string in the intersection. If the conjectured automaton \(M\) has \(l\) states at this point and \(P\) has \(m\) states, then the strings \(\sigma_1, \sigma_2\) have length at most \(l m\). We can input \(\text{reset}\sigma_1\) to the black box followed by repeated applications of \(\sigma_2\) until either the black box does not accept it or we run out of time. As described before, we can also interleave the application of successively longer prefixes of \(\sigma_1 \sigma_2^*\) with running the VC algorithm for approximately equal amounts of time, until either we find a counterexample or run out of time. In the first case, we have found at least one new state and we continue the algorithm as before. In the second case, that is, if we run out of time after executing \(r\) repetitions of \(\sigma_2\), then we can conclude that either there is an error, or the size of the implementation machine exceeds \(r\).

Finally, if the conjectured automaton has an empty intersection with the property automaton then we perform conformance testing for increasingly larger values of the bound \(n\) on the black box. At the end we can place again a lower bound on the size of the black box or conclude that it is otherwise correct.

Let us comment finally on the exponential lower bound derived from the combination lock automata. Obviously these are rather pathological, worst case examples. The 'average' automata are much better behaved and do not exhibit this nasty performance bottleneck. This can be formalized by considering a probabilistic model for machines with output. Formally, there has been extensive work studying the properties of random machines [23]. The usual model of a random Mealy machine on \(l\) states is defined as follows. For each state and input symbol choose the next state and output uniformly at random. For the average machine, polynomial time will suffice to find an error. In the following statement, 'almost all machines' means that the probability tends to 1 as the size goes to infinity.
Theorem 4 For almost all black box Mealy machines with \( l \) states, if an error is present, it will be found after a test of length \( O(lp\log^2 l + lm) \) (respectively, length \( O(lp\log^2 l + lm) \) for properties described by automata on finite strings).

This can be shown using the following two nice properties of almost all random machines: (1) if a state \( q \) can reach another state \( q' \) then it can reach it in \( O(\log_p l) \) steps; (2) any two states can be distinguished by input strings of length \( \log_p \log_p l \) [23]. Of course, if there is no error and we want to make sure that we do not have any other automaton at hand with at most \( n \) states, then we still would need to do the conformance testing (at a cost exponential in the difference \( n - l \)) in order to be certain of the correctness.

5 Conclusions

We defined the problem of black box checking, showed lower bounds and provided three strategies for solving the problem. The lower bound in Theorem 2 implies that in the worst case the complexity of black box checking is exponential in the estimated size of the unknown automaton. For comparison, checking the emptiness of the intersection of the same automata (now both structures are given) is in NLOGSPACE-complete. In conformance testing, one checks whether a given known automaton \( P \) of length \( l \) is equivalent to a black box automaton \( B \) of length bounded by some \( n \geq l \). Vasilevskii and Chow [25, 4] showed a lower and upper bound of \( O(l^2 n^{n+1}) \) for conformance testing with reliable resets. When \( n = l \), namely the actual size of the black box automaton is known, this is a much more tractable complexity than that of black box checking. Thus, if a model (abstract design) is available or feasible to construct, then a good strategy for the developer of a system is to separately do a conformance test of an abstract design against the system, and then model check the design with respect to various properties. However, when a model is not available or \( n \) can be considerably bigger than \( l \) then in the worst case we cannot avoid the exponential complexity.

It is quite clear that the off-line strategy is suboptimal as it does not take advantage of the property at hand. On the other hand, the on-the-fly strategies, while still exponential, may work in practice in some important cases. One case is when an error exists and the estimate \( n \) is much higher than the actual size of the checked system or the size of a portion of the system that provides a counterexample. Another case is when the specification automaton \( P \) limits the possible bad executions considerably. An example for a "helpful" specification will be that \( P \) specifies sequences of the form \( a^* (\beta + \gamma) \). An example of an "unhelpful specification" is \( X^* \alpha \), where \( X \) allows any letter of \( \Sigma \) except \( \alpha \).

The last strategy, based on conformance testing and learning, uses the property \( P \) while trying to learn the structure of \( B \). Thus, an error may be found before completing the construction of a minimized automaton equivalent to \( B \). It is also possible that no explicit bound is given on the size of the black box automaton. In this case, we can use the strategy as long as we are willing to spend time. Finally, as we pointed out, the exponential lower bound occurs because of certain pathological examples like the combination lock automaton, and in fact for most machines ("almost all" in a probabilistic sense), if there is an error, the strategy will find it almost surely in polynomial time.

There is a number of issues in black box checking that deserve further investigation. Some open problems are finding strategies for partially specified automata, or known automata where the actual
implementation deviates from the known design in no more than $k$ changes ('implementation errors').

Another problem is to develop an algorithm for black box checking when reliable reset moves are not available. It is possible that similar techniques can be used by combining the learning algorithm of [20] with the conformance testing algorithm of [26] for machines without a reset capability.

We have run several experiments of black box checking. An implementation was programmed by Alex Groce from CMU. We used it to verify communication protocols with up to two thousand states. We are currently experimenting with additional heuristics for increasing the number of states that we can handle.

References


Computer Science Reports

Department of Mathematics and Computer Science
Technische Universiteit Eindhoven

If you want to receive reports, send an email to: m.m.j.l.philips@tue.nl (we cannot guarantee the availability of the requested reports)

*In this series appeared:*

97/02 J. Hooman and O. v. Rooymalen

97/03 J. Blanco and A. v. Deursen
Basic Conditional Process Algebra, p. 20.

97/04 J.C.M. Baeten and J.A. Bergstra

97/05 J.C.M. Baeten and J.J. Vereijken

97/06 M. Franssen

97/07 J.C.M. Baeten and J.A. Bergstra
Bounded Stacks, Bags and Queues, p. 15.

97/08 P. Hoogendijk and R.C. Backhouse
When do datatypes commute? p. 35.

Communication Modeling: The Language/Action Perspective, p. 147.

97/10 P.C.N. v. Gorp, E.J. Luit, D.K. Hammer
Distributed real-time systems: a survey of applications and a general design model, p. 31.

97/11 A. Engels, S. Mauw and M.A. Reniers

97/12 D. Haucholdt, E. Verbeek and W. van der Aalst

97/13 W.M.P. van der Aalst
Exploring the Process Dimension of Workflow Management, p. 56.

97/14 J.F. Groote, F. Monin and J. Springintveld
A computer checked algebraic verification of a distributed summation algorithm, p. 28

97/15 M. Franssen
AP-: A Pure Type System for First Order Logic with Automated Theorem Proving, p. 35.

97/16 W.M.P. van der Aalst
On the verification of Inter-organizational workflows, p. 23

97/17 M. Vaccari and R.C. Backhouse
Calculating a Round-Robin Scheduler, p. 23.

97/18 Werkgezamen~hip Informatiewetenschap
redactie: P.M.E. De Bra
Informawetenschap 1997
Wetschappelijke bijdragen aan de Vijfde Interdisciplinaire Conferentie
Informatiewetenschap, p. 60.

98/01 W. Van der Aalst

98/02 M. Voorhoeve
State / Event Net Equivalence, p. 25

98/03 J.C.M. Baeten and J.A. Bergstra
Deadlock Behaviour in Split and ST Bisimulation Semantics, p. 15.

98/04 R.C. Backhouse
Pair Algebras and Galois Connections, p. 14

98/05 D. Dams
Flat Fragments of CTL and CTL*: Separating the Expressive and Distinguishing Powers. P. 22.

98/06 G. v.d. Bergen, A. Kaldewaij
Maintenance of the Union of Intervals on a Line Revisited, p. 10.

V.J. Diekissen

98/07 Proceedings of the workshop on Workflow Management:
Net-based Concepts, Models, Techniques and Tools (WFMS98)
June 22, 1998 Lisbon, Portugal
edited by W. v.d. Aalst, p. 209

98/08 Informal proceedings of the Workshop on User Interfaces for Theorem Provers:
edited by R.C. Backhouse, p. 180

T. Basten and J. Hooman Process Algebra in PVS

J. Zwanenburg The Proof-assistant Yarrow, p. 15


V. Bos and J.J.T. Kleijn Structured Operational Semantics of Ξ, p. 27

H.M.W. Verbeek, T. Basten and W.M.P. van der Aalst Diagnosing Workflow Processes using Woflan, p. 44

R.C. Backhouse and P. Hoogendijk Final Dialgebras: From Categories to Allegories, p. 26

S. Andova Process Algebra with Interleaving Probabilistic Parallel Composition, p. 81


T. Basten and W. v.d. Aalst Inheritance of Workflows: An Approach to tackling problems related to change, p. 66


T.A.C. Willemse The Analysis of a Conveyor Belt System, a case study in Hybrid Systems and timed \( \mu \)CRL, p. 44


K.M. van Hee, R.A. van der Toorn, J. van der Woude and P.A.C. Verkoulen A Framework for Component Based Software Architectures, p. 19

A. Engels and S. Mauw Why men (and octopuses) cannot juggle a four ball cascade, p. 10


T. Basten, W.M.P. v.d. Aalst Inheritance of Behavior, p. 83

J.C.M. Baeten and T. Basten Partial-Order Process Algebra (and its Relation to Petri Nets), p. 79


I.C.M. Baeten and I.A. Bergstra Mode Transfer in process Algebra, p. 14

J.C.M. Baeten Process Algebra with Explicit Termination, p. 17.

S. Mauw and M.A. Reniers A process algebra for interworkings, p. 63.

R. Bloo, J. Hooman and E. de Jong Semantical Aspects of an Architecture for Distributed Embedded Systems*, p. 47.

The Parallel Composition of Uniform Processes with Data, p. 19

Variable Binding Operators in Transition System Specifications, p. 27.

Grammars Compared: A study on determining a suitable grammar for parsing and generating natural language sentences in order to facilitate the translation of natural language and MSC use cases, p. 33.

A Formal Development of Distributed Summation, p. 35


Enhancing Partial-Order Reduction via Process Clustering, p. 14

Message Sequence Charts in the Software Engineering Process, p. 26

Termination in Timed Process Algebra, p. 36

Impossible Futures and Determinism, p. 14

An Interactive Viewer for Mathematical Content based on Type Theory, p. 24.

Characterizing λ-terms with equal reduction behavior, p. 12

Belief Revision with Explicit Justifications: an Exploration in Type Theory, p. 30.

Parameters in Pure Type Systems, p. 41.

Specifying Internet applications with DiCons, p. 9


A Rewriting Approach to Binary Decision Diagrams, p. 27

Interpretations of Automata, p. 41

Systems for Open Terms: An Overview, p. 39

Beyond Zeno-behaviour, p. 15

Design of the mine pump control system, p. 69

Axiomatizing GSOS with termination, p. 22

Proceedings: Third Workshop on Adaptive Hypertext and Hypermedia Sonthofen, Germany, July 14, 2001, Aarhus, Denmark, August 15, 2001

Operational semantics of DiCons, a formal language for developing Internet applications.

Operational Conservativity with Binding Terms.
