A CHECKER FOR MODAL FORMULAS FOR PROCESSES WITH DATA

Jan Friso Groote and Tim A.C. Willemse
Department of Mathematics and Computer Science,
Technische Universiteit Eindhoven, P.O. Box 513,
5600 MB Eindhoven, The Netherlands
J.F.Groote@tue.nl T.A.C.Willemse@tue.nl

Abstract

We propose an algorithm for the automatic verification of first-order modal μ-calculus formulae on infinite state, data-dependent processes. The use of *boolean equation systems* for solving the model-checking problem in the finite case is well-studied. In this paper, we extend on this solution, such that we can deal with infinite state, data-dependent processes. We provide a transformation from the model checking problem to first order boolean equation systems. Moreover, we present an algorithm to solve these equation systems and discuss the capabilities of the algorithm, implemented in a prototype. We also present the application of our prototype tool to several well-known infinite state processes from the literature. This prototype has also been successfully applied in proving properties of systems that we could not deal with using other available tools.

keywords: Model Checking, Verification, Process Algebra, μCRL, First Order Modal μ-Calculus, First Order Boolean Equation Systems, Infinite Data-Types, Infinite State Systems

1 Introduction

Model checking has come about as one of the major advances in automated verification of systems in the last decade. It has earned its medals in many application areas (e.g. communications protocols, timed systems and hybrid systems), originating from both academic and industrial environments.

However, the class of systems to which model checking techniques are applicable, is restricted to systems in which dependencies on infinite data-types are absent, or can be abstracted from. The models for such systems therefore do not always represent these systems best. In particular, for some systems the most vital properties express requirements on data. There, the model checking technique breaks down. This clearly calls for an extension of model checking techniques for systems that are data dependent.

In this paper, we explore a possibility for extending model checking techniques to deal with processes which can depend on data. We describe an algorithm, for which we have also implemented a prototype, that verifies a given property on a given data-dependent process. The problem in general is easily shown to be undecidable, so, while we can guarantee soundness of our algorithm, we cannot guarantee termination of the algorithm. However, it turns out that many interesting systems with infinite state spaces can be verified using our algorithm, as several examples suggest, including systems with extremely large but finite state spaces, for which all other available techniques failed to provide answers.

The framework we use for describing the behaviour of a system is process algebraic. We use the process algebraic language μCRL [14, 16], which is an extension of ACP [3]; this language includes a formal treatment of data, as well as an operational and axiomatic semantics of process terms. Compared to CCS or ACP, the language μCRL is more expressive. This is due to the presence of data. For our model checking algorithm, we assume that the processes are written in a special format, the *Linear Process Equation* format, which is discussed in e.g. [27]. Note that this does not pose a restriction on the set of processes that can be modelled using μCRL, as all sensible process descriptions can be transformed to this format [15]. When dealing with data-types, an explicit representation of the entire state space is often not possible, since it can very well be infinite. Using the LPE format has the advantage of working with a finite representation of the (possibly infinite) state space.

The language we use to denote our properties in is an extension of the modal μ-calculus [19]. In particular, we allow first order logic predicates and parameterised fixpoint variables in our properties. These extensions, which are also described in e.g. [13], are needed to express properties about data.

The approach we follow is very much inspired by the work of Mader [21], and uses (in our case, first order) boolean equation systems as an intermediate formalism. We present a translation of first order modal μ-calculus expressions to first order boolean equation systems in the presence of a fixed Linear Process Equation. The algorithm for solving the first order boolean equation systems is based on the Gauß elimination algorithm described in, e.g. [21].

This paper is structured as follows: Section 2 briefly introduces the language μCRL and the Linear Process Equations format that is used in all subsequent sections. In Section 3, we describe the first order modal μ-calculus. Section 4 discusses first order boolean equation systems and describes how we can translate first order modal μ-calculus formulas, given a Linear Process Equation, to a sequence of first order boolean equations. An algorithm for solving the first order boolean equations is then described in Section 5; its implementation is discussed in Section 6, along with two small examples, demonstrating the capabilities of a prototype implementation. Section 7 is reserved for concluding remarks.

Related Work. We distinguish automatic verification from semi-automatic and manual verification. As an example of the latter two types of verification, theorem proving systems are often employed. The advantages of theorem proving lie in the ability of the user to express virtually any system and prove any property. Of course, this comes at the cost of reduced automation, and is considered intractable for large systems.

In the fully automatic verification of systems, we can distinguish between two different approaches. On the one hand, dedicated techniques are being developed to deal with specialised classes of systems. Such classes contain communication protocols (e.g. regular expressions [1], queue representations [5]) process networks (e.g. Presburger arithmetic [9]) and parameterised systems (e.g. counter abstraction [24]). This clearly contrasts to our work, as we do not restrict the class of systems we consider. The other approach is to deal with a restricted class of properties that can be verified. A promising techniques in this direction is the *Counter arithmetic with Lambda expressions and Uninterpreted functions* (CLU) by Bryant et al. [8]. CLU is very general in that it can be used to model both data and control, and in [8], it is shown to be decidable. The tool, based on CLU, UCLID, is however restricted to dealing with safety properties only. Our approach is more general, as it allows safety, liveness and fairness properties to be verified automatically. Moreover, CLU is restricted to the quantifier-free fragment of first order logic, whereas the logic we use in our approach employs the full first order logic.

2 Preliminaries

Our main focus in this paper is on processes with data. As a framework, we use the process algebra μCRL [14]. Its basic constructs are along the lines of ACP [3] and CCS [23], though its syntax is influenced mainly by ACP. In the process algebra μCRL, data is an integral part of the language, which makes the language more expressive than CCS or ACP (see discussion in [20]). As we enforce no restrictions on data or on data-types, we here introduce the more abstract notion of data by considering only a *data algebra*.

Definition 2.1 (*Data Algebra*).
A *Data Algebra* is a tuple $\mathcal{A} = (\mathcal{D}, \mathcal{F})$, where $\mathcal{D}$ is a collection of sets called *data domains*. The set $\mathcal{F}$ contains functions from data domains to some single data domain.

For the exhibition of the remainder of the theory, we assume we work in the context of a data algebra without explicitly mentioning its constituent components. As a convention, we assume the data algebra contains all the required data types; in particular, we always have the domain $\mathbb{B}$ of booleans with functions $\mathsf{t}\colon \to \mathbb{B}$ and $\mathsf{f}\colon \to \mathbb{B}$, representing *true* and *false* at our disposal.

The language μCRL has only a small number of carefully chosen operators and primitives. Processes are the main objects in the language. A set of parameterised actions *Act* is assumed; actions can be considered as functions from a data domain to a process. An action $a \in \mathit{Act}$ represents an atomic event, taking a number of data arguments. The process representing no behaviour, i.e. the process that cannot perform any actions is denoted δ. This constant is often referred to as *deadlock* or *inaction*. Note that all actions a terminate successfully immediately after executing the action, whereas the process $a \cdot \delta$ does not terminate successfully.

Processes are constructed using several operators. The main operators are alternative composition ($p+q$ for some processes p and q) and sequential composition ($p \cdot q$ for some processes p and q). The sequential composition operator is often not written down explicitly . Conditional behaviour is denoted using a ternary operator (we write $p \lhd b \rhd q$ when we mean process p if b holds and else process q). The process b::$\rightarrow p$ serves as a shorthand for the process $p \lhd b \rhd \delta$, which represents the process p under the premise that b holds. Recursive behaviour is specified using equations. Data is intertwined with processes such that process variables can be considered as functions from a data domain to processes. Consider the following process.

$$X(n{:}\mathbb{N}) = \mathit{up} \cdot X(n+1) + \mathit{show}(n) \cdot X(n) + [n > 0]{::}{\rightarrow} \mathit{down} \cdot X(n-1)$$

The behaviour denoted by process $X(n)$ is the increasing and the decreasing of an internal counter n or showing its current value. Note that the *up* and *down* actions do not have parameters. For the formal exposition, however, it can be more convenient to assume that actions and processes have a single parameter. This assumption is easily justified, as we can assume the existence of a singleton data domain, together with adequate pairing and projection functions.

A more complex notion of process composition consists of the parallel composition of processes (we write $p \| q$ to denote the process p parallel to the process q). Synchronisation is achieved using a separate communication function γ, prescribing the result of a communication of two actions (e.g. $\gamma(a, b) = c$ denotes the communication between actions a and b, resulting in action c). Two parameterised actions $a(n)$ and $b(n')$ can communicate to action $c(n'')$ only if the communication between actions a and b results in action c (i.e. $\gamma(a, b) = c$) and $n'' = n' = n$.

The communication function is used to specify *when* communication is possible; this, however, does not mean communication is enforced. To this end, we must encapsulate the individual occurrences of the actions that participate in the communication. This is done using the encapsulation operator (we write $\partial_H(p)$ to specify that all actions in the set of actions H are to be encapsulated in process p).

The last operator considered here is data-dependent alternative quantification (we write $\sum_{d:D} p$ to denote the alternatives of process p, dependent on some arbitrary datum d selected from the (possibly infinite) data domain D). The $\sum$-operator is best compared to e.g. input prefixing, but is more expressive (see e.g. [20]). As an example of the $\sum$-operator we consider a process that can set an internal counter to an arbitrary value, which can be read at will:

$$V(n{:}\mathbb{N}) = \mathit{read}(n) \cdot V(n) + \sum_{n':\mathbb{N}} \mathit{set}(n') \cdot V(n')$$

For verification or analysis purposes, it is often most convenient to eliminate parallelism in favour of sequential composition and (quantified) alternative composition. A behaviour of a process can then be denoted as a state-vector of typed variables, accompanied by a set of condition-action-effect rules. Processes denoted in this fashion are called *Linear Process Equations*.

Definition 2.2 (*Linear Process Equations*).
A *Linear Process Equation* (LPE) is a parameterised equation taking the form

$$X(d{:}D) = \sum_{i:I} \sum_{e_i:D_i} [c_i(d, e_i)] ::\rightarrow a_i(f_i(d, e_i)) \cdot X(g_i(d, e_i))$$

where I is a finite index set; D and D_i are data domains; d and e_i are data variables; $a_i \in \mathit{Act}$ are actions with parameters of sort D_{a_i}; $f_i{:}D \times D_i \rightarrow D_{a_i}$, $g_i{:}D \times D_i \rightarrow D$ and $c_i{:}D \times D_i \rightarrow \mathbb{B}$ are functions. The function f_i yields, on the basis of the current state d and the bound variable e_i, the parameter for an action a_i; the "next-state" is encoded in the function g_i, and is determined on the basis of the current state and the bound variable e_i. The function c_i describes when action a_i can be executed.

In this paper, we restrict ourselves to the use of non-terminating processes, i.e. we do not consider processes that, apart from executing an infinite number of actions, also have the possibility to perform a finite number of actions and then terminate successfully. Including termination into our theory does not pose any theoretical challenges, but is omitted in our exposition for brevity. Several techniques and tools exist to translate a guarded μCRL process to linear form (see e.g. [15, 27]). In the remainder of this paper, we use the LPE-notation as a vehicle for our exposition of the theory and practice.

The operational semantics for μCRL can be found in e.g. [14, 16]. Since we restrict our discussions to process expressions in LPE-form, we here only provide a definition of the labelled transition system as it is induced by a process in LPE-form.

Definition 2.3 (*Transition System of an LPE*).
The *labelled transition system* of a Linear Process Equation as defined in Def. 2.2 is a quadruple $M = \langle S, \Sigma, \longrightarrow, s_0 \rangle$, where

- $S = \{X(d) \mid d{\in}D\}$ is the (possibly infinite) set of *states*;
- $\Sigma = \{a_i(d_{a_i}) \mid i{\in}I \wedge a_i{\in}\mathit{Act} \wedge d_{a_i}{\in}D_{a_i}\}$ is the (possibly infinite) set of labels;
- $\rightarrow = \{(X(d), a_i(d'_a), X(d')) \mid i{\in}I \wedge a_i{\in}\mathit{Act} \wedge \exists_{e_i \in D_i} c_i(d, e_i) \wedge d'_a = f_i(d, e_i) \wedge d' = g_i(d, e_i)\}$ is the *transition relation*. We write $X(d) \xrightarrow{a(e)} X(d')$ rather than $(X(d), a(e), X(d')){\in} \rightarrow$;
- $s_0 = X(d_0){\in}S$, for a given $d_0{\in}D$, is the *initial state*.

3 First Order Modal μ-Calculus

The logic we consider is based upon the modal μ-calculus [19], extended with data variables, quantifiers and parameterisation (see [13]). This logic allows us to express data dependent properties. We refer to this logic as the *first order modal μ-calculus*. Its syntax and semantics are defined below.

Definition 3.1 (*Action Formulae*).
An *Action Formula* is defined by the following grammar

$$\alpha \ ::= \ a(e) \mid \top \mid \bot \mid \neg\alpha_1 \mid \alpha_1 \wedge \alpha_2 \mid \alpha_1 \vee \alpha_2 \mid \exists d{:}D.\alpha_1 \mid \forall d{:}D.\alpha_1$$

Here, a is a parameterised action of set *Act* and e is some data expression of the datatype D.

The action formulae are interpreted over a labelled transition system M, which is induced by an LPE (see Def. 2.3). In this paper, we use *environments* for registering the (current) values of variables. Hence, an environment is a (partial) mapping of a set of variables to elements of a given type. The action formulae are interpreted in the context of a data environment ε. We use the following notational convention: we write $\varepsilon[v/d]$ for the data environment ε', defined as $\varepsilon'(d') = \varepsilon(d')$ for all $d' \not\equiv d$ and $\varepsilon'(d) = v$. In effect, $\varepsilon[v/d]$ stands for the data environment ε where the value of d has changed to v. The interpretation of a datum d in a data environment ε is written as $d\varepsilon$.

Definition 3.2 (*Interpretation of Action Formulae*).
Let ε be a data environment and α be an action formula. The interpretation of α in the context of data environment ε is denoted $[\![\alpha]\!]\varepsilon$, and is defined inductively as:

$$\begin{array}{ll}
[\![\top]\!]\varepsilon & = \Sigma \\
[\![\bot]\!]\varepsilon & = \emptyset \\
[\![a(e)]\!]\varepsilon & = \{a(e\varepsilon)\} \\
[\![\neg\alpha]\!]\varepsilon & = \Sigma \setminus [\![\alpha]\!]\varepsilon \\
[\![\alpha_1 \wedge \alpha_2]\!]\varepsilon & = [\![\alpha_1]\!]\varepsilon \cap [\![\alpha_2]\!]\varepsilon \\
[\![\alpha_1 \vee \alpha_2]\!]\varepsilon & = [\![\alpha_1]\!]\varepsilon \cup [\![\alpha_2]\!]\varepsilon \\
[\![\exists d{:}D.\alpha]\!]\varepsilon & = \bigcup v{\in}D \ [\![\alpha]\!]\varepsilon[v/d] \\
[\![\forall d{:}D.\alpha]\!]\varepsilon & = \bigcap v{\in}D \ [\![\alpha]\!]\varepsilon[v/d]
\end{array}$$

Hence, we can use $\top$ to denote an arbitrary (parameterised) action. This is useful for expressing e.g. progress conditions. We subsequently define the set of *state formulae*. We present these in *Positive Normal Form*. This means that negation only occurs on the level of atomic propositions and, in addition, all bound variables are distinct.

Definition 3.3 (*State Formulae*).
A *State Formula* is given by the following grammar.

$$\begin{aligned}\varphi \; ::= \; & b \mid Y(e) \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid [\alpha]\varphi_1 \mid \langle\alpha\rangle\varphi_1 \mid \\ & \exists d{:}D.\varphi \mid \forall d{:}D.\varphi \mid (\mu Z(d{:}D).\varphi)(e) \mid (\nu Z(d{:}D).\varphi)(e)\end{aligned}$$

where b is an expression of the domain $\mathbb{B}$, e is some data expression, α is an action formula and Y is a propositional variable. We assume all names in $(\sigma X(d{:}D).\varphi)(e)$, where $\sigma{\in}\{\mu,\nu\}$ is a fixpoint operator, are unique, i.e. each variable is bound only once by a fixpoint operator.

State formulae are interpreted over a labelled transition system M, induced by an LPE, according to Def. 2.3. The value of a propositional variable X in the context of a propositional environment ρ is written as $X\rho$ and is a function of type $D \to 2^S$. We use the same notational conventions for propositional environments as we used for data environments. We interpret state formulae in the context of a data environment ε and a propositional environment ρ.

Definition 3.4 (*Interpretation of State Formulae*).
Let ε be a data environment, ρ a propositional environment and let φ be a state formula. The interpretation of φ in the context of data environment ε and propositional environment ρ is denoted $[\![\varphi]\!]\rho\varepsilon$, and is defined inductively as:

$$\begin{array}{ll}
[\![b]\!]\rho\varepsilon & = \begin{cases} S \text{ if } [\![b]\!]\varepsilon \\ \emptyset \text{ otherwise}\end{cases} \\
[\![X(e)]\!]\rho\varepsilon & = (X\rho)(e\varepsilon) \\
[\![\varphi_1 \wedge \varphi_2]\!]\rho\varepsilon & = [\![\varphi_1]\!]\rho\varepsilon \cap [\![\varphi_2]\!]\rho\varepsilon \\
[\![\varphi_1 \vee \varphi_2]\!]\rho\varepsilon & = [\![\varphi_1]\!]\rho\varepsilon \cup [\![\varphi_2]\!]\rho\varepsilon \\
[\![[\alpha]\varphi]\!]\rho\varepsilon & = \{X(v){\in}S \mid \;\; \forall v'{\in}D \; \forall a{\in}\mathit{Act} \; \forall v_a{\in}D_a \\
 & \qquad\qquad (X(v) \xrightarrow{a(v_a)} X(v') \wedge a(v_a){\in}[\![\alpha]\!]\varepsilon) \to X(v'){\in}[\![\varphi]\!]\rho\varepsilon\} \\
[\![\langle\alpha\rangle\varphi]\!]\rho\varepsilon & = \{X(v){\in}S \mid \;\; \exists v'{\in}D \; \exists a{\in}\mathit{Act} \; \exists v_a{\in}D_a \\
 & \qquad\qquad (X(v) \xrightarrow{a(v_a)} X(v') \wedge a(v_a){\in}[\![\alpha]\!]\varepsilon \wedge X(v'){\in}[\![\varphi]\!]\rho\varepsilon)\} \\
[\![\forall d{:}D.\varphi]\!]\rho\varepsilon & = \bigcap_{v'\in D} [\![\varphi]\!]\rho(\varepsilon[v'/d]) \\
[\![\exists d{:}D.\varphi]\!]\rho\varepsilon & = \bigcup_{v'\in D} [\![\varphi]\!]\rho(\varepsilon[v'/d]) \\
[\![(\mu Z(d{:}D).\varphi)(e)]\!]\rho\varepsilon & = (\bigcap\{X{:}D \to 2^S \mid [\![\varphi]\!](\rho[X/Z])\varepsilon \dot{\subseteq} X\})(e\varepsilon) \\
[\![(\nu Z(d{:}D).\varphi)(e)]\!]\rho\varepsilon & = (\bigcup\{X{:}D \to 2^S \mid X \dot{\subseteq} [\![\varphi]\!](\rho[X/Z])\varepsilon\})(e\varepsilon)
\end{array}$$

Note: for $X{:}2^{D\to 2^S}$ and $d{\in}D$, we write $X(d)$ for the set of elements $\{x(d)|x{\in}X\}$.

Here, we define the ordering $\dot{\subseteq}$ on the set $D \to 2^S$ as $X\dot{\subseteq}Y$ iff for all $d{:}D$ we have $X(d) \subseteq Y(d)$. The set $(D \to 2^S, \dot{\subseteq})$ forms a complete lattice. From Lemma 3.5, stated below, the existence and uniqueness of fixpoints in state formulae immediately follows.

Lemma 3.5. The operator $\Psi{:}(D \to 2^S) \to (D \to 2^S)$, associated to state formula $(\sigma Z(d{:}D).\psi)(e)$, and defined as $\Psi = \lambda X{:}D \to 2^S.\lambda v{:}D.[\![\psi]\!](\rho[X/Z])(\varepsilon[v/d])$ for data environment ε and propositional environment ρ is monotonic over the complete lattice $(D \to 2^S, \dot{\subseteq})$.

Proof. Follows from the fact that the state formulae are presented in Positive Normal Form. □

The modal μ-calculus is quite expressive, but also renowned for its incomprehensibility. An enlightening explanation of the modal μ-calculus can be found in e.g. [6]. We here provide two sample expressions and give an explanation of their meaning.

Example 3.6. An example of a (first order) modal μ-calculus formula is one that identifies processes for which progress is ensured. The expression $\nu X.([\top]X \wedge \langle\top\rangle\top)$ expresses that we can "infinitely often" perform at least a single step. Thus, this expression must be interpreted as freedom of deadlock.

Example 3.7. Assume a process with at least the states s_0, s_1 and s_2, the labels $a(\top)$ and $a(\bot)$ and the state formula φ (see Fig. 1). We write $s \models \varphi$ to denote that φ is satisfied in state s, and, likewise, we write $s \not\models \varphi$ to denote that φ is not satisfied in state s. We illustrate the difference in data-quantification in action formulae and data-quantification in state formulae. Then, we can formulate two properties, showing the distinction between data-quantification in action formulae and in state formulae:

1. The state formula $(\exists b{:}\mathbb{B}.\ [a(b)]\varphi)$ holds in state s_0. Basically, this expression states there exists a data-parameter b, such that after executing an action $a(b)$, we end up in a state satisfying φ.
2. The state formula $([\exists b{:}\mathbb{B}.a(b)]\varphi)$ does not hold in state s_0. This expression states that, whatever the value of the parameter of the action a is, we end up in a state satisfying φ, which, obviously, is not true.

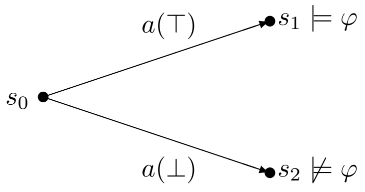

Figure 1: Example of a simple transition system.

Note that data-quantification in action formulae can be used for abstracting from the actual values for parameterised actions.

Note also that we have several identities between action formulae and state formulae (see Lemma 3.8). Using these identities, we can rephrase the second state formula of the last example to the equivalent state formula $\forall b{:}\mathbb{B}.[a(b)]\varphi$, which makes the difference between both state formulae in that particular example more obvious.

Lemma 3.8. Let φ be a state formula, such that $d \notin FV(\varphi)$, and let α be an action formula. Then, we have the following identities:

- $\langle\exists d{:}D.\alpha\rangle\varphi \Leftrightarrow \exists d{:}D.\langle\alpha\rangle\varphi$,
- $[\exists d{:}D.\alpha]\varphi \Leftrightarrow \forall d{:}D.[\alpha]\varphi$,
- $\exists d{:}D.[\alpha]\varphi \Rightarrow [\forall d{:}D.\alpha]\varphi$,
- $\langle\forall d{:}D.\alpha\rangle\varphi \Rightarrow \forall d{:}D.\langle\alpha\rangle\varphi$

Note: here we use implication as an abbreviation for $\subseteq$ and bi-implication as an abbreviation for $=$ on the interpretations of the state formulae.

Proof. Follows immediately from the interpretations of action formulae and state formulae. $\square$

Notice that the converse of the latter two identities is in general not true. For this, we consider the following example.

Example 3.9. Assume again a process with at least the states s_0, s_1 and s_2, the labels $a(\top)$ and $a(\bot)$ and the state formula φ. Consider the part of this process visualised by Fig. 2. We show that the converse of the latter two identities in Lemma 3.8 does not hold.

1. The state formula $\forall b{:}\mathbb{B}.\langle a(b)\rangle\varphi$ obviously holds in state s_0: since the universal quantifier ranges over a finite domain, we can write this formula as $\langle a(\top)\rangle\varphi \wedge \langle a(\bot)\rangle\varphi$. However, the state formula $\langle\forall b{:}\mathbb{B}.a(b)\rangle\varphi$ does not hold in state s_0: we can write this formula as $\langle\bot\rangle\varphi$, which actually holds in no state.

2. Similarly, we can prove that state formula $[\forall b{:}\mathbb{B}.a(b)]\neg\varphi$ holds in state s_0. However, state formula $\exists b{:}\mathbb{B}.[a(b)]\neg\varphi$ does not hold in state s_0, since both transition $a(\top)$ and $a(\bot)$ lead to a state where φ holds, contradicting the requirement that φ should not hold.

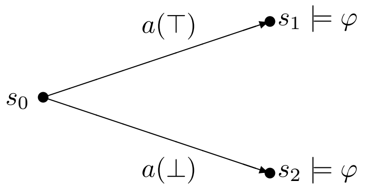

Figure 2: Example of another transition system.

This last example shows that the quantifiers inside action formulae cannot in general be removed in favour of the quantifiers in state formulae. Thus, compared to the fragment of the first order modal μ-calculus that disallows quantifiers inside action formulae, the quantifiers inside action formulae indeed increase the expressivity of the whole first order modal μ-calculus.

4 Equation Systems

We aim at verifying first order modal μ-calculus expressions on processes, specified as a Linear Process Equations. For this, we follow the approach, outlined in e.g. [21]. In essence, we use an extension of the formalism of *boolean equation systems* as an intermediate formalism that allows us to combine a Linear Process Equation with a first order modal μ-calculus expression.

Definition 4.1 (*First Order Boolean Expression*).
A *first order boolean expression* is a formula φ in positive form, defined by the following grammar:

$$\varphi ::= b \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid X(e) \mid \forall d{:}D.\varphi \mid \exists d{:}D.\varphi$$

where b is an expression of datatype $\mathbb{B}$, X is a variable of a set $\mathcal{X}$ of variables and e is a term of data-type D.

We define the ordering $\Rrightarrow$ on the set $D \to \mathbb{B}$ as $\varphi{\Rrightarrow}\psi$ iff for all $d{:}D$, we have $\varphi(d) \Rightarrow \psi(d)$. The set of first order boolean expressions $(D \to \mathbb{B}, \Rrightarrow)$ forms a complete lattice and is isomorphic to the powerset of the set D.

The propositional variables $X{\in}\mathcal{X}$, occurring as free variables in first order boolean expressions are bound in *first-order boolean equation systems*, used in the sequel. The interpretation of the variables X is given by an environment θ for propositional variables, assigning functions of type $D \to \mathbb{B}$ to the variables in the set $\mathcal{X}$. For a given environment θ, we write $\varphi\theta$ for the first order boolean expression that is obtained by replacing all occurrences of free variables X in φ with $\theta(X)$. We again use the convention to write $\theta[\psi/X]$, denoting the environment θ', defined as $\theta'(X') = \theta(X')$ for all $X' \not\equiv X$ and $\theta'(X) = \psi$.

We define the ordering $\leq$ on the set of propositional environments $\mathcal{X} \to D \to \mathbb{B}$ as $\theta_1 \leq \theta_2$ iff for all $X{\in}\mathcal{X}$, we have $\theta_1(X){\Rrightarrow}\theta_2(X)$. The set $(\mathcal{X} \to D \to \mathbb{B}, \leq)$ is (for fixed sets $\mathcal{X}$ and D), a complete lattice.

Definition 4.2 (*Interpretation of First Order Boolean Expression*).
Let θ be a propositional environment and η be a data environment. The *interpretation* of a first order boolean expression φ in the context of environments θ and η, written as $[\![\varphi]\!]\theta\eta$ is either true or false,

determined by the following induction:

$$
\begin{array}{lcl}
[\![b]\!]\theta\eta & = & [\![b]\!]\eta \\
[\![\varphi_1 \wedge \varphi_2]\!]\theta\eta & = & [\![\varphi_1]\!]\theta\eta \wedge [\![\varphi_2]\!]\theta\eta \\
[\![\varphi_1 \vee \varphi_2]\!]\theta\eta & = & [\![\varphi_1]\!]\theta\eta \vee [\![\varphi_2]\!]\theta\eta \\
[\![X(e)]\!]\theta\eta & = & \theta(X)([\![e]\!]\eta) \\
[\![\forall d{:}D.\varphi]\!]\theta\eta & = & \begin{cases} \text{true, if for all } v{:}D \text{ it holds that } [\![\varphi]\!]\theta(\eta[v/d]) \\ \text{false, otherwise} \end{cases} \\
[\![\exists d{:}D.\varphi]\!]\theta\eta & = & \begin{cases} \text{true, if there exists an } v{:}D \text{ such that } [\![\varphi]\!]\theta(\eta[v/d]) \\ \text{false, otherwise} \end{cases}
\end{array}
$$

Definition 4.3 (*First Order Boolean Equation System*).
A *first order boolean equation system* $\mathcal{E}$ is a finite sequence of equations of the form $\sigma X(d{:}D) = \varphi$. Here, σ represents either the greatest or least fixed points ν or μ, and $\varphi{:}D \to \mathbb{B}$ is a first order boolean expression. We require that all bound variables are distinct.

In the sequel, we refer to first order boolean equation systems as *equation systems*. The equation system $\mathcal{E}'$ that is obtained by applying an environment θ to an equation system $\mathcal{E}$ is the equation system in which every free variable $X{\in}\mathcal{X}$ is assigned the value $\theta(X)$.

Definition 4.4 (*Solution to an Equation System*).
Given a propositional environment θ, and an equation system $\mathcal{E}$. The solution $\mathcal{E}\theta$ to the equation system $\mathcal{E}$ is an environment that is defined as follows (see also e.g. [21], Definition 3.3), where σ is either the greatest fixpoint or the least fixpoint ν or μ.

$$
\begin{array}{lcl}
[\epsilon]\theta & = & \theta \\
[(\sigma X(d{:}D) = \varphi)\mathcal{E}]\theta & = & [\mathcal{E}](\theta[\sigma X.\varphi([\mathcal{E}]\theta)/X])
\end{array}
$$

where

$$
\begin{array}{lcl}
\mu X.\varphi([\mathcal{E}]\theta) & = & \bigwedge\{\psi{:}D \to \mathbb{B} \mid \varphi([\mathcal{E}]\theta[\psi/X]) \dot{\Rightarrow} \psi\} \\
\nu X.\varphi([\mathcal{E}]\theta) & = & \bigvee\{\psi{:}D \to \mathbb{B} \mid \psi \dot{\Rightarrow} \varphi([\mathcal{E}]\theta[\psi/X])\}
\end{array}
$$

Note: we represent an empty equation system as ϵ. The operators $\bigwedge$ and $\bigvee$ resp. denote the *greatest lower bound* and the *least upper bound* of the complete lattice $(D \to \mathbb{B}, \dot{\Rightarrow})$.

Lemma 4.5 (*Monotonicity of First Order Boolean Expressions*).
Let $X(d{:}D) = \varphi$ be an equation, let θ be a propositional environment and η a data environment. For an equation $X(d{:}D) = \varphi$, we define an operator $\Phi{:}(D \to \mathbb{B}) \to (D \to \mathbb{B})$ as $\Phi = \lambda F{:}D \to \mathbb{B}.\lambda v{:}D.[\![\varphi]\!](\theta[F/X])(\eta[v/d])$. The operator Φ is monotonic over the complete lattice $(D \to \mathbb{B}, \dot{\Rightarrow})$.

Proof. Assume we are given an equation $X(d{:}D) = \varphi$, a propositional environment θ and a data environment η, and assume we have first order boolean expressions $\psi_1, \psi_2{:}D \to \mathbb{B}$. We proceed by induction on the structure of φ.

- Suppose $\varphi \equiv b$. Then, $\Phi(\psi_1)$ equals $\lambda v{:}D.[\![b]\!](\theta[\psi_1/X])(\eta[v/d])$. As there is no occurrence of X in b, this is equivalent to $\lambda v{:}D.[\![b]\!](\eta[v/d])$. Using the same steps in reverse order, we find this is equivalent to $\lambda v{:}D.[\![b]\!](\theta[\psi_2/X])(\eta[v/d])$ and therefore $\Phi(\psi_2)$.

- Suppose $\varphi \equiv Y(e)$. Then, $\Phi(\psi_1)$ is equivalent to $\lambda v{:}D.\psi_1([\![e]\!])(\eta[v/d])^{(*)}$, given that $Y \equiv X$ (if not, then we are done immediately). Since $\psi_1 \dot{\Rightarrow} \psi_2$, we therefore also have that $^{(*)}$ is at most $\lambda v{:}D.\psi_2([\![e]\!])(\eta[v/d])$, which is equivalent to $\Phi(\psi_2)$.

- Suppose $\varphi \equiv \varphi_1 \wedge \varphi_2$. Assume for first order boolean expressions φ_1 and φ_2, we already have $\Phi_1(\psi_1) \dot{\Rightarrow} \Phi_1(\psi_1)$ and $\Phi_2(\psi_1) \dot{\Rightarrow} \Phi_2(\psi_2)$. Then, $\Phi(\psi_1)$ is equal to the conjunction of the functionals $\lambda v{:}D.[\![\varphi_1]\!](\theta[\psi_1/X])(\eta[v/d])$ and $\lambda v{:}D.[\![\varphi_2]\!](\theta[\psi_1/X])(\eta[v/d])$. By induction, we know this is at most the conjunction of $\lambda v{:}D.[\![\varphi_1]\!](\theta[\psi_2/X])(\eta[v/d])$ and $\lambda v{:}D.[\![\varphi_2]\!](\theta[\psi_2/X])(\eta[v/d])$, from which we can deduce $\Phi(\psi_2)$. Similarly, we prove $\Phi(\psi_1) \dot{\Rightarrow} \Phi(\psi_2)$ in the case of $\varphi \equiv \varphi_1 \vee \varphi_2$.

- Suppose $\varphi \equiv \forall e{:}D.\varphi_e$. Assume for first order boolean expressions φ_e, we already have for all $e{:}D$, $\Phi_e(\psi_1) \leq \Phi_e(\psi_2)$. Then, $\Phi(\psi_1)$ is equivalent to $\lambda v{:}D.[\![\varphi_e]\!](\theta[\psi_1/X])(\eta[v/d][x/e])$ for all $x{:}D$. By induction, this is at most $\lambda v{:}D.[\![\varphi_e]\!](\theta[\psi_2/X])(\eta[v/d][x/e])$ for all $x{:}D$, which is equal to $\Phi(\psi_2)$. Similarly, we prove $\Phi(\psi_1) \Rightarrow \Phi(\psi_2)$ in the case of $\varphi \equiv \exists e{:}D.\varphi_e$.

□

Lemma 4.6. Let θ, θ' be propositional environments and let $\mathcal{E}$ be an equation system. Then, if $\theta \leq \theta'$ we also have $[\mathcal{E}]\theta \leq [\mathcal{E}]\theta'$.

Proof. We use induction on the length of the equation system. Let $\theta \leq \theta'$ be propositional environments.

- suppose $\mathcal{E} = \epsilon$. Then, $[\epsilon]\theta = [\epsilon]\theta'$.
- Suppose $\mathcal{E}$ is of the form $(\sigma X(d{:}D) = \varphi)\mathcal{E}'$. Assume we have $[\mathcal{E}']\theta \leq [\mathcal{E}']\theta'$.

  Now, $[(\sigma X(d{:}D) = \varphi)\mathcal{E}']\theta \leq [(\sigma X(d{:}D) = \varphi)\mathcal{E}']\theta'$ follows from

  $[\mathcal{E}'](\theta[\sigma X(d{:}D).\varphi([\mathcal{E}']\theta)/X]) \leq [\mathcal{E}'](\theta'[\sigma X(d{:}D).\varphi([\mathcal{E}']\theta')/X])$ (see Def. 4.4).

  By induction, this holds, as $\theta[\sigma X(d{:}D).\varphi([\mathcal{E}']\theta)/X] \leq \theta'[\sigma X(d{:}D).\varphi([\mathcal{E}']\theta')/X]$, follows from $\theta \leq \theta'$ and Lemma 4.5.

□

We next discuss how to use the formalism of equation systems as an intermediate formalism for solving the model-checking problem for processes with data. We define a translation that takes a Linear Process Equation and a first order modal μ-calculus formula and yields an equation system. Then, verifying a first order modal μ-calculus formula on an LPE is equivalent to calculating the solution to the equation system that takes the LPE and the expression as its input.

Definition 4.7. Let φ be a first order μ-calculus expression, such that φ is of the form $(\sigma X(d{:}D).\Phi)(e)$, and let $Y(d_p{:}D_p) = \sum_{i:I}\sum_{e_i:D_i}[c_i(d,e_i)]{::}{\rightarrow} a_i(f_i(d,e_i)) \cdot Y(g_i(d,e_i))$ be a Linear Process Equation, where $d_p{:}D_p$ is the parameter of the process Y and a_i for $i{:}I$ is an action.
The equation system $\mathcal{E}$ that corresponds to the expression φ for the LPE Y, is given by $\mathbf{E}_{[]}(\varphi)$, where $[]$ denotes the empty list of parameters. The translation function $\mathbf{E}_{\vec{d_l}:\vec{D_l}}$ is defined by structural induction in Table 1, and action satisfaction, denoted by $\alpha \models \alpha'$ is defined using structural induction in Table 2 and assumes the LPE Y as given.

The translation function $\mathbf{E}$ breaks down the μ-calculus expression given as an argument into several equations. The left-hand side of each equation is defined by the function $\mathbf{E}$, whereas its right-hand side is given by the function $\tilde{\mathbf{E}}$. Below, we illustrate the translation by means of a small example.

Example 4.8. Consider a coffee-vending machine that produces either cappuccino or espresso on the insertion of a special coin. The coffee-vending machine is clever enough to notice when it can no longer dispense a type of coffee; it accepts coins as long as there is at least one type of coffee that can still be dispensed. If the machine has run out of a type of coffee, it signals this type must be replaced (which is assumed to be done immediately after the signal).

$$\begin{array}{rcl}
\textbf{proc}\ M(b{:}\mathbb{B}, c, e{:}\mathbb{N}) & = & [b \wedge c > 0]{::}{\rightarrow}\ \textit{cappuccino} \cdot M(\neg b, c-1, e)\\
& + & [b \wedge e > 0]{::}{\rightarrow}\ \textit{espresso} \cdot M(\neg b, c, e-1)\\
& + & [\neg b \wedge c + e > 0]{::}{\rightarrow}\ \textit{coin} \cdot M(\neg b, c, e)\\
& + & [\neg b \wedge c = 0]{::}{\rightarrow}\ \textit{refill}_{\textit{cappuccino}} \cdot M(b, C, e)\\
& + & [\neg b \wedge e = 0]{::}{\rightarrow}\ \textit{refill}_{\textit{espresso}} \cdot M(b, c, E)
\end{array}$$

Here, the boolean b indicates whether a coin has been inserted or not; the variable c, resp. e registers the number of servings of cappuccino, reps. espresso are left in the coffee-vending machine.

$\mathbf{E}_{\vec{d_l}:\vec{D_l}}(b)$	$\overset{\text{def}}{=}$	ϵ
$\mathbf{E}_{\vec{d_l}:\vec{D_l}}(X(d_f{:}D_f))$	$\overset{\text{def}}{=}$	ϵ
$\mathbf{E}_{\vec{d_l}:\vec{D_l}}(\Phi_1 \wedge \Phi_2)$	$\overset{\text{def}}{=}$	$\mathbf{E}_{\vec{d_l}:\vec{D_l}}(\Phi_1)\mathbf{E}_{\vec{d_l}:\vec{D_l}}(\Phi_2)$
$\mathbf{E}_{\vec{d_l}:\vec{D_l}}(\Phi_1 \vee \Phi_2)$	$\overset{\text{def}}{=}$	$\mathbf{E}_{\vec{d_l}:\vec{D_l}}(\Phi_1)\mathbf{E}_{\vec{d_l}:\vec{D_l}}(\Phi_2)$
$\mathbf{E}_{\vec{d_l}:\vec{D_l}}([\alpha]\Phi)$	$\overset{\text{def}}{=}$	$\mathbf{E}_{\vec{d_l}:\vec{D_l}}(\Phi)$
$\mathbf{E}_{\vec{d_l}:\vec{D_l}}(\langle\alpha\rangle\Phi)$	$\overset{\text{def}}{=}$	$\mathbf{E}_{\vec{d_l}:\vec{D_l}}(\Phi)$
$\mathbf{E}_{\vec{d_l}:\vec{D_l}}(\forall d{:}D.\Phi)$	$\overset{\text{def}}{=}$	$\mathbf{E}_{\vec{d_l}:\vec{D_l};d:D}(\Phi)$
$\mathbf{E}_{\vec{d_l}:\vec{D_l}}(\exists d{:}D.\Phi)$	$\overset{\text{def}}{=}$	$\mathbf{E}_{\vec{d_l}:\vec{D_l};d:D}(\Phi)$
$\mathbf{E}_{\vec{d_l}:\vec{D_l}}((\sigma X(d_f{:}D_f).\Phi)(d))$	$\overset{\text{def}}{=}$	$(\sigma\tilde{X}(d_f{:}D_f, d_p{:}D_p, \vec{d_l}{:}\vec{D_l}) = \tilde{\mathbf{E}}_{\vec{d_l}:\vec{D_l}}(\Phi)\ )\ \mathbf{E}_{\vec{d_l}:\vec{D_l}}(\Phi)$
$\tilde{\mathbf{E}}_{\vec{d_l}:\vec{D_l}}(b)$	$\overset{\text{def}}{=}$	b
$\tilde{\mathbf{E}}_{\vec{d_l}:\vec{D_l}}(X(d))$	$\overset{\text{def}}{=}$	$\tilde{X}(d, d_p, \vec{d_l})$
$\tilde{\mathbf{E}}_{\vec{d_l}:\vec{D_l}}(\Phi_1 \wedge \Phi_2)$	$\overset{\text{def}}{=}$	$\tilde{\mathbf{E}}_{\vec{d_l}:\vec{D_l}}(\Phi_1) \wedge \tilde{\mathbf{E}}_{\vec{d_l}:\vec{D_l}}(\Phi_2)$
$\tilde{\mathbf{E}}_{\vec{d_l}:\vec{D_l}}(\Phi_1 \vee \Phi_2)$	$\overset{\text{def}}{=}$	$\tilde{\mathbf{E}}_{\vec{d_l}:\vec{D_l}}(\Phi_1) \vee \tilde{\mathbf{E}}_{\vec{d_l}:\vec{D_l}}(\Phi_2)$
$\tilde{\mathbf{E}}_{\vec{d_l}:\vec{D_l}}([\alpha]\Phi)$	$\overset{\text{def}}{=}$	$\bigwedge_{i:I} \forall_{e_i:D_i}(a_i(f_i(d,e_i)) \models \alpha \wedge c_i(d,e_i)) \rightarrow$ $\tilde{\mathbf{E}}_{\vec{d_l}:\vec{D_l}}(\Phi)[g_i(d,e_i)/d_p]$
$\tilde{\mathbf{E}}_{\vec{d_l}:\vec{D_l}}(\langle\alpha\rangle\Phi)$	$\overset{\text{def}}{=}$	$\bigvee_{i:I} \exists_{e_i:D_i}(a_i(f_i(d,e_i)) \models \alpha \wedge c_i(d,e_i) \wedge$ $\tilde{\mathbf{E}}_{\vec{d_l}:\vec{D_l}}(\Phi)[g_i(d,e_i)/d_p])$
$\tilde{\mathbf{E}}_{\vec{d_l}:\vec{D_l}}(\forall d{:}D.\Phi)$	$\overset{\text{def}}{=}$	$\forall d{:}D.\tilde{\mathbf{E}}_{\vec{d_l}:\vec{D_l};d:D}(\Phi)$
$\tilde{\mathbf{E}}_{\vec{d_l}:\vec{D_l}}(\exists d{:}D.\Phi)$	$\overset{\text{def}}{=}$	$\exists d{:}D.\tilde{\mathbf{E}}_{\vec{d_l}:\vec{D_l};d:D}(\Phi)$
$\tilde{\mathbf{E}}_{\vec{d_l}:\vec{D_l}}((\sigma X(d_f{:}D_f).\Phi)(d))$	$\overset{\text{def}}{=}$	$\tilde{X}(d, d_p, \vec{d_l})$

Table 1: Translation of first order μ-calculus formula and LPE to an equation system. Note that $\tilde{X}$ is a fresh variable, associated to the variable X

$a(d) \models a'(d')$	$\overset{\text{def}}{=}$	$a = a' \wedge d = d'$
$a(d) \models \top$	$\overset{\text{def}}{=}$	true
$a(d) \models \neg\alpha$	$\overset{\text{def}}{=}$	$\neg(a(d) \models \alpha)$
$a(d) \models \alpha_1 \wedge \alpha_2$	$\overset{\text{def}}{=}$	$(a(d) \models \alpha_1) \wedge (a(d) \models \alpha_2)$
$a(d) \models \alpha_1 \vee \alpha_2$	$\overset{\text{def}}{=}$	$(a(d) \models \alpha_1) \vee (a(d) \models \alpha_2)$
$a(d) \models \exists d'{:}D.\alpha$	$\overset{\text{def}}{=}$	$\exists d'{:}D.(a(d) \models \alpha)$
$a(d) \models \forall d'{:}D.\alpha$	$\overset{\text{def}}{=}$	$\forall d'{:}D.(a(d) \models \alpha)$

Table 2: Action Satisfaction

Consider the first order μ-calculus expression $\mu Z.\langle \mathit{coin} \vee \mathit{cappuccino}\rangle Z \vee \langle \mathit{refill}_{\mathit{cappuccino}}\rangle \top$, expressing that there exists a path where eventually cappuccino is refilled when cappuccino is the only thing that has been ordered. Following the translation of Def. 4.7, we obtain the following equation system.

$$\mu Z(b{:}\mathbb{B}, c, e{:}\mathbb{N}) = (\neg b \wedge c + e > 0 \wedge Z(\neg b, c, e)) \vee (b \wedge c > 0 \wedge Z(\neg b, c-1, e)) \vee (\neg b \wedge c = 0)$$

Notice that, even though the first order modal μ-calculus expression did not use parameterised variables, the resulting equation system consists of an equation carrying the parameters of the Linear Process Equation.

We continue by establishing two results that allow us to define an algorithm for computing a solution to an equation system. The first lemma states that for an arbitrary equation system, we may replace an occurrence of an equation variable with its first order boolean expression in all equations prior to its defining equation.

Lemma 4.9. Let $\mathcal{E}_1, \mathcal{E}_2$ and $\mathcal{E}_3$ be equation systems and let $\sigma_1 X_1(d{:}D) = \varphi$ and $\sigma_2 X_2(e{:}D) = \psi$ be equations. Then, we have the following identity:

$$\begin{array}{ll} & [\![\mathcal{E}_1(\sigma_1 X_1(d{:}D) = \varphi)\mathcal{E}_2(\sigma_2 X_2(e{:}D) = \psi)\mathcal{E}_3]\!]\theta \\ = & \\ & [\![\mathcal{E}_1(\sigma_1 X_1(d{:}D) = \varphi[\psi/X_2])\mathcal{E}_2(\sigma_2 X_2(e{:}D) = \psi)\mathcal{E}_3]\!]\theta \end{array}$$

Proof. The proof is analogous to the proof of Lemma 6.3 in [21]. □

If we have the solution to a single equation in an equation system, then we can remove this equation from the equation system and update the environment to store the solution to this single equation. This means that if we can successively solve all single equations, the solution of the entire equation system follows.

Lemma 4.10. Let $\mathcal{E}, \mathcal{E}'$ be equation systems and let $\sigma X(d{:}D) = \psi$ be an equation, where $X \notin FV(\psi)$. Let θ be an arbitrary propositional environment. Then $[\mathcal{E}(\sigma X(d{:}D) = \psi)\mathcal{E}']\theta = [\mathcal{E}\mathcal{E}']\theta[\psi/X]$.

Proof. The proof proceeds by induction on the size of the equation system $\mathcal{E}$.

- $[(\sigma X(d{:}D) = \psi)\mathcal{E}']\theta$ is by definition equivalent to $[\mathcal{E}'](\theta[\sigma X(d{:}D).\psi/X])$, which in turn is equivalent to $[\mathcal{E}'](\theta[\psi/X])$, as since $X \notin FV(\psi)$, $\sigma X(d{:}D).\psi$ is ψ.
- Suppose $\mathcal{E}$ is of the form $(\sigma' Y(d{:}D) = \varphi)\mathcal{E}_0$, and assume $[\mathcal{E}_0(\sigma X(d{:}D) = \psi)\mathcal{E}']\theta = [\mathcal{E}_0\mathcal{E}']\theta[\psi/X]$ for all environments θ. Then, $[\mathcal{E}(\sigma X(d{:}D) = \psi)\mathcal{E}']\theta$ is by definition equivalent to $[\mathcal{E}_0(\sigma X(d{:}D) = \psi)\mathcal{E}'](\theta[\sigma' Y(d{:}D).\varphi/Y])$, which is by induction equivalent to $[\mathcal{E}_0\mathcal{E}'](\theta[\sigma' Y(d{:}D).\varphi/Y])[\psi/X])$. Again, by definition, this is equivalent to $[\mathcal{E}\mathcal{E}']\theta[\psi/X]$.

□

5 Algorithm

Mader [21] describes an algorithm for solving boolean equation systems. The method she uses resembles the well-known Gauß elimination algorithm for solving linear equation systems, and is therefore also referred to as Gauß elimination. The algorithm we use (see Table 3) is an extension of the Gauß elimination algorithm of [21]. The essential difference is the addition of an extra loop for calculating a stable point in the approximation for each first order boolean equation.

The reduction of a first order boolean equation system proceeds in two separate steps. First, a stabilisation step is issued, in which a first order boolean equation $\sigma_i X_i(d{:}D) = \varphi_i$ is reduced to a stable equation $\sigma_i X_i(d{:}D) = \varphi_i'$, where φ_i' is an expression containing no occurrences of X_i. Second, we substitute each occurrence of X_i by φ_i' in the rest of the equations of the first order boolean equation system. Since there are no more occurrences of X_i in the right-hand side of the equations, it suffices to reduce a smaller first order boolean equation system. The algorithm terminates iff the stabilisation step terminates for each first order boolean equation.

Input: $(\sigma_1 X_1(d_1{:}D_1) = \varphi_1) \dots (\sigma_n X_n(d_n{:}D_n) = \varphi_n)$.

```
1.  $i := n$;
2.  **while not** $i = 0$
3.    **do**
4.      $j := 0; \psi_0 := \sigma_{b_i}$;
5.      **repeat**
6.          $\psi_{j+1} := \varphi_i[X_i := \psi_j]$;
7.          $j := j + 1$
8.      **until** $(\psi_j \equiv \psi_{j-1})$
9.      $\varphi_i := \psi_j$;
10.     **for** $k = 1$ **to** $i - 1$ **do** $\varphi_k := \varphi_k[X_i := \varphi_i]$ **od** ;
11.     $i := i - 1$
12.   **od**
```

Remark: σ_{b_i} is $\top$ if $\sigma_i = \nu$, else $\bot$

Table 3: Algorithm for computing the solution of an equation system

Theorem 5.1 (*Soundness*).
On termination of the algorithm in Table 3, the solution of the given equation system has been computed.

Proof. The technique to solve a single equation is based on well-known approximation techniques. Termination of this approximation means we have computed a solution to a single equation. This solution can then be substituted in the lexicographical smaller equations of the equation system, as a result of Lemmas 4.9 and 4.10. Termination of the algorithm, therefore means we have correctly computed the solution to all equations in the equation system. □

Note that as it is undecidable whether a first order boolean equation system has a solution, the possible non-termination of our algorithm is unavoidable. Below, we provide three small examples, showing the application of the algorithm and its possible non-termination on systems that use data.

Example 5.2. Consider a counter that counts up to nine, starting from zero, and at nine cycles back to zero. Each time the counter increases, an *inc* event is issued. Upon reaching nine, the counter issues a *reset* event, signalling the counter has been reset to zero. A process algebraic description (in LPE form) of such a process is provided below.

$$\begin{array}{lcl} \textbf{proc } C(n{:}\mathbb{N}) & = & [n \geq 9] {::}\!\rightarrow \textit{reset} \cdot C(0) \\ & + & [n < 9] {::}\!\rightarrow \textit{inc} \cdot C(n+1) \end{array}$$

Our goal is to verify whether it is possible to always execute a *reset* action. To this end, we specify the formula $\mu Y.[\top]Y \vee \langle \textit{reset} \rangle \top$. This basically expresses that on all infinite paths, eventually a *reset* action is executed. The first order boolean equation system for this expression is (after reduction) $\mu Z(n{:}\mathbb{N}) = (n \geq 9 \vee Z(n+1))$.

Following the algorithm, we first compute ψ_0 and ψ_1, being resp. $\bot$ and $n \geq 9$. Then, we iterate until we end up with a formula $\psi_{10} = 0 \leq n$, which is equivalent to ψ_{11}. Since this is a stable solution of the equation, we can assess the truth of the equation system by substituting ψ_{10} for Z in our equation, thereby obtaining $\mu Z(n{:}\mathbb{N}) = \top$.

Example 5.3. As an example of a system with an infinite state-space, we consider a process that counts from zero to infinity, and reports its current state via an action *current*. A process algebraic description in LPE form is provided below.

$$\textbf{proc } C(n{:}\mathbb{N}) = \textit{current}(n) \cdot C(n+1)$$

Given the simplicity of this process, it is unfortunate to find that with most current technologies, we cannot even automatically prove absence of deadlock for process C. Using our algorithm, this boils down to verifying $\nu X.\langle\top\rangle\top \wedge [\top]X$ on the process C. Following the translation, we derive the associated equation system $\nu Z(n{:}\mathbb{N}) = Z(n+1) \wedge \top$. Substituting $\top$ for $Z(n+1)$ immediately leads to the stable solution $\top$.

Example 5.4. Consider a process *C* representing a counter that counts down from a randomly chosen natural number to zero and then randomly selects a new natural number.

$$\begin{array}{rcl} \textbf{proc } C(n{:}\mathbb{N}) & = & \sum_{m:\mathbb{N}} [n = 0] {::}{\rightarrow}\ \textit{reset} \cdot C(m) \\ & + & [n > 0] {::}{\rightarrow}\ \textit{dec} \cdot C(n-1) \end{array}$$

Our goal is again to verify whether it is possible to always execute a *reset* action. This is again expressed as follows: $\mu Y.[\top]Y \vee \langle \textit{reset}\rangle\top$. The equation system for this expression is $\mu Z(n{:}\mathbb{N}) = n = 0 \vee Z(n-1)$.

The algorithm prescribes computing a stable solution for this equation. However, this computation does not terminate, as we end up with approximations ψ_k, where $\psi_k = n \leq k$. This means, we cannot find a ψ_j, such that $\psi_j = \psi_{j+1}$, and therefore, the algorithm does not terminate. However, it is straightforward to see that the minimal solution for this equation is $\mu Z(n{:}\mathbb{N}) = \top$.

6 Verification of Data-Dependent Systems in Practice

Based on our algorithm, described in the previous section, we have implemented a prototype of a tool[1]. In this section, we briefly sketch this implementation, without going into too much detail. To test the applicability of the prototype, we have applied it on a large number of protocols. For brevity, we here report on the findings of only two smaller protocols having an infinite state-space.

6.1 Implementation

The prototype implementation of our algorithm employs *Equational Binary Decision Diagrams* (EQ-BDDs) [18] for representing first-order boolean expressions. These EQ-BDDs extend on standard BDDs [7] by explicitly allowing equality on nodes. We first define the grammar for EQ-BDDs.

Definition 6.1 (*Grammar for EQ-BDDs*).
We assume a set P of propositions and a set V of variables. The formulae we consider are given according to the following grammar.

$$\Phi ::= \mathbf{0} \mid \mathbf{1} \mid \mathrm{ITE}(V = V, \Phi, \Phi) \mid \mathrm{ITE}(P, \Phi, \Phi)$$

The constants **0** and **1** represent *false* and *true*. An expression of the form ITE(φ, ψ, ξ) must be read as an *if-then-else* construct, i.e. $(\varphi \wedge \psi) \vee (\neg\varphi \wedge \xi)$, or, alternatively, $(\varphi \Rightarrow \psi) \wedge (\neg\varphi \Rightarrow \xi)$. For data variables d and e, and φ of the form $d = e$, the extension to EQ-BDDs is used, i.e. we explicitly use ITE$(d = e, \psi, \xi)$ in such cases. Using the standard BDD and EQ-BDD encodings [7, 18], we can then represent all quantifier-free first order boolean expressions. The representation of expressions that contain quantifiers over finite domains is done in a straightforward manner, i.e. we construct explicit encodings for each distinct element in the domain. Expressions containing quantifiers over infinite domains are in general problematic, when it comes to representation and calculation. The following theorem, however, identifies a number of cases in which we can deal with these.

Theorem 6.2 Quantification over data-types can be reduced in the following cases: Suppose d does not occur in ψ. We find:

- $\exists d{:}D.\mathrm{ITE}(d = e, \varphi, \psi) = \varphi[e/d] \vee \psi$ provided D contains at least two (different) elements.
- $\forall d{:}D.\mathrm{ITE}(d = e, \varphi, \psi) = \varphi[e/d] \wedge \psi$ provided D contains at least two (different) elements.

[1]In due time, the techniques developed in this paper are intended to culminate in a freely available tool that is distributed as part of the μCRL tool-suite [4].

- $\exists d{:}D.\text{ITE}(d = e_1, \varphi_1, \text{ITE}(d = e_2, \varphi_2, \ldots, \text{ITE}(d = e_n, \varphi_n, \psi) \ldots)) = \bigvee_{1 \leq i \leq n}((\bigwedge_{1 \leq j < i} e_j \neq e_i) \wedge \varphi_i[e_i/d]) \vee \psi$ provided D contains at least one element not in $\{e_i | 1 \leq i \leq n\}$.

- $\forall d{:}D.\text{ITE}(d = e_1, \varphi_1, \text{ITE}(d = e_2, \varphi_2, \ldots, \text{ITE}(d = e_n, \varphi_n, \psi) \ldots)) = \bigwedge_{1 \leq i \leq n}((\bigvee_{1 \leq j < i} e_j = e_i) \vee \varphi_i[e_i/d]) \wedge \psi$ provided D contains at least one element not in $\{e_i | 1 \leq i \leq n\}$.

Proof. The identities follow directly from the observations that

- $\exists d{:}D.\text{ITE}(d = e, \varphi, \psi) = \varphi[e/d] \vee \exists d{:}D(d \neq e \wedge \psi)$.

- $\forall d{:}D.\text{ITE}(d = e, \varphi, \psi) = \varphi[e/d] \wedge \forall d{:}D(d = e \vee \psi)$.

□

Note that the last two items of the theorem above actually say that if d only occurs in equations within a formula φ and the domain of D is sufficiently large, quantification over d can be removed, because each such formula can be brought into the form given above.

Even though Theorem 6.2 applies to a restricted class of first order boolean expressions, we find that in practice, it adds considerably to the verification power of the prototype implementation.

6.2 Example Verifications

We have used the prototype on several applications, including many communications protocols, such as the IEEE-1394 firewire, the sliding window protocol, the bounded retransmission protocol, etc. As an example of the capabilities, we here report on the use of our prototype on two small systems, viz. Lamport's Bakery Protocol [25], and the Alternating Bit Protocol [3]. Both systems have infinite state-spaces due to the use of infinite data domains, and the properties we are interested in are both liveness and safety properties. We first briefly introduce the two systems, and the properties we study.

Bakery Protocol The first example we consider is Lamport's Bakery protocol. A μCRL specification of this protocol is given in Table 4. The data-types are given as abstract data-types, but are omitted in this presentation. The bakery protocol we consider is restricted to two processes. An informal explanation of the protocol is as follows. A process, waiting to enter its critical section can choose a number, larger than any other number already chosen. Then, the process with the lower number is allowed to enter the critical section before the process with the larger number.

Given the unbounded growth of the numbers that can be chosen, the protocol clearly has an infinite state-space. Hence, verification of the bakery protocol is usually performed on an altered version, abstracting in some way from these numbers. Our techniques, however, are immediately applicable. Below, we list a number of key properties we verify for the bakery protocol.

1. No deadlock can occur, i.e. in every reachable state of the protocol, an action is enabled,

2. All processes requesting a number can eventually enter the critical section,

3. All processes requesting a number inevitably enter the critical section.

The results for the verification of these properties are listed in Table 5. The second and third property deserve some extra attention, as their difference is quite subtle; it is best compared to the difference between "may" and "must". The second property states that if a process requests a number, there is a path leading towards a state in which a process may gain access to the critical section. The third property states that if a process requests a number, all paths inevitably lead to states in which the process must gain access to the critical section. Note that requesting a number (using action *request*) is not a sufficient condition for entering the critical section, as this is only guaranteed when the process has also received its number (using action *get*). This explains why the third property does not hold: it can be the case that the request of the number is not followed by the receiving of a number, in which case the other process can infinitely often access the critical section.

comm *get*, *send* $= c$

init $\partial_{\{get,\ send\}}(P(\top) \| P(\bot))$

proc $P(b{:}\mathbb{B}) =$
  $request(b) \cdot P_0(b, 0)$
$+send(b, 0) \cdot P(b)$

proc $P_0(b{:}\mathbb{B}, n{:}\mathbb{N}) =$
$\sum_{m:\mathbb{N}} get(\neg b, m) \cdot P_1(b, m+1)$
$+send(b, n) \cdot P_0(b, n)$

proc $P_1(b{:}\mathbb{B}, n{:}\mathbb{N}) =$
$\sum_{m:\mathbb{N}} get(\neg b, m) \cdot (C_1(b, n) \lhd n < m \vee m = 0 \rhd P_1(b, n))$
$+send(b, n) \cdot P_1(b, n)$

proc $C_1(b{:}\mathbb{B}, n{:}\mathbb{N}) =$
  $enter(b) \cdot C_2(b, n)$
$+send(b, n) \cdot C_1(b, n)$

proc $C_2(b{:}\mathbb{B}, n{:}\mathbb{N}) =$
  $leave(b) \cdot P(b)$
$+send(b, n) \cdot C_2(b, n)$

Table 4: Lamport's Bakery Protocol

Alternating Bit Protocol The *Alternating Bit Protocol* (ABP, see e.g. [3]) is a basic communications protocol utilising two channels. A sender sends a message, tagged with a bit, via an unreliable channel. It repeatedly resends this message (including the bit), until it receives an acknowledgement (with the right bit) from the receiver, via the other channel. It then starts the entire procedure again with a new message, and inverts the bit it sends along with the message.

The ABP is a famous communications protocol, and is often used to illustrate that a formalism or technique is capable of dealing with real systems of small to medium size. When applying well-established, fully-automatic techniques, the data that is transmitted in this (and other) communications protocols, has to be fixed. Here, we show that, with the use of our prototype, no alterations to the ABP are necessary, and the messages we transmit are indeed arbitrarily chosen from an infinite set of messages. Communications protocols usually have an external behaviour, similar to the behaviour of a buffer, i.e. messages sent at one end are eventually received at the other end. The ABP is no exception to this rule. The properties we verified for ABP are listed below.

Nr.	Formal Property	Satisfied	Time
1.	$\nu X.([\top]X \wedge \langle\top\rangle\top)$	yes	2sec
2.	$\nu X.([\top]X \wedge \forall b{:}\mathbb{B}.[request(b)]\mu Y.\langle\top\rangle Y \vee \langle enter(b)\rangle\top)$	yes	60sec
3.	$\nu X.([\top]X \wedge \forall b{:}\mathbb{B}.[request(b)]\mu Y.(([\top]Y \wedge \langle\top\rangle\top) \vee \langle enter(b)\rangle\top))$	no	5sec

Table 5: Verification results of the Bakery protocol. All computations were performed on a 1 GHz Intel Pentium III processor with 512Mb main memory running Linux version 2.4. The field "Time" states the amount of computation time needed to perform the verification.

comm $r2, s2 = c2$
$r3, s3 = c3$
$r5, s5 = c5$
$r6, s6 = c6$

init $\partial_{\{r2,r3,r5,r6,s2,s3,s5,s6\}}(S \| K \| L \| R)$

proc $S =$
$S(0) \cdot S(1) \cdot S$

proc $S(n{:}\textit{bit}) =$
$\sum_{d:D} \; r1(d) \cdot S(d, n)$

proc $S(d{:}D, n{:}\textit{bit}) =$
$s2(d, n) \cdot ((r6(\textit{invert}(n)) + r6(e)) \cdot S(d, n) + r6(n))$

proc $R =$
$R(1) \cdot R(0) \cdot R$

proc $R(n{:}\textit{bit}) =$
$(r3(e) + \sum_{d:D} \; r3(d, n)) \cdot s5(n) \cdot R(n)$
$+ \sum_{d:D} \; r3(d, \textit{invert}(n)) \cdot s4(d) \cdot s5(\textit{invert}(n))$

proc $K =$
$\sum_{d:D} \; \sum_{n:\textit{bit}} \; r2(d, n) \cdot (i \cdot s3(d, n) + i \cdot s3(e)) \cdot K$

proc $L =$
$\sum_{n:\textit{bit}} \; r5(n) \cdot (i \cdot s6(n) + i \cdot s6(e)) \cdot L$

Table 6: Alternating Bit Protocol

1. No deadlock can occur,
2. A message that is sent always eventually is received,
3. The protocol does not create messages,
4. The protocol does not duplicate messages.

The two latter properties state that the protocol does not allow for any miracles to happen. The results of the verification of these properties are listed in Table 7.

7 Closing Remarks

7.1 Discussion

In this paper, we discussed a technique for model checking systems that depend on (possibly infinite) datatypes. The techniques and algorithm we used, are based upon the techniques and algorithm, described by e.g. Mader [21]. We utilise equation systems as an intermediate formalism to which we translate both our system description (given in μCRL) and a property description (given in a first order modal μ-calculus). Given that the problem in general is not decidable, we have assessed the applicability of our solution on a large number of cases, both finite and infinite systems. In Section 6, we have reported on the results obtained for two small systems with infinite state-spaces, showing that the tool indeed functions as expected.

Nr.	Formal Property	Satisfied	Time
1.	$\nu X.([\top]X \wedge \langle\top\rangle\top)$	yes	2sec
2.	$\nu X.([\top]X \wedge \forall d{:}D.[r1(d)]\mu Y.\langle\top\rangle Y \vee \langle s4(d)\rangle\top)$	yes	15sec
3.	$\nu X.(\forall d{:}D.([\neg r1(d)]X \wedge [s4(d)]\bot))$	yes	60sec
4.	$\nu X.([\top]X \wedge \forall d{:}D.[r1(d)]\nu Y.([\neg r1(d) \vee s4(d)]Y \wedge [r1(d)]\bot)$	yes	5sec

Table 7: Verification results of the Alternating Bit Protocol. All computations were performed on a 1 GHz Intel Pentium III processor with 512Mb main memory running Linux version 2.4. The field "Time" states the amount of time needed to perform the verification.

The experiences we obtained by verifying properties in the other systems (e.g. the Bounded Retransmission Protocol) show that the tool indeed sometimes fails to terminate, but in general, termination is achieved in an acceptable run-time. Indeed, in several instances, we have been able to use our prototype in situations where existing, well-established, tool-sets failed to produce the result. Slightly surprising is one such instance, viz. a subsystem of the *EUV Wafer Stepper Machine* [22, 2] of ASML, for which we had two different specifications. Using our prototype, we were able to verify properties of the specification given in [22] where the general μCRL tool-suite [4] (using the Cæsar-Aldébaran [12, 11] front-end) failed to even build an internal representation of the state-space. The specification, given in [2] was no problem for the general μCRL tool-suite, yet proved troublesome for our prototype.

7.2 Summary

Summarising, we find that the verifications take in many cases an acceptable run-time, even though for systems with finite state spaces, our prototype is often outperformed by most well-established tool-suites. However, we expect some improvements can still be made on the prototype. More importantly, as we have demonstrated in Section 6, we are able to automatically verify properties of systems with infinite state-spaces in a reasonable time. Also, we have successfully applied our prototype on a system with a finite, yet extremely large state-space, for which established techniques failed to calculate the exact state-space. Since this is where the current state-of-the-art technology breaks down, our technique is clearly an improvement on the current technology.

The prototype implementation, however, also revealed a number of new issues to be resolved. We were not able to prove absence of deadlock of the Bounded Retransmision Protocol [17], with arbitrary bound on the number of retransmissions. As it turns out, the current rewrite strategy, used for rewriting the abstract data types is not particularly well-suited for dealing with this case. The possible solutions to this problem may lie in considering e.g. *Associative-Commutative* rewriting (see e.g. [10]).

Still, several other issues remain to be investigated. First, in [13], Groote and Mateescu provide four deduction rules for manually establishing the truth or falsity of a formula on an infinite state-space. It is interesting to see if some, or parts of these proof rules can be automated, thereby solving problems that our algorithm cannot deal with. Second, techniques, such as developed by Pnueli et al. [24], and Bryant et al. [8, 26] may be incorporated to increase the success rate of the algorithm we proposed. Third, the prototype has only limited diagnostic features. It requires additional research to obtain more meaningful diagnostics, such as failure traces, to increase the usability of the prototype.

References

[1] P. Abdulla, A. Bouajjani, and B. Jonsson. On-the-fly analysis of systems with unbounded, lossy FIFO channels. In A.J. Hu and M.Y. Vardi, editors, *10th International Conference on Computer Aided Verification, CAV'98*, volume 1427 of *Lecture Notes in Computer Science*, pages 305–318. Springer-Verlag, 1998.

[2] K. Aben, P. van den Brand, and R. Schouten. Specification of the EUV wafer handler controller, July 2002. Eindhoven University of Technology.

[3] J.C.M. Baeten and W.P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[4] S.C.C. Blom, W.J. Fokkink, J.F. Groote, I. Van Langevelde, B.Lisser, and J.C. van de Pol. μCRL: A toolset for analysing algebraic specification. In *13th International Conference on Computer Aided Verification, CAV'01*, volume 2102 of *Lecture Notes in Computer Science*, pages 250–254. Springer-Verlag, 2001.

[5] B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. The power of QDDs. In P. van Hentenryck, editor, *Static Analysis, 4th International Symposium, SAS '97*, volume 1302 of *Lecture Notes in Computer Science*, pages 172–186. Springer-Verlag, 1997.

[6] J.C Bradfield and C. Stirling. Modal logics and mu-calculi. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 4, pages 293–330. Elsevier (North-Holland), 2001.

[7] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[8] R.E. Bryant, S.K. Lahiri, and S.A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *14th International Conference on Computer Aided Verification, CAV 2002*, volume 2404 of *Lecture Notes in Computer Science*, pages 78–92. Springer-Verlag, 2002.

[9] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using Presburger arithmetic. In O. Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV'97*, volume 1254 of *Lecture Notes in Computer Science*, pages 400–411. Springer-Verlag, 1997.

[10] N. Dershowitz and D.A. Plaisted. Rewriting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 9, pages 535–610. Elsevier Science, 2001.

[11] H. Garavel. OPEN/CAESAR: An Open Software Architecture for Verification, Simulation, and Testing. In B. Steffen, editor, *Proceedings of the Fourth International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'98*, volume 1384 of *Lecture Notes in Computer Science*, pages 68–84. Springer-Verlag, March 1998.

[12] Hubert Garavel. An overview of the EUCALYPTUS toolbox. In Z. Brezočnik and T. Kapus, editors, *Proceedings of the COST 247 International Workshop on Applied Formal Methods in System Design (Maribor, Slovenia)*, pages 76–88. University of Maribor, 1996.

[13] J.F. Groote and R. Mateescu. Verification of temporal properties of processes in a setting with data. In A.M. Haeberer, editor, *Proceedings of the 7th International Conference on Algebraic Methodology and Software Technology AMAST'98 (Amazonia, Brazil)*, volume 1548 of *Lecture Notes in Computer Science*, pages 74–90. Springer-Verlag, January 1999.

[14] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes '94*, Workshops in Computing Series, pages 26–62. Springer Verlag, 1995.

[15] J.F. Groote, A. Ponse, and Y.S. Usenko. Linearization in parallel pCRL. *Journal of Logic and Algebraic Programming*, 48(1-2):39–72, June 2001.

[16] J.F. Groote and M.A. Reniers. Algebraic process verification. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 17, pages 1151–1208. Elsevier (North-Holland), 2001.

[17] J.F. Groote and J.C. van de Pol. A bounded retransmission protocol for large data packets. a case study in computer checked verification. In M. Wirsing and M. Nivat, editors, *Proceedings of AMAST'96, Munich*, volume 110 of *Lecture Notes in Computer Science*, pages 536–550. Springer-Verlag, 1996.

[18] J.F. Groote and J.C. van der Pol. Equational binary decision diagrams. In M. Parigot and A. Voronkov, editors, *Logic for Programming and Reasoning, LPAR2000*, volume 1955 of *Lecture Notes in Artificial Intelligence*, pages 161–178. Springer-Verlag, 2000.

[19] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[20] S.P. Luttik. *Choice quantification in process algebra*. PhD thesis, University of Amsterdam, April 2002.

[21] A. Mader. *Verification of Modal Properties Using Boolean Equation Systems*. PhD thesis, Technical University of Munich, 1997.

[22] L.E. Mamane, L. Cleophas, and E. Smeets. Specification & analysis of embedded systems assignment report, July 2002. Eindhoven University of Technology.

[23] R. Milner. *Communication and Concurrency*. Prentice Hall International, 1989.

[24] A. Pnueli, J. Xu, and L. Zuck. Liveness with (0,1,infinity)-counter abstraction. In E. Brinksma and K.G. Larsen, editors, *14th International Conference on Computer Aided Verification, CAV 2002*, volume 2404 of *Lecture Notes in Computer Science*, pages 107–122. Springer-Verlag, 2002.

[25] M. Raynal. *Algorithms for Mutual Exclusion*. North Oxford Academic, 1986.

[26] O. Strichman, S.A. Seshia, and R.E. Bryant. Deciding separation formulas with SAT. In *14th International Conference on Computer Aided Verification, CAV 2002*, volume 2404 of *Lecture Notes in Computer Science*, pages 209–222. Springer-Verlag, 2002.

[27] Y.S. Usenko. *Linearization in μCRL*. PhD thesis, Eindhoven University of Technology, December 2002.