Experiments in Unifying Model Checking Approaches

Radboud University Nijmegen

Master Thesis

Author:
Peter Maandag

Supervisor:
Hans Zantema

September 3, 2014

Abstract

A new tool is proposed that unifies different model checking approaches through a new scripting language. The tool makes advantage of this new language that allows the specification of semantically equivalent models in many different approaches to be defined just once. This specification is translated to several back-ends, which are then run in parallel. Experiments show that the unified approach is beneficial to identifying the optimal model checking approach.

1 Introduction

Model checking refers to a set of algorithms for exploring the state space of a transition system to determine compliance with a specification or to verify properties of desired behaviour. These algorithms execute a search of the associated state transition graphs and can perform exhaustive verification in a highly automatic manner (Clarke, Biere, Raimi, & Zhu, 2001). The simplest model checking algorithm traverses the transition graph while looking for desired properties. During the search a list of visited states is kept in memory to avoid getting stuck in loops and to make the search process more efficient. Besides this method, different model checking techniques exist that are based on satisfiability solving or a combination of these and conventional model checking algorithms. One of these techniques is called symbolic state space exploration.

In symbolic model checking, a breadth first search of the state space is performed through the construction of Binary Decision Diagrams (BDDs) (Mishchenko, 2001). The BDDs hold the characteristic functions of sets of states, and allow computation of transitions among sets of states rather than individual states. Although many formulas can be represented efficiently through BDDs, the method still suffers from state space explosion, because its size is worst case exponential in the size of the system that is modelled (McMillan, 1993).

A newer type of model checking technique, bounded model checking (BMC) with satisfiability solving, has given promising results (Yin, He, & Gu, 2013). The method can be used on its own to verify safety and liveness properties, but is also commonly used as a complement to BDD-based model checking. Safety and liveness properties are two important classes of properties. Model checking with BMC and all aforementioned methods can be used to verify both classes of properties. The definitions and examples of verifying such properties will be discussed further in section 2.1.

The research of (Clarke et al., 2001) describes bounded model checking as follows: "Essentially, there are two steps in bounded model checking. In the first step, the sequential behaviour of a transition system over a finite interval is encoded as a propositional formula. In the second step, that formula is given to a propositional decision procedure, i.e., a satisfiability solver, to either obtain a satisfying assignment or to prove there is none. Each satisfying assignment that

is found can be decoded into a state sequence which reaches states of interest. In bounded model checking only finite length sequences are explored." Due to this limitation some properties cannot be proven, since only counter-examples of finite length can be found.

As an example of a transition system, consider the graph in Figure 1. In the paper this example will return frequently. The circles denote the possible states of the transition system, which consist of the value of a variable x that ranges from 0 to 100. The initial state is 1, denoted by the incoming arrow. The property that we want to verify is if state 99 can be reached, depicted by the circle colored in yellow. This transition system allows substraction with 1 or multiplication by 2 on the value of x such that it remains in $[0, 100]$. Obviously state 99 can be reached, but it is important to notice that there are infinitely many ways to reach it.

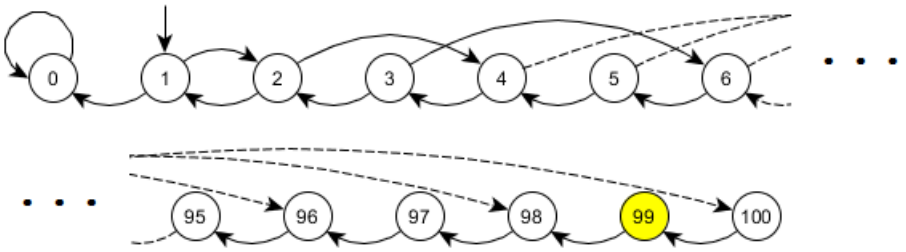

Figure 1: Example transition system

1.1 Motivation

Many different model checking tools of both academic and non-academic nature have been developed (Dutertre & De Moura, 2006; Cimatti, Clarke, Giunchiglia, & Roveri, 1999; Behrmann et al., 2006; Barnat et al., 2013). These tools usually share a common expressive power, but every tool has its own strength and weakness when dealing with instances of certain problem classes. Often it is not immediately clear, which tool one should use for a particular type of problem and it is a hassle to manually test out each tool to find out which one can provide the fastest answer, since they all have their own unique scripting languages and particularities. For example it turns out that some searches for counterexamples can be done with remarkable efficiency with bounded model checking, on designs that would be difficult for BDD based model checking and vice versa (Clarke et al., 2001). In bigger applications, such as protocol or program verification, specifying your problem instance also requires a huge amount of code to be written or generated that is prone to errors.

To make the declaration of transition systems in these tools easier, a scripting language is proposed that can express non-deterministic transition systems in

a straight-forward manner. This scripting language is interpreted by a custom developed tool that automatically generates code and includes optimizations for state-of-the-art SAT-solvers and model checking programs of various classes. These include a SMT-based satisfiability solver such as (Dutertre & De Moura, 2006), symbolic model checkers such as NuSMV (Cimatti et al., 1999) and a non-symbolic model checker called UPPAAL (Behrmann et al., 2006). The tool automatically compiles the custom scripting language to all tool-specific languages and then runs several instances these tools in parallel, allowing for quick and easy testing.

The main design principle for this tool keeps the end-user in mind, who often does not need the full expressiveness of for example the LTL formal language, but simply wants to verify a protocol or program and check safety or liveness properties or determine whether loops exists in their specification (See Section 2.1). Also the end-user should not have to worry about multiple back-ends that can be used to find the solution. The fastest solution is in most cases the desired solution and the process that leads to the result is usually of less importance. With these principles and simplifications in mind, the goal is to create a tool that minimizes the time find to solutions to specific problem instances by combing the joint effort of several model checking tools and using a new scripting language that is easy to read and intuitive to use.

2 Definitions

Before diving into the specifics of the new tool and language, some formal definitions are introduced.

Definition 1 (Transition System)
A Transition System is defined by the tuple $\mathcal{TS} = (\mathcal{S}, \mathcal{I}, \rightarrow)$*, where* $\mathcal{S}$ *is a set of states,* $\mathcal{I} \subseteq \mathcal{S}$ *and* $\rightarrow \subseteq \mathcal{S} \times \mathcal{S}$*. We call* $\mathcal{I}$ *the set of initial states and* $\rightarrow$ *the transition relation. We write* $s \rightarrow s'$ *if* $(s, s') \in \rightarrow$*. In addition we define* $Run(\mathcal{TS}) = \{(s_0, \ldots, s_n) \in \mathcal{S}^\star \mid \forall_{i=0}^{n-1} : s_i \rightarrow s_{i+1} \wedge s_0 \in \mathcal{I}\}$ *as the set of state sequences that represent finite paths through the transition system defined by* $\mathcal{TS}$*.*

Example 1 (Transition System)
Consider Figure 1 as a transition system $\mathcal{TS}$*. It is formally defined as*

$\mathcal{TS} = (\mathcal{S}, \mathcal{I}, \rightarrow)$*, where* $\mathcal{S} = \{0, \ldots, 100\}$*,* $\mathcal{I} = \{1\}$ *and*
$\rightarrow = \{(i, j) \in \mathcal{S} \times \mathcal{S} \mid : j = 2i \vee j = i - 1\}$*.*
The sequence $(1, 2, 4, 3) \in Run(\mathcal{TS})$ *is a possible run of this transition system that starts in state* 1*. This run represents a path through the transition system that corresponds with the state transitions* $1 \rightarrow 2 \rightarrow 4 \rightarrow 3$*.*

Definition 2 (Update Transition System)
Let $\mathcal{V}$ *be a set of variables. Let* $\mathbb{B} = \{true, false\}$ *and* $\mathbb{N}_{ab} = \{a \ldots b\}$ *with* $a, b \in \mathbb{N}$ *and* $b > a$*. Let* $T : \mathcal{V} \rightarrow \{\mathbb{N}_{ab}, \mathbb{B}\}$ *be the function that returns the type of a variable* $v \in \mathcal{V}$*. Let* $\mathcal{U} \subseteq \mathcal{S} \nrightarrow \mathcal{S}$ *be a set of partial functions, called update functions.*
Then an Update Transition System is defined by the tuple $\mathcal{TU} = (\mathcal{S}, \mathcal{I}, \mathcal{T}_r)$*, where* $\mathcal{S} = \prod_{v \in \mathcal{V}} T(v)$ *is the set of states,* $\mathcal{I} \subseteq \mathcal{S}$ *and* $\mathcal{T}_r = \{(s, s') \in \mathcal{S} \times \mathcal{S} \mid \exists u \in \mathcal{U} : u(s) = s'\}$*. We call* $\mathcal{I}$ *the set of initial states and* $\mathcal{T}_r$ *the transition relation.*

Example 2 (Update Transition System)
Now consider Figure 1 as an Update Transition System $\mathcal{TU}$*. It is formally defined as*

$\mathcal{TU} = (\mathcal{S}, \mathcal{I}, \mathcal{T}_r)$*, where*
$\mathcal{S} = \{0, \ldots, 100\}$*,* $\mathcal{I} = \{1\}$*,*
$\mathcal{V} = \{x\}$*,* $\mathcal{T}(x) = \{0, \ldots, 100\}$*,*
$\mathcal{T}_r = \{(s, s') \in \mathcal{S} \times \mathcal{S} \mid \exists u \in \mathcal{U} : u(s) = s'\}$ *and*
$\mathcal{U} = \{u_1, u_2\}$ *with*

$$u_1 : S \nrightarrow S,\ u_1(x) = \begin{cases} 2x & \text{if } x \leq 50 \\ undef & \text{otherwise} \end{cases}$$

$$u_2 : S \nrightarrow S,\ u_2(x) = \begin{cases} x - 1 & \text{if } x \geq 1 \\ undef & \text{otherwise} \end{cases}$$

Here, $(1, 2, 4, 3) \in Run(\mathcal{TU})$ *is a possible run of this transition system that starts in state* 1 *and ends in state* 3*. This run is a result of applying the update functions in the following order:* $u_1(1) = 2, u_1(2) = 4, u_2(4) = 3$*.*

The next definition combines several transition systems together in one system. The new state space of the combined system is the Cartesian product of the state spaces of each involved transition system. The initial state of this system is comprised of all the initial states of the individual transition systems. The combined transition relation for the new system allows only discrete updates of one of the transition systems at a time in the shared state space. The definition of the combined system expands on the previous definitions and is given below as Parallel Transition System.

Definition 3 (Parallel Transition System)
Given a set of n transition systems $\{(\mathcal{S}_1, \mathcal{I}_1, \rightarrow_1), \ldots, (\mathcal{S}_n, \mathcal{I}_n, \rightarrow_n)\}$.
Then a Parallel Transition System is defined by the tuple $\mathcal{TP} = (\mathcal{S}, \mathcal{I}, \mathcal{T})$, *where* $\mathcal{S} = \mathcal{S}_1 \times \ldots \times \mathcal{S}_n$, $\mathcal{I} \subseteq \mathcal{I}_1 \times \ldots \times \mathcal{I}_n$ *and* $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{S}$ *such that* $((s_1, \ldots, s_n), (s'_1, \ldots s'_n)) \in \mathcal{T} \Leftrightarrow \exists i : (s_i, s'_i) \in \rightarrow_i \wedge \forall_{j=1}^{n} : i \neq j \Rightarrow s_j = s'_j$.

Example 3 (Parallel Transition System)
Given a set of two transition systems that are instances of the Update Transition System defined in Example 2 $\{t_1 ::= (\mathcal{S}_1, \mathcal{I}_1, \mathcal{T}_1), t_2 ::= (\mathcal{S}_2, \mathcal{I}_2, \mathcal{T}_2)\}$, *a Parallel Transition System* $\mathcal{TP}$ *can be formally defined as*

$\mathcal{TP} = (\mathcal{S}_1 \times \mathcal{S}_2, \mathcal{I}_1 \times \mathcal{I}_2, \mathcal{T})$, *where*
$\mathcal{T} \subseteq (\mathcal{S}_1 \times \mathcal{S}_2) \times (\mathcal{S}_1 \times \mathcal{S}_2)$ *such that*
$((s_1, s_2), (s'_1, s'_2)) \in \mathcal{T} \Leftrightarrow \exists i : (s_i, s'_i) \in \rightarrow_i \wedge \forall_{j=1}^{2} : i \neq j \Rightarrow s_j = s'_j$.

Transition systems t_1 *and* t_2 *are both part of the state of the new Parallel Transition System and will be updated simultaneously during a transition. The sequence* $((1, \underline{1}), (2, \underline{1}), (4, \underline{1}), (4, \underline{2}), (4, \underline{4}), (4, \underline{3}), (3, \underline{3})) \in Run(TP)$ *is an example of a run of* $\mathcal{TP}$. *Here, the states that belong to* t_2 *are underlined to easier discriminate between the two systems.*
This run corresponds with the following state transitions of t_1 *and* t_2*:*
$t_1 : 1 \rightarrow 2 \rightarrow 4 \rightarrow 4 \rightarrow 4 \rightarrow 4 \rightarrow 3$
$t_2 : \underline{1} \rightarrow \underline{1} \rightarrow \underline{1} \rightarrow \underline{2} \rightarrow \underline{4} \rightarrow \underline{3} \rightarrow \underline{3}$.

Finally, we can define a limit on the amount of steps a Transition System can take as a whole.

Definition 4 (Bounded Transition System)
Given a transition system $(\mathcal{S}, \mathcal{I}, \mathcal{T})$ *and* $n \in \{1 \ldots \infty\}$.
Then a Bounded Transition System is defined by the tuple $\mathcal{TB} = (n, \mathbb{N} \times \mathcal{S}, \{0\} \times \mathcal{I}, \mathcal{T}_b)$, *where* $\mathcal{I} \subseteq \mathcal{S}$ *and* $\mathcal{T}_b = \{((i, s), (i+1, s')) \mid (s, s') \in \mathcal{S} \times \mathcal{S} \wedge i \in \mathbb{N} \wedge i < n\}$.
We $\mathcal{T}_b$ *the bounded transition relation.*

Example 4 (Bounded Transition System)
Now consider Figure 1 as a Bounded Transition System $\mathcal{TB}$ *with a maximum number of 100 allowed steps. It is formally defined as*

$\mathcal{TB} = (n, \mathbb{N} \times \mathcal{S}, \{0\} \times \mathcal{I}, \mathcal{T}_b)$, *where*
$n = 100$, $\mathcal{S} = \{0, \ldots, 100\}$, $\mathcal{I} = \{1\}$,
$\mathcal{V} = \{x\}$, $\mathcal{T}(x) = \{0, \ldots, 100\}$,
$\mathcal{T}_b = \{((i, s), (i+1, s')) \mid (s, s') \in \mathcal{S} \times \mathcal{S} \wedge i \in \mathbb{N} \wedge i < n \wedge \exists u \in \mathcal{U} : u(s) = s'\}$ *and*

$\mathcal{U} = \{u_1, u_2\}$ *with*

$$u_1 : S \nrightarrow S,\ u_1(x) = \begin{cases} 2x & \text{if } x \leq 50 \\ undef & \text{otherwise} \end{cases}$$

$$u_2 : S \nrightarrow S,\ u_2(x) = \begin{cases} x - 1 & \text{if } x \geq 1 \\ undef & \text{otherwise} \end{cases}$$

The sequence $((0, \underline{1}), (1, \underline{2}), (2, \underline{4}), (3, \underline{3})) \in Run(\mathcal{TB})$ *is an example of a run of* $\mathcal{TB}$*. The states are underlined in order to easily discriminate them from the step counter. This run corresponds with the state transitions* $\underline{1} \rightarrow \underline{2} \rightarrow \underline{4} \rightarrow \underline{3}$*.*

The new language allows the specification of (Bounded) Update Transition Systems and (Bounded) Parallel Transition Systems that are composed of Update Transition Systems. On these systems a model checking analysis can be performed to verify several properties that will be discussed now.

2.1 Verification

The newly developed tool offers support to verify reachability, safety and liveness properties that are expressed by Boolean formulas. Besides that there is support for deadlock and infinite run detection. These are special cases of safety properties that are often used in system verification.
Each of the five different properties will now be defined. During the definition we assume a given transition system $\mathcal{TS} = (\mathcal{S}, \mathcal{I}, \rightarrow)$. Furthermore let $\mathcal{P} \subseteq \mathcal{S}$ be a set of desired states.

2.1.1 Reachability

During the verification of a reachability property it is asked whether a run exists of which the final state is in $\mathcal{P}$.

$$Reach(P) \Leftrightarrow \exists (s_0, \ldots, s_n) \in Run(\mathcal{TS}) : s_n \in \mathcal{P} \qquad \text{(Reachability)}$$

The verification result will be negative if no such run exists. Otherwise a run is returned that leads to the final state that is in $\mathcal{P}$.

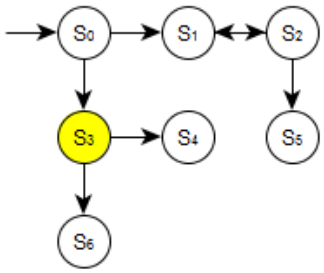

Figure 2: Reachability property is satisfied by $s_0 \rightarrow s_3$

2.1.2 Infinite run

An infinite run exists if there exists a run that can always be extended. A cycle or loop exists whenever a run exists with two states that are equivalent to each other. Such a run can be extended infinitely many times by repeating the state sequence after the first occurrence of the duplicate state. In a finite state space an infinite run can only exist due to a cycle in the transition system. Therefore the following definition of a cyclic run is equivalent to an infinite run in the finite state space:

$$\exists(s_0, \ldots, s_n) \in \mathit{Run}(\mathcal{TS}) : [\exists i < n : s_i = s_n] \qquad \text{(Cyclic run)}$$

If a cycle exists, the tool returns a run starting from the initial state s_0, that demonstrates the cycle. The last state of this run will be equivalent to another state in this run.

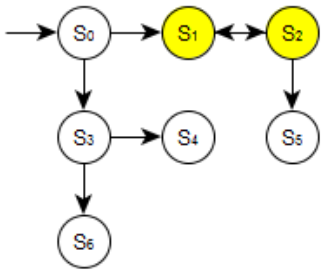

Figure 3: Infinite run exists due to $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_1$

2.1.3 Safety

The verification of a safety property involves assuring that a set of undesirable states is not reachable, or equivalently, that a set of desirable states cannot

be escaped from (Biere, Cimatti, Clarke, Strichman, & Zhu, 2003; Alpern & Schneider, 1985). This means that each state in every possible run must be in $\mathcal{P}$.

$$Safe(P) \Leftrightarrow \forall(s_0, \ldots, s_n) \in Run(\mathcal{TS}) : s_n \in \mathcal{P} \qquad \text{(Safety)}$$

This definition implies that a state that is not in $\mathcal{P}$ may not be reachable and is therefore equivalent to $\neg Reach(\neg\mathcal{P})$.
A simple example of a safety property is an invariant, a property that must hold in all reachable states. If a sequence of states can be found in which the property does not hold, then the invariant is false. When verifying a safety property the tool will check for every state if it is in $\mathcal{P}$. It will return a run that leads to a state that is not in $\mathcal{P}$ if it exists. Otherwise it will return a positive result.

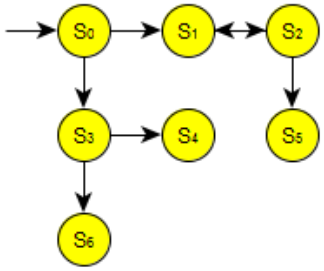

Figure 4: Satisfied safety property

2.1.4 Liveness

The verification of a liveness property assures that something good always keeps happening. In other words, a collection of desired states is always reachable from any run. There exist many different variants of liveness definitions (Clarke et al., 2001; Alpern & Schneider, 1985). The definition given below is the most general unrestrictive definition of liveness.

$$Live(P) \Leftrightarrow \forall(s_0, \ldots, s_n) \in Run(\mathcal{TS}) : \qquad \text{(Liveness)}$$
$$s_n \notin \mathcal{P} \Rightarrow [\exists(s_0, \ldots, s_n, \ldots, s_m) \in Run(\mathcal{TS}) : s_m \in \mathcal{P}]$$

In practice liveness properties cannot be verified without checking for cycles and thus infinite runs. An infinite run is live if there is at least one state in its cycle that is in $\mathcal{P}$. The liveness property is verified if the last state of all other maximum length runs that do not contain cycles is also in $\mathcal{P}$. Liveness could therefore be expressed as a combination of a special case of a reachability and infinite run property.
The tool will return a run that represents a path where a desired state could

never be reached if such a run exists. Otherwise it will return a positive result.

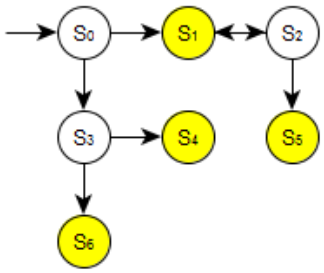

Figure 5: Satisfied liveness property

2.1.5 Deadlock

A deadlock exists whenever a transition system contains a state where no update rules can be applied.

$$\exists(s_0, \ldots, s_n) \in Run(\mathcal{TS}) : \neg[\exists(s_0, \ldots, s_n, s_{n+1}) \in Run(\mathcal{TS})] \quad \text{(Deadlock)}$$

The deadlock check could also be expressed as a reachability property for a certain $\mathcal{P}$ that encapsulates the states where no update rules can be applied. The tool will return a run that leads to a deadlock state if it exists. Otherwise it will return a negative result.

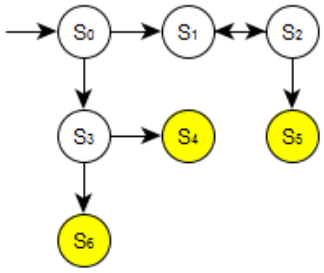

Figure 6: Deadlock exists. A possible path is $s_0 \rightarrow s_3 \rightarrow s_6$

3 Language

The scripting language allows to define instances of (Bounded) Update Transition Systems ((B)UTS) and (Bounded) Parallel Transition Systems ((B)PTS) that are composed of Update Transition Systems. In this section the core language syntax is explained using the transition system of Figure 1 as an example. The complete syntax of the language is given in Appendix I.

Each script is an instance of a *Formula*:

⟨*Formula*⟩ ::= ⟨*TransitionSystem*⟩ | ⟨*ProcessModel*⟩

First we will begin explaining how a UTS is modelled. In section 3.2 the modelling of a PTS is explained.

3.1 Transition system

The *TransitionSystem* grammar element corresponds with a (B)UTS. It consists of a few simple blocks that cover each component of the transition system. This has the following form:

⟨*TransitionSystem*⟩ ::= [‘`MAXSTEPS:`’ ⟨*Constant*⟩]
‘`VARS:`’ ⟨*VarDeclTS*⟩$^{+}$
[‘`INIT:`’ ⟨*VarInit*⟩*]
‘`TRANS:`’ ⟨*Stmt*⟩$^{+}$
⟨*Goal*⟩

⟨*Goal*⟩ ::= ‘`REACH:`’ ⟨*BoolExp*⟩
| ‘`INF:`’
| ‘`SAFE:`’ ⟨*BoolExp*⟩
| ‘`LIVENESS:`’ ⟨*BoolExp*⟩
| ‘`DEADLOCK:`’

The elements enclosed in square brackets are optional. The last element of the *TransitionSystem* is a choice between one of the verifyable properties that were discussed in section 2.1. They will be closely examined in section 3.1.5 The usage of each element will now be discussed, with a step by step example implementation of the transition system shown in Figure 1.

3.1.1 The MAXSTEPS block

The `MAXSTEPS` block corresponds with n, which defines an upper bound on the amount of steps that can be taken during the exploration of the state space of the transition system. This is an optional block, but when used, it can cause a significant increase in performance since it limits the search depth of the transition system, thus forcing the tools to alternative paths. However if the maximum sequence length is too short it may prevent a state with a desired

property from being detected, even though the transition system allows it to be reached. If the `MAXSTEPS` block is defined, n is translated to a `steps` variable internally, which may be read from, but not updated.

In the transition system of Figure 1 we may expect the solution to be found in less than 100 steps, so we can provide this as a limit in the `MAXSTEPS` block:

```
MAXSTEPS:
  100
```

3.1.2 The VARS block

The `VARS` block corresponds with the set $\mathcal{V}$, which defines all the variables that are going to be used in the transition system. A variable declaration in a transition system has the following form:

⟨*VarDeclTS*⟩ ::= '`const`' ⟨*Type*⟩ ⟨*ID*⟩ '`=`' ⟨*Constant*⟩ '`;`'
| ['`nondet`'] ⟨*Type*⟩ ⟨*ID*⟩ ['`:`' ⟨*Range*⟩] '`;`'

Variables can be of type `int` and a type `bool`, which correspond to commonly known integer and Boolean types. Some tools require integer variables to be ranged. If a range is omitted when using such a tool, the default range of [-1023, 1024] is used. Variables in this block can also be declared `const`, which means they need to be given a value immediately and cannot be reassigned or initialized elsewhere anymore. Finally a variable can also be declared `nondet`, which means that it can take any value during a transition if it is not updated in the corresponding transition rule.
The variable declaration block requires that there must be at least one variable defined. There are some variables that may not be declared, because they are used internally. The compiler will throw an error message if a restricted *ID* is used.

The transition system of Figure 1 has one variable x, that is updated during some transitions and must retain the same value otherwise. We declare it as follows:

```
VARS:
  int x;
```

3.1.3 The INIT block

The `INIT` block corresponds to s_0, which defines the initial state, i.e. the initial values of the variables. A variable initialization block in a transition system has the following form:

⟨*VarInit*⟩ ::= ⟨*BoolExp*⟩ ';' ⟨*VarInit*⟩
| ⟨*BoolExp*⟩ ';'

| ⟨*ID*⟩ '`=`' ⟨*Exp*⟩ '`;`' ⟨*VarInit*⟩
| ⟨*ID*⟩ '`=`' ⟨*Exp*⟩ '`;`'

⟨*Exp*⟩ ::= ⟨*BoolExp*⟩ | ⟨*NumExp*⟩

Variable initialization restrictions come either in the form of variable assignments or Boolean expressions over variables or both, according to one's preference. It is allowed to spread these restrictions over multiple statements to improve readability. The variable initialization block is optional. Variables that are not used in the initialization block may have any value in the initial state. In the transition system of Figure 1 we want variable x to have value 1 in the initial state. We can express that as the following Boolean formula:

```
INIT:
  x == 1;
```

Alternatively, it can be expressed as an initialization statement:

```
INIT:
  x = 1;
```

3.1.4 The TRANS block

The `TRANS` block specifies a set of variable update rules that correspond with the update functions of a (B)UTS. In the grammar such a rule is called a *Stmt*:

⟨*Stmt*⟩ ::= ['`?`' ⟨*BoolExp*⟩ '`->`'] ⟨*IDEnum*⟩$^+$ '`=`' ⟨*ExpEnum*⟩$^+$

⟨*IDEnum*⟩ ::= ⟨*ID*⟩ | ⟨*ID*⟩ '`,`' ⟨*IDEnum*⟩

⟨*ExpEnum*⟩ ::= ⟨*Exp*⟩ | ⟨*Exp*⟩ '`,`' ⟨*ExpEnum*⟩

An update rule consists of an ordered list of all variables that need to be updated followed by another ordered list that specifies their new respective values. The context-free grammar cannot specify that these lists should be equally long, however this will be enforced by the compiler via a semantics check. Optionally, an update rule can be preceded by a guard. A guard is a Boolean expression that imposes a restriction on whether or not the update rule may be applied. If variable x is not part of an update rule and the rule is applied, then the value of x in the next state is equal to its value in the current state, unless x was declared `nondet`. In the latter case, x may have any value after the update rule is applied.

The usage of the `nondet` keyword has important consequences for the execution of the update rules. If no variables are declared `nondet`, mutual exclusion of update rules is guarenteed. This means that only one update rule can be applied per step. If some variables are declared `nondet`, mutual exclusion may **not** always be guarenteed. This means that more than one update rule can be

applied in one step or parts of one rule and other rules may be applied in one step. Consider the following script:

```
MAXSTEPS:
  1
VARS:
  nondet int truck1;
  nondet int truck2;
  nondet int amount;
INIT:
  truck1 = 100;
  truck2 = 100;
TRANS:
  truck1 = truck1 - amount; //Rule 1
  truck2 = truck2 - amount; //Rule 2
REACH:
  truck1 == 10 & truck2 == 10
```

The script consist of three `nondet` declared variables. The variables `truck1` and `truck2` start with the value 100. The transition system is allowed to do one update operation. The update rules substract any value from `truck1` and `truck2`. In symbolic model checkers the update rules are translated as Boolean expressions. If `truck1` and `truck2` were not declared `nondet` the update rules would be translated as follows:

```
(truck1 == truck1 - amount && truck2 == truck2) ||
(truck2 == truck2 - amount && truck1 == truck1)
```

This translation guarentees that unused variables keep the same value during an update. The result of this is that the rules are mutual exclusive and the final state would never be reachable, since there are at least two steps needed. However in the script `truck1` and `truck2` were declared `nondet`, so the update rules would be translated as

```
truck1 == truck1 - amount || truck2 == truck2 - amount
```

Because the value of `truck1` and `amount` is not explicitly set in Rule 2 and the value of `truck2` and `amount` is not explicitly set in Rule 1, the result may be that the rules are applied simultaneously. Therefore the reachability property can be satisfied in a single step. To avoid this behaviour it is forbidden to update variables that are declared `nondet`.

In model checkers like UPPAAL this behaviour would be non-existant since the update rules are translated as seperate transitions to different states. However the performance with non-deterministic variables is very bad in non-symbolic model checkers. For these reasons `nondet` variables are not supported in those tools. An extensive explanation is given in section 4.3.

Let us extract the update rules from the example transition system of Figure 1. It looks quite complicated, but it is actually very simple. Each state allows two

basic transitions: either x is decreased by 1 or x is multiplied by two. This can be directly translated as follows:

```
TRANS:
  x = x-1;
  x = x*2;
```

We could also interpret transition systems like the one shown in Figure 1 as a finite state machine (FSM). If one would implement a script respresenting an FSM, it is worth noting that there is no explicit notion of states in the scripting language. If you would like to directly translate an FSM from paper to this language, then you can introduce a state variable that keeps track of which state the machine currently is in. In the guard of an update rule, the value of the state variable is checked, such that only update rules that belong to outgoing edges of that particular state can be applied. In the update rule, the state variable must also be updated appropriately. Update rules without a guard correspond to a transition that is possible in each state as seen above. A more complex example of a system where a state variable is used is given in section 6.1.1.

3.1.5 The verification property block

The last block of a script corresponds with any verifyable property discussed in section 2.1. In the script this has the following form:

⟨*Goal*⟩ ::= '`REACH:`' ⟨*BoolExp*⟩
| '`INF:`'
| '`DEADLOCK:`'
| '`SAFE:`' ⟨*BoolExp*⟩
| '`LIVENESS:`' ⟨*BoolExp*⟩

Either a reachability property, the existance of loops, the existance of deadlock, an safety property or a liveness property can be verified, specified by a `REACH`, `INF`, `DEADLOCK`, `SAFE` or `LIVENESS` block respectively. The `INF` and `DEADLOCK` block take no parameters. The other blocks must always be parameterized with a Boolean expression that specifies what conditions should hold in the desired states.

In the transition system of Figure 1 we want to verify that a state is reachable in which x equals 100:

```
REACH:
  x == 100
```

If we want to get a sequence of transitions that represents a cycle in the transition graph it would be specified as follows:

```
INF:
```

If we want to verify that x is always bigger than 0, we would specify it as follows:

```
SAFETY:
  x > 0
```

If we want to verify that there are always update rules applicable, we would specify it as follows:

```
DEADLOCK:
```

If we want to verify that x always eventually reaches the value 99, we would specify it as follows:

```
LIVENESS:
  x == 99
```

The complete implementation of the transition system of Figure 1 is given in Appendix A. More examples of transition systems are given in Section 6.

3.2 Process model

The *ProcessModel* grammar element corresponds with a (B)PTS of which each transition system it is composed of is a UTS. It consists of a few blocks for the global model definition, followed by a sequence of process declarations. The process model covers each component of a PTS and is defined as follows:

⟨*ProcessModel*⟩ ::= [‘`MAXSTEPS:`’ ⟨*Constant*⟩]
‘`VARS:`’ ⟨*VarDeclPM*⟩$^+$
[‘`INIT:`’ ⟨*VarInit*⟩*]
‘`PROGRAM:`’ ⟨*ProcExp*⟩
⟨*Goal*⟩
⟨*Process*⟩$^+$

A process represents an element of the set of n transition systems of which a PTS is composed. It consists of a few simple blocks of the following form:

⟨*Process*⟩ ::= ‘`PROCESS`’ ⟨*ID*⟩ ‘`:`’
‘`VARS:`’ ⟨*VarDeclTS*⟩$^+$
[‘`INIT:`’ ⟨*VarInit*⟩*]
‘`TRANS:`’ $Stmt^+$

The elements enclosed in square brackets are optional. The process model script contains a `PROGRAM` block that defines through a subset of process algebra which processes should run in parallel. Furthermore a sequence of processes is defined through multiple `PROCESS` blocks. Processes can be thought of as several transition systems running in parallel, but they are not truly concurrent. Only a single transition of a single process can be executed during each step. For example if process A is allowed to perform an update and process B is allowed to perform an update, then in the next step either one of the two transitions will

be executed. Simultaneous transitions of multiple processes are not possible, thus a process update rule can be seen as an atomic operation in the execution chain.

The meaning of each block will now be discussed, with a step by step example implementation of the transition system shown in Figure 1.

3.2.1 The MAXSTEPS block

The `MAXSTEPS` block corresponds with m. It is defined once, posing an upper bound on the amount of steps that can be taken for the complete process model, rather than an individual process. This means that if there are two processes A and B specified to run in parallel and `MAXSTEPS` is set to 2, that either process A or B completes two steps or both processes complete one step. The implementation of the `MAXSTEPS` block for the example transition system as *ProcessModel* does not change:

```
MAXSTEPS:
  100
```

3.2.2 The VARS block

The `VARS` block corresponds with a set of global variables that can be read/write accessed from any process. Global constants or non-deterministic variables can also be declared here. A global variable declaration in a process model has the following form:

⟨*VarDeclPM*⟩ ::= '`const`' ⟨*Type*⟩ ⟨*ID*⟩ '`=`' ⟨*Constant*⟩ '`;`'
| ['`nondet`'] ⟨*Type*⟩ ⟨*ID*⟩ ['`:`' ⟨*Range*⟩] '`;`'
| '`proc`' ⟨*ID*⟩ '`=`' ⟨*ID*⟩ '`;`'

The important difference with the *TransitionSystem* is that it is now also required to define one or more variables of type `proc`. These will hold an instance of a later to be defined process and must be instantiated immediately in a similar fashion as constants.

We try to model the transition system of Figure 1 as a process model now. This means that variable x will be local to a process. Thus in the global variable block we only need to declare an instance of the example process, which will later be defined in Section 3.2.6. The new `VARS` declaration is as follows:

```
VARS:
  proc P = example;
```

3.2.3 The INIT block

In the *ProcessModel* script the global `INIT` block can pose restrictions on the initial values of the global variables. In the example transition system of Figure 1 there are no global variables to initialize, so the `INIT` block can be omitted.

3.2.4 The PROGRAM block

The `PROGRAM` block states a process algebraic expression that defines which processes should be run in parallel. A process algebraic expression has the following form:

⟨*ProcExp*⟩ ::= ⟨*ID*⟩ ⟨*ProcOp*⟩ ⟨*ProcExp*⟩ | ⟨*ID*⟩

⟨*ProcOp*⟩ ::= '`||`'

Currently only one operator is defined to combine processes running in parallel. This may be extended in the future to support full process algebra. Suggestions on how this may be done are given in the Discussion in section 8.
The transition system of Figure 1 is run as a single process of which the implementation is given later in the script:

```
PROGRAM:
  P;
```

3.2.5 The verification property block

The *Goal* grammar element corresponds with a verifyable property similar to its *TransitionSystem* variant. The property may reference global variables as well as process variables from any process instance. A variable `x` local to a process `p` is accessed through an expression of the form `p.x`. In the *ProcessModel* script for the transition system in Figure 1 our property now refers to local process variable x:

```
REACH:
  P.x == 100;
```

3.2.6 The PROCESS block

A `PROCESS` block corresponds to an element of the n transition systems of a BPTS. A `PROCESS` block is given a name, such that it can be referenced during global initialization.

⟨*Process*⟩ ::= '`PROCESS`' ⟨*ID*⟩ '`:`'
'`VARS:`' ⟨*VarDeclTS*⟩$^+$

['INIT:' $\langle VarInit \rangle^*$]
'TRANS:' $Stmt^+$

The variables declared within the `VARS` block are local to the process and cannot have the same name as any previously declared global variables. Only the local variables can be initialized in the corresponding local `INIT` block. The `TRANS` block follows exactly the same rules as in a *TransitionSystem* script. In here, the updating of local and global variables is allowed. The transition system of Figure 1 as a process now looks like this:

```
PROCESS example:
  VARS:
    int x;
  INIT:
    x = 1;
  TRANS:
    x = x-1;
    x = x*2;
```

The complete implementation of the transition system of Figure 1 as a *ProcessModel* is given in Appendix B. More examples of process models are given in Section 6.

4 Implementation

The scripting language can be translated to several back-ends. Currently translations to languages used by Yices, NuSMV and UPPAAL are supported, but the tool is set up flexibly so that it is easy to extend to other languages in the future. Different tools have different particularities and in some of them limitations are unavoidable. This section discusses the different back-ends, the implementation for each one of them and limitations and optimizations amongst different back-ends.

4.1 Back-ends

Yices
Yices is an SMT-solver that can process input written in the SMT-Lib notation (Barrett, Stump, & Tinelli, 2010). This makes it possible to perform Bounded Model Checking (BMC) that uses satisfiability solving (SAT-solving) techniques, specifically Satisfiable Modulo Theories (SMT) to formally verify these models. SAT-solving is the problem of deciding whether it is possible for a given propositional Boolean formula ϕ to evaluate to *true*. If so, an instance can be given that demonstrates this result, otherwise the formula is unsatisfiable. Satisfiable Modulo Theories extend on standard SAT-solving techniques by allowing linear inequations of the following form to be taken into the formula

$$\sum_{i=1}^{n} a_i x_i \leq c$$

Here, every x_i is a variable and a_i and c are arbitrary given values. They must all be in either $\mathbb{R}$ or $\mathbb{N}$. With this addition the basic building blocks of propositional formula ϕ may be Boolean variables and linear inequalities.
The translation of the transition system that is modelled by the new script language to a model that can be checked by Yices is done by unfolding the transition system into a conjunction of n Boolean formulas that each describe a single transition of one particular state to another. For each variable x, n copies are made such that the value of x_i is the value of x after i steps. For each step the transition relation is copied n times to update the variables of step i. This will be explained in detail in section 4.2.1.
Because of this method the depth of the search tree is limited, since the maximum amount of discrete transitions that can be taken is bounded by n. Due to this technique bounded model checkers have the disadvantage of not being able to prove the absence of errors, however if an error exists it can be found by increasing the maximum number of steps until the presence of the error can be shown. Increasing the maximum number of steps can cause the generated formula to grow very large in big transition systems with many steps. This often brings along an even bigger decrease in performance, even when a resulting trace can be found by means of a trivial case. However, formulas of many

megabytes long may still be solved efficiently.

NuSMV
NuSMV is a symbolic model checker that has its own SMV scripting language, that in many cases resembles the syntax of the new scripting language. Its solving algorithm is based on a combination of a tableau constructor for the LTL formula with standard CTL model checking. NuSMV allows writing down transition rules that specify how the value of variables changes to the next state. The search space of this model checker is not bounded by a maximum number of steps, but only by the worst case exponential amount of memory and time that is needed to solve problems. Numeric variables that are declared must be bounded to a range. Internally NuSMV uses BDDs to represent numbers, which grow exponentially in size depending on how many bits it takes to represent a number. Therefore, problems with variables that have a very wide range become quickly intractable, even for trivial cases.

UPPAAL
Finally there is the non-symbolic model checker UPPAAL. It is designed as a tool to verify systems that can be modelled as networks of timed automata, but it can also handle simpler transition systems that can be specified in our new scripting language. Non-symbolic model checkers such as UPPAAL tend to run out of memory very fast, because an array of visited states is stored in memory. UPPAAL enumerates all possible follow-up states for every transition, which means that non-deterministic variables quickly cause a state space explosion. This is a problem on Windows since for this platform there is only a 32 bit version of the software available. However in many cases and especially trivial cases UPPAAL returns solutions quickly.

4.2 Code generation

First the code generation for transition systems scripts as described in section 3.1 will be explained. Then the code generation for process models as defined in section 3.2 follows. Example 5 will be used to explain how different parts of the syntax are translated. The example is not an executable script and has no intended meaning, it is merely a collection of a part of the possible syntax to show how each individual element is translated.

Example 5 (Transition System Syntax)
```
MAXSTEPS:
  30
VARS:
  bool bVar;
  int iVar : 0..10;
INIT:
 iVar = 1;
```

```
 bVar = true;
TRANS:
 iVar,bVar = iVar + 1, !bVar; //rule 1
 ? bVar == true -> iVar = iVar - 2; //rule 2

REACH:
 iVar == 10 & bVar //Reachability property

SAFE:
 iVar == 10 & bVar //Safety property

LIVENESS:
 iVar == 10 & bVar //Liveness property

INF: //Loop checking

DEADLOCK: //Deadlock checking
```

An Abstract Syntax Tree (AST) based on the formal grammar defined in Appendix I is built internally when the script is parsed. Depending on the selected back-end the AST might undergo several transformations. A transformation that every AST undergoes is simple constant propagation. In this transformation anywhere a constant variable is used it is replaced with its concrete value. Then the constant declarations and initialisations are completely removed from the AST. This is an optimization that benefits the performance of SAT-based solvers and also makes an impact on memory usage, since there are less variables to consider. Other transformations like adding a steps variable, adding extra conditions or rewriting update rules might occur depending on the selected back-end and script type. These transformations will be discussed later when they are encountered in a specific implementation process.

4.2.1 Transition system

Yices
First each variable declaration is translated to a function that returns a value that has the type of the variable. Furthermore this function has one integer parameter that denotes the current step. The variable ranges will be dealt with later. The example `VARS` block is translated as follows:

```
:extrafuns ((bVar Int bool) (iVar Int Int))
```

This translation allows us to know the value of variables in each step. An expression like `(iVar 0)` would return the value of `iVar` in step 0.

Then, the rest of the translation to Yices basically comes down to a big conjunction of Boolean formulas. Each part of the script is part of that conjunction. Its general form is as follows:

```
:formula
(and
;For each step, put parts of the script as Boolean formulas here
)
```

The transition system is unfolded into $n + 1$ discrete steps, where n is defined in the `MAXSTEPS` block. The first step is an initialization step. In subsequent steps the update rules are applied. The upper bound n must always be defined. If the `MAXSTEPS` block is omitted in a script a default value is used to determine the number of unfoldings.

Each variable initialization is translated to a Boolean equality formula that is specified in the initialization step 0 and added to the general conjunction. Any range constraints on variables are also directly translated for the first step. The example `INIT` block would be translated as follows:

```
;Initial values for iVar and bVar:
(= (iVar 0) 1)
(= (bVar 0) true)
;Range constraint for iVar:
(>= (iVar 0) 0)
(<= (iVar 0) 10)
```

Then the update rules are translated for n steps. A single step consists of the disjunction of all update rules. A single update rule is the conjunction of its guard and the variable updates. In an internal transformation of the AST the guard is extended with conditions that assure that after the update all variables are still in their proper range. Then the guards, which are simple Boolean expressions, can be directly translated per transition. The variable updates can be translated as Boolean equalities each specifying the value a variable should have in the next step. Variables that were not in the update rule are also included as Boolean equalities that state that the variables should remain equal to themselves in the next step.
Due to this translation all the rules are mutually exclusive and thus can be placed together in a disjunction. When a variable was declared `nondet` and not updated, it is left out of the equation. This can compromise the mutual exclusivity of the rules if it is assigned a value as was described in section 3.1.4. Therefore it is forbidden to assign a value to a variable that is declared `nondet`. Finally we must always include an identity rule that makes no changes. This rule must be added to cover the case where no transition rules are applicable. Without this rule the formula can become unsatisfiable just due to a lack of applicable update rules, which could unjustifiably influence the result of the property that is verified. An update rule that has no guard and no updates of ranged variables can always be applied. If such an update rule exists the identity rule need not be compiled. The transition rules of the example are translated as follows for the first step:

```
;Transition step 0
```

```
(or
;rule 1
(and
;generated range constraint:
(and (<= (+ (iVar 0) 1) 10) (>= (+ (iVar 0) 1) 0))
(= (iVar 1) (+ (iVar 0) 1))
(= (bVar 1) (not (bVar 0)))
)
(and
;rule 2
(and (= (bVar 0) true) ;guard
;generated range constraint:
(and (<= (- (iVar 0) 2) 10) (>= (- (iVar 0) 2) 0)))
(= (iVar 1) (- (iVar 0) 2)) ;update
(= (bVar 1) (bVar 0)) ;unupdated variable
)
;identity rule:
(and
(not (or
;rule 2 not applicable?
(and (= (bVar 0) true)
(and (<= (- (iVar 0) 2) 10) (>= (- (iVar 0) 2) 0)))
;rule 1 not applicable?
(and (<= (+ (iVar 0) 1) 10) (>= (+ (iVar 0) 1) 0))))
;do nothing
(= (iVar 1) (iVar 0))
(= (bVar 1) (bVar 0))
)
)
```

Finally the verifyable property has to be translated. Depending on the property, the entire formula must be either satisfiable or unsatisfiable. In case of a reachability problem the final state can be directly translated as a disjunction of $n + 1$ times the same Boolean expression for each step. If the formula is satisfiable an instance is returned that shows all the steps needed to reach the goal state. The example reachability property is translated as follows:

```
(or
(and (= (iVar 0) 10) (bVar 0))
(and (= (iVar 1) 10) (bVar 1))
...
(and (= (iVar 30) 10) (bVar 30))
)
```

During loop checking we verify if two separate states exist in which each variable is equal in both states. SMT-Lib allows for universal and existential quantifiers, that realise a very short-hand notation of this property. However running in-

stances with quantifiers often leads to significantly worse performance than unfolding the whole statement. Therefore loop checks are unfolded by comparing every state with its future states resulting in a still manageable $O(k^2)$ increase in file size, while providing a significant speed increase. This method usually leads to the fastest results. However if the file size becomes too big there is also another way to encode this property without making use of quantifiers. In the generated script we can define a variable i of type int and a variable j of type int. We can now add a simple and-clause that specifies that there must be some state i which occurred before state j in which all the variables are equal, like so:

```
(and
  (< i j)
  (<= i 30) ;i <= MAXSTEPS
  (<= j 30) ;j <= MAXSTEPS
  (> i 0)
  (> i 0)
  ;state i equals state j
  (= (iVar i) (iVar j))
  (= (bVar i) (bVar j))
)
```

If the formula is satisfiable an instance is returned that shows all the steps needed to create a loop. An important thing to note here is that the identity rule could cause a loop to be detected when no transitions are possible anymore. Therefore a Boolean `loop` variable is introduced. It is initialized to `true` and not altered in any transition except the identity rule. In the identity rule the loop variable is set to `false`. Finally as an addition to the loop property above, we demand that the loop variable must be true in every step. This ensures that no loops are possible due to the identity rule:

```
;Loop conditions:
(= (loop 0) true)
(= (loop 1) true)
...
(= (loop 30) true)
```

With these conditions, paths that exit from a loop and end up in a deadlock state are excluded. To improve performance we can relax the loop condition such that `loop` only has to be true until the end of the loop. After all we are only interested in the loop detection and not in the rest of the execution. Relaxing the loop condition also makes more satisfying assignments available, which could increase solving performance. With the following condition there are more possible satisfying assignments, but we can still be sure that a returned path contains a loop.

```
(= (loop j) true))
```

Checking for a safety property specified by the Boolean expression p is the same as checking that $\neg p$ is never reachable. Therefore when we insert the reachability property $\neg p$. The formula is unsatisfiable iff the invariant holds. If the formula is satisfiable it will return a counter-example showing that $\neg p$ is reachable. The example safety property is translated as follows:

```
(or
(not (and (= (iVar 0) 10) (bVar 0)))
(not (and (= (iVar 1) 10) (bVar 1)))
...
(not (and (= (iVar 30) 10) (bVar 30)))
)
```

If none of the transition rules applies the system is in a deadlock. This means that if the identity rule can be applied, the system is in deadlock. To detect a deadlock we check in every step if the guard of the identity rule applies. If the formula is satisfiable a deadlock exists and a trace leading to the deadlock situation is returned. If the formula is not satisfiable the system is deadlock free. In case there is always a transition possible (i.e. there is no identity rule) a deadlock can never occur. In that case we insert `false` as a property such that the tool returns the right answer.
The deadlock check for a system consisting of just rule 1 would be translated as follows:

```
(or
(not (and (<= (+ (iVar 0) 1) 10) (>= (+ (iVar 0) 1) 0)))
(not (and (<= (+ (iVar 1) 1) 10) (>= (+ (iVar 1) 1) 0)))
...
(not (and (<= (+ (iVar 30) 1) 10) (>= (+ (iVar 30) 1) 0)))
)
```

Finally we may check for a liveness property. The liveness property states that always eventually a state where the Boolean expression p holds must be reached. Since we are working with a limited number of possible steps there is a violation of the liveness property if either of the two following conditions holds:

- a loop exists wherein property p never holds
- another path exists that is not a loop and ends in a state that does not satisfy p.

The example liveness property is translated into two parts:

```
;;Liveness property
(or
;;Detect a loop that is not live
...
;;Detect a path that is not live and has no loops
...
)
```

The first condition uses the same code as used for infinite path checking in conjunction with a check that verifies if all states inside the loop do not satisfy p:

```
;;First condition
(and
(and
;;Insert Inf Path checking conditions here
...
;;Check if all states in the loop are invalid:
(and
(or
(< 0 loopi)
(> 0 loopj)
(not (and (>= (iVar 0) 10) (bVar 0)))
)
(or
(< 1 loopi)
(> 1 loopj)
(not (and (>= (iVar 1) 10) (bVar 1)))
)
...
(or
(< 30 loopi)
(> 30 loopj)
(not (and (>= (iVar 30) 10) (bVar 30)))
)
))
```

The second condition checks if there is a path that ends in a state that does not satisfy p. This final state should be in a path that is not a loop. So in addition it is checked that for every possible section of a path where the start state is equal to the end state that there is no state that satisfies p.

```
;;Detect a final invalid state that is not part of any loop
(and
(not (and (>= (iVar 30) 3) (bVar 30)))

(not
;;for each i, j > i, check if it is a loop and if p holds:
(or
(and
(= (iVar 0) (iVar 1))
(= (bVar 0) (bVar 1))
(or
(and (>= (iVar 0) 3) (bVar 0))
)
```

```
)
(and
(= (iVar 0) (iVar 2))
(= (bVar 0) (bVar 2))
(or
(and (>= (iVar 0) 3) (bVar 0))
(and (>= (iVar 1) 3) (bVar 1))
)
)
...
(and
(= (iVar 29) (iVar 30))
(= (bVar 29) (bVar 30))
(or
(and (>= (iVar 29) 3) (bVar 29))
)
)
)))
```

NuSMV

In NuSMV we do not need to unfold the transition system in separate steps or worry about which step we are in if `MAXSTEPS` is not defined. Only if there is a `MAXSTEPS` declaration an extra `steps` variable is introduced that keeps track of the amount of steps taken. It is initialized to 0 and incremented after each transition.

There is some resemblence between the new scripting language and the SMV language that allows the `VARS` and `INIT` blocks to be almost directly copied. NuSMV requires all variables to have a range. Therefore it is forbidden to compile a script with unranged variables for NuSMV.
Assume a script based on `MAXSTEPS`, `VARS` and `INIT` blocks as defined in Example 5. The translation of these blocks to the SMV language would be as follows:

```
VAR
bVar : boolean;
iVar : 0..10;
steps : 0..30;
INIT
(iVar = 1) & (bVar = TRUE) & (steps = 0)
```

The update rules are translated as seperate state transitions that are connected using the | operator. This operator denotes a choice between transitions. The update rules are translated as Boolean conjunctions similar to Yices, but with different syntax. Guards can be translated as the Boolean expressions they are and variable updates are translated with a `next` statement that indicates the value the variable should have after the transition. NuSMV requires that there is always a transition possible, otherwise the model is considered incomplete.

Therefore the identity rule is inserted as was described in the previous section. This rule is inserted to assure that in every case NuSMV does not return wrong answers due to an incomplete model specification. The final translation of the `TRANS` block of the example is as follows:

```
TRANS
--rule 1:
(steps + 1 <= 30 &
iVar + 1 <= 10 & iVar + 1 >= 0 &
next(iVar) = iVar + 1 &
next(bVar) = !bVar &
next(steps) = steps + 1) |
--rule 2:
(bVar = TRUE & steps + 1 <= 30 &
iVar - 2 <= 10 & iVar - 2 >= 0 &
next(iVar) = iVar - 2 &
next(steps) = steps + 1 &
next(bVar) = bVar) |
--identity rule:
(!((bVar = TRUE &
steps + 1 <= 30 &
iVar - 2 <= 10 & iVar - 2 >= 0) |
(steps + 1 <= 30 &
iVar + 1 <= 10 & iVar + 1 >= 0)) &
next(steps) = steps &
next(iVar) = iVar &
next(bVar) = bVar)
```

The verifyable properties are translated to an LTL equivalent in the SMV language. Consider an example reachability property p. It can be translated as an LTL expression that states that finally p should hold:

```
LTLSPEC F ( p )
```

If this property holds we will not see a trace from NuSMV, because it will only return a trace if a counter example was found. To circumvent this, we can inverse the specification as follows:

```
LTLSPEC G !( p )
```

Here NuSMV will be verifying that $\neg p$ holds in every possible state, so in the case where p holds we will get back a trace, presented as counter example.

When verifying a loop checking property an extra `loop` variable is added and initialized to `TRUE`. During a transition this variable keeps its current value. If no more transitions are allowed the loop variable is put to `FALSE` in the identity rule. If the loop variable remains `TRUE` at all times this means that it is always possible to execute a transition, hence a cycle exists. When checking for loops, the usage of the steps variable is forbidden as it will always prevent loops from

occurring. These limitations regarding loop checking are explained in detail in section 4.3. The following loop checking LTL specification will return a trace of a step sequence that contains a loop:

```
LTLSPEC F (loop = FALSE)
```

A safety property p should hold in every state. If p does not hold a counter-example is returned. The safety property p is translated as follows:

```
LTLSPEC G ( p )
```

Checking for deadlock is handled in the same way as checking for loops. We simply use the `loop` variable in a different way. If the loop variable remains `TRUE` at all times, it means that it was always possible to execute a transition. Here it is also not allowed to use the `MAXSTEPS` block, since it would always cause a deadlock if the maximum number of steps is exceeded. The following LTL specification will return a counter-example if a deadlock exists. It verifies that no path exists in which no transitions are possible.

```
LTLSPEC G (loop = TRUE)
```

Finally a liveness property p can be directly translated to a simple LTL expression that defines that always eventually p must hold. A counter-example is returned if the liveness property does not hold.

```
LTLSPEC G (F (p))
```

UPPAAL

The translation from a script to a model in UPPAAL is less straight forward than with the other tools; it poses some serious limitations and requires a few workarounds to get it right as will be further explained in section 4.3.

A simple transition system will be translated to a single timed automaton in UPPAAL. The variable declarations and initializations are declared locally to the automaton. These must be constant initial values, since UPPAAL does not support non-deterministic variable initialization in the local declaration. Ranged integer variables can be translated to bouded integer variables in UPPAAL. A translation of the `VARS` and `INIT` blocks of the example would look like the following:

```
bool bVar = true;
int [0,10] iVar = 1;
```

The rest of the model consists of a timed automaton that has a start state and at least one transition state. In case there exist no update rules without guards after the internal transformation of the AST there is also a deadlock state. The start state contains outgoing arrows that lead to unique other states of which one is generated for each transition. It may also contain a transition to the deadlock state to cover the case that no transitions are possible. All the transition states lead directly back to the start state. An implementation of the automaton that is translated from the `TRANS` block from the example is given in

Figure 7. We could also omit all the transition states and create only outgoing edges from the start state to itself. However when UPPAAL returns a trace it only returns state changes. With the extra transition states we can quickly see which transitions were taken.

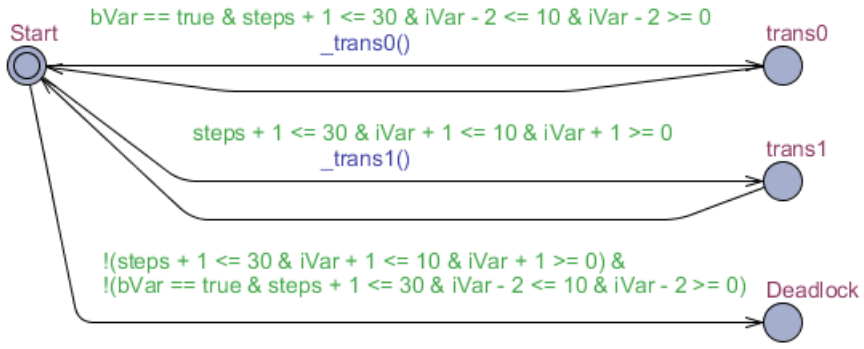

Figure 7: UPPAAL implementation

The update rules defined in the script are translated to functions that are local to the automaton. The functions are called when a transition is taken. In Figure 7 the function calls are shown in blue. Guards on the transitions can be directly translated as guards on the edges of a transition arrow. They are shown in green. The implementation of the transition functions is as follows:

```
void _trans1()
{
  int iVar_old = iVar;
  bool bVar_old = bVar;
  iVar = iVar_old + 1;
  bVar = !bVar_old;
  steps = steps + 1;
}

void _trans0()
{
  int iVar_old = iVar;
  iVar = iVar_old - 2;
  steps = steps + 1;
}
```

There is the possibility to put the update statements directly onto the edge, which is stated to be a few procent faster than using functions (Behrmann, David, & Larsen, 2004), but in our case this is very inefficient. With UPPAAL, the updating of a variable is an atomic step during a transition. The updating steps are executed in sequence. This means that variables that are updated and

then used in another statement of the transition update have already changed value. For this reason local copies must be made that hold old values of the variables, which can then be used to update each variable to their new values without being disturbed by intermediate results. However, in the update field of the edge it is not possible to create local variables, therefore every variable has to have a local copy defined in the process definition. This needlessly slows down execution time and causes out-of-memory exceptions much faster. Luckily UPPAAL supports the definition of local functions in which function-local temporary variables can be defined. In this way the state variable doubling is avoided and performance increased.

UPPAAL offers a secondary language to specify properties that need to be verified. A reachability property can be translated with an existential quantifier and a diamond operator that specifies that there exists a state that can finally be reached and satisfies property p:

```
E<> p
```

The translation for loop checking is more peculiar. In a similar way as with translations to other tools a Boolean loop variable is introduced that is set to false as soon as a transition to the deadlock state is taken. Then a loop could be checked with the `E[]` operator of which the definition is as follows: "`E[]` ϕ says that there should exist a maximal path such that ϕ is always true. A maximal path is a path that is either infinite or where the last state has no outgoing transitions." (Behrmann et al., 2004). If an infinite path exists where the loop variable is true this proves the existence of a loop:

```
E[] Process.loop
```

However since UPPAAL works with timed automata this property will never be satisfied using just the above translation. Because it is allowed to stay in the same state for a longer period of time, it is theoretically possible to stay in the same state forever, thus never satisfying the above property because the path is not infinite and the state has outgoing transitions. For this reason UPPAAL must be forced to make a transition after some time units. To accomplish this a clock variable is defined and in each state the restriction is added that it can only stay in the same state for one time unit. With this addition the loop checking is possible. An instance of a loop checking automaton that consists of just rule 1 of the example syntax is depicted in Figure 8.

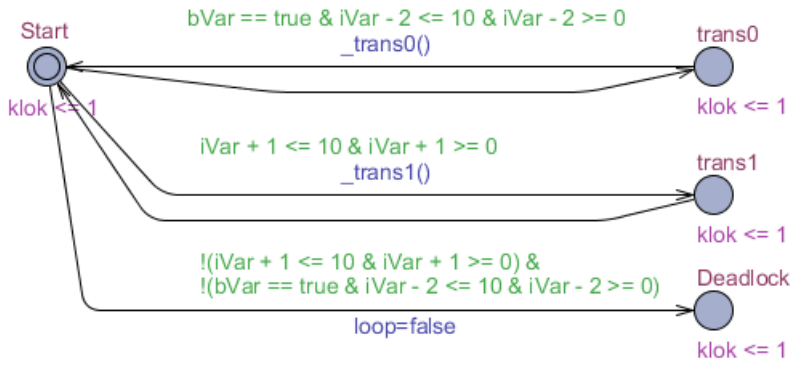

Figure 8: UPPAAL loop implementation

To the local variable declarations the clock variable is added:

```
clock klok;
```

Safety properties can be directly translated to an expression that states that `p` should be true in all reachable states:

```
A[] p
```

Checking for deadlock is as simple as checking if the deadlock state can be reached. In this case it is not allowed to use the `MAXSTEPS` block, since it would always cause a deadlock if the maximum number of steps is exceeded. The deadlock checking property would be translated as follows:

```
E<> Process.Deadlock
```

Finally checking for liveness properties can be done by using the following expression which states that p should always be satisfied eventually. Due to the same reason as with loop checking this also requires the introduction of the clock variable in order to always progress the system.

```
A<> p
```

4.2.2 Process models

Yices and NuSMV
For SMT-Lib and NuSMV the parsed abstract syntax tree of the process model script is transformed internally to its transition system equivalent. Both SMT-Lib and NuSMV undergo the same transformation. The entire process consists of a few steps:

1. The `PROGRAM` block is evaluated for processes that are being used. The original `PROCESS` declarations are removed and for each process instance

a new copy is added to the syntax tree. The process identifiers will be replaced with the name of their corresponding process variable.

2. The local variables in each process are replaced with fresh variables, i.e. variables with a name that has not been used anywhere else in the script. The replacements are applied to the `VARS`, `INIT` and `TRANS` blocks. For easy identifyability the local variables of each process are prefixed with their process identifier. If this results into name clashes, another 'fresher' name is chosen internally.

3. The global and local variables from each process are put together, as well as the `INIT` and `TRANS` statements, while the `PROGRAM` block and remaining `PROCESS` syntax and process variable declarations are removed.

What remains after the transformation is the equivalent of the process model, now expressed as a regular transition system. Take for example the parallel adding script given in Appendix C. By applying each step the transformation is as follows:

Step 1: copying processes

```
PROCESS P:
  VARS:
    int state : 0..2;
    int x : 0..256;
  INIT:
    x = 0;
    state = 0;
  TRANS:
    ? (tx == 0) -> x,state = c,1;
    ? (tx == 1) -> x,state = x+c,2;
    ? (tx == 2) -> c,state = x,0;
PROCESS Q:
  VARS:
    int state : 0..2;
    int x : 0..256;
  INIT:
    x = 0;
    state = 0;
  TRANS:
    ? (tx == 0) -> x,state = c,1;
    ? (tx == 1) -> x,state = x+c,2;
    ? (tx == 2) -> c,state = x,0;
```

Step 2: introducing fresh variables

```
PROCESS P:
  VARS:
    int P_state : 0..2;
```

```
    int P_x : 0..256;
  INIT:
    P_x = 0;
    P_state = 0;
  TRANS:
    ? (P_tx == 0) -> P_x,P_state = c,1;
    ? (P_tx == 1) -> P_x,P_state = P_x+c,2;
    ? (P_tx == 2) -> P_c,P_state = P_x,0;
PROCESS Q:
  VARS:
    int Q_state : 0..2;
    int Q_x : 0..256;
  INIT:
    Q_x = 0;
    Q_state = 0;
  TRANS:
    ? (Q_tx == 0) -> Q_x,Q_state = c,1;
    ? (Q_tx == 1) -> Q_x,Q_state = Q_x+c,2;
    ? (Q_tx == 2) -> Q_c,Q_state = Q_x,0;
```

Step 3: merging the results

```
MAXSTEPS:
  30
VARS:
  int c : 0..256;
  int P_state : 0..2;
  int P_x : 0..256;
  int Q_state : 0..2;
  int Q_x : 0..256;
INIT:
  c = 1;
  P_x = 0;
  P_state = 0;
  Q_x = 0;
  Q_state = 0;
TRANS:
  ? (P_tx == 0) -> P_x,P_state = c,1;
  ? (P_tx == 1) -> P_x,P_state = P_x+c,2;
  ? (P_tx == 2) -> P_c,P_state = P_x,0;
  ? (Q_tx == 0) -> Q_x,Q_state = c,1;
  ? (Q_tx == 1) -> Q_x,Q_state = Q_x+c,2;
  ? (Q_tx == 2) -> Q_c,Q_state = Q_x,0;
REACH:
  c == 245
```

The result is the transition system equivalent of the original script. The rest of the compilation proceeds as if it was a regular transition system, which is described in the previous section.

UPPAAL

The translation from a process model to UPPAAL undergoes a less drastic transformation and is in fact fairly direct. The `PROGRAM` block is evaluated for processes that are being used. Each used `PROCESS` block is translated to an UPPAAL automaton following the same rules as a normal transition system. Each automaton is given the unique process identifier as name.

Then, the `PROGRAM` block is translated as a system declaration. It defines which instances of automata to run in parallel. The example script defines two instances of the automaton to run in parallel:

```
p2 = p();
p1 = p();
system p2, p1;
```

The global `MAXSTEPS`, `VARS` and `INIT` blocks can be translated as global declarations:

```
int [0,256] c = 1;
int steps = 0;
```

Finally the translation of the verifyable property is translated and must now take into account all the different processes. The translation of the example reachability property may remain the same:

```
E<> (c == 165)
```

A loop checking property should detect any possible loop. A loop can exist in any running process. Therefore we check that at least one of the processes can keep its loop variable true at all times:

```
E[] p1.loop or p2.loop
```

A safety property p can still be translated as before:

```
A[] p
```

Checking for deadlock means that in no case the entire system can progress anymore. Therefore we check if we can reach a state in which all processes are in deadlock:

```
E<> p1.Deadlock and p2.Deadlock
```

Finally the translation of the liveness property may remain the same:

```
A<> p
```

4.3 Limitations

Other limitations exist besides the ones mentioned in the previous sections. The verification of several properties disallows the usage of the `MAXSTEPS` declaration in NuSMV and UPPAAL. Since a maximum number of steps is not a required part for these tools, a step variable is declared internally that is incremented by one after each transition. In the SMT-Lib translation the step counter can easily be discarded when checking for loops, but for NuSMV and UPPAAL this variable prevents finding a loop even if it exists, since the step variable cannot be excluded from the state. The verification of other properties is also hindered by the step variable. Therefore for these tools the definition of a maximum number of steps is only allowed in scripts with a reachability, safety or deadlock property.

Non-deterministic variables pose serious restrictions on the solving capability of UPPAAL as was described in the previous sections. Therefore non-deterministic variables are not supported right now when compiling to UPPAAL. This also means that all the declared variables must be initialized and they must be initialized to concrete values instead of non-deterministic Boolean restrictions. Finally due to the stateless nature of the scripting language the generated models in UPPAAL will not be the most efficient possible. The translation of the transition system without explicit states is less efficient in UPPAAL than in tools that do not need to represent states explicitly, since we cannot optimally make use of the hidden states that reside in a script. In the case of the parallel adding problem this results in an automaton that has needless checks to govern the state, which could have been translated as separate states resulting in a shorter model.
A lot of other overhead due to checks and needless variable copying also causes a severe decrease in performance. Even while the translation from a process model introduces a more 'natural' translation to UPPAAL, making use of its capabilities to define several systems running in parallel, the parallel adding problem seems to run slower using this model. While the goal was to increase the performance, instances specified as a process model took the longest to complete. Instances specified as a normal transition system could be up to two times faster, but still were not as fast as a manual optimized translation of the problem. Furthermore the generated instances run out of memory a bit sooner than the manual translations.
Although these shortcomings may prevent problems to be efficiently solved in UPPAAL, it is still possible to translate a transition system script defined in the new language to UPPAAL to make use of its model checking capabilities.

5 Usage

In order to use the newly developed tool you may download the prototype that was used to obtain the results in this paper. It is available in the git-repository that can be found online[1]. A warning in advance: the newly developed academic tool is highly experimental and considered an unstable prototype that contains unfinished features and may contain bugs.

5.1 Prerequisites

The online repository contains the full Netbeans project, a pre-compiled executable jar file and the benchmark results that are presented in this paper. In order to open the code project you need at least Netbeans 8.0[2] and the Java JDK 8[3]. Those that only wish to run the included compiled jar need the Java Runtime Environment 8[4]. Currently only 64 bit systems running Windows are supported. The tool was developed and tested on Windows 7 Professional only so it is recommended to use Windows 7. With these prerequisites you are only able to compile script files in the new language to script files in languages of other tools. To make use of the tool's automatic features that call other programs you need to download Yices 1.0.40[5], NuSMV 2.5.4[6] and UPPAAL 4.0.13[7] and follow the install instructions that are included in the git-repository. Compatibility with any other versions of these tools is not guarenteed.

5.2 Commands

The tool is intended to be used as a command-line utility. The downloadable version is able to perform the following tasks:

- Compiling scripts that are written according to the language specification of section 3 to multiple back-ends.
- Calling other model checking programs with the compiled scripts.
- Benchmarking the run time performance of other model checking programs.

Besides these functions, you might find other useful features, which are described in the included help file of the tool. To be able to perform the three tasks for

[1] `https://bitbucket.org/VultureX/unifiedmodelcheckingprototype`
[2] `https://netbeans.org/downloads/`
[3] `http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html`
[4] `http://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html`
[5] `http://yices.csl.sri.com/download-yices1.shtml`
[6] `http://nusmv.fbk.eu/NuSMV/download/getting-v2.html`
[7] `http://www.uppaal.org/`

Yices, NuSMV and UPPAAL and a custom script named "a test.script" you can use the following commands.

- `java -jar MCC.jar -compile "a test.script" -y -n -u`
- `java -jar MCC.jar -run "a test.script" -y -n -u`
- `java -jar MCC.jar -benchmark "a test.script" -y -n -u 600`

The first compiles "a test.script" to Yices, NuSMV and UPPAAL. The second compiles and runs "a test.script" for Yices, NuSMV and UPPAAL. The last benchmarks Yices, NuSMV and UPPAAL with a timeout of 600 seconds. The results are stored in a Benchmarks folder in the parent folder of each tool. A full list of commands and features is available in the download.

6 Example Problems

In this section some guidelines for specifying problems are introduced as well as a few concrete example problems. These examples are also used in the benchmarks of Section 7

6.1 Weighted graphs

With the scripting language it is possible to define a weighted graph. A weighted graph is a directed graph, of which every edge is labeled by a weight. One can easily write a script that automatically generates an instance of a graph with n states, m weighted edges, an initial state and update rules that take the weighted edges into account while traversing the state space.
Take for example the following weighted graph:

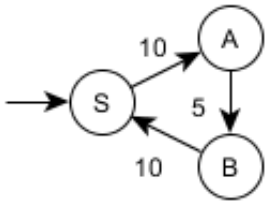

Figure 9: Example weighted graph

The states and edges can simply be represented as constant integers, where the a state constant represents a particular state and an edge constant represents the weight of a particular edge. An extra variable is introduced that contains the sum of the weighted edges that have been traversed so far. The initial state is set to S and the initial sum value is initialized to zero:

```
VARS:
  int sum;
  int currentState 0..2;
  const int S = 0;
  const int A = 1;
  const int B = 2;
  const int S_A = 10;
  const int A_B = 5;
  const int B_S = 10;
INIT:
  currentState = S;
  sum = 0;
```

Transition rules can then be easily added for each edge. The state is changed and a sum value is increased with the weight of the edge.

```
TRANS:
  currentState == S -> currentState,value = A,value+S_A;
  currentState == A -> currentState,value = B,value+A_B;
  currentState == B -> currentState,value = S,value+B_S;
```

Finally a verifyable property for the model of the weighted graph can be specified. As an example we can ask the question if there is a path starting from S of total weight 25:

```
REACH:
  sum == 25
```

A full implementation is given in Appendix D.

6.1.1 Truck delivery problem

A special variant of the weighted graph is the delivery problem. Suppose that there is a truck with a capacity of c food packages. A city with the name A also has a maximum capacity of A_c food packages that it can store. The truck can unload an amount of a food packages in each city, depending on its own current load and the maximum capacity of a city. Traveling from city to city takes time. During the travel period of the truck each city consumes an amount of food packages. If the amount of food packages in a city is 0, the city dies of starvation. The truck's starting location is at a depot, which has an infinite amount of food packages available. When the truck arrives back at the depot it will be fully reloaded.

The example weighted graph of Figure 9 can be interpreted as a truck delivery problem. Each state represents a city. Their maximum capacities may be randomly chosen. Each edge represents a traveling time and the amount of food packages that are consumed during the trip from one city to another. State S represents the depot.

In addition to the script of the regular weighted graph, the current number of food packages in each state or city (except the depot) is now represented by a variable. They are initialized with a self chosen number of food packages. Additional constants are added to define the maximum capacity of each city and the truck. The truck starts fully loaded.

```
VARS:
  int truck : 0.. TRUCK_MAX;
  int state : 0..2;
  int Ac : 0..A_MAX; //Amount of packages in A
  int Bc : 0..B_MAX; //Amount of packages in B
  const int S = 0; //depot
```

```
  const int A = 1;
  const int B = 2;
  const int S_A = 10;
  const int A_B = 5;
  const int B_S = 10;
  const int TRUCK_MAX = 20;
  const int A_MAX = 30;
  const int B_MAX = 30;
INIT:
  currentState = S;
  truckLoad = TRUCK_MAX;
  Ac = 20;
  Bc = 20;
```

A transition now decreases the amount of food packages in each state. And besides that, the amount of food packages in the truck should also be updated appropriately. The amount of food packages that the truck can unload in each state can either be deterministic or non-deterministic. In case of the deterministic problem we decide that the truck always unloads a maximum amount of packages. This involves calculating how many packages can be unloaded in the state it arrives in, while also taking into account the traveling time:

```
TRANS:
  ? state == S & Ac-S_A >= 0 & truck - (A_MAX - (Ac - S_A)) >= 0
  -> state, truck, Ac, Bc
  = A, truck - (A_MAX - (Ac - S_A)), A_MAX, Bc-S_A;
  ? state == S & A-S_A >= 0 & truck - (A_MAX - (Ac - S_A)) < 0
  -> state, truck, Ac, Bc
  = A, 0, (Ac - S_A) + truck, Bc-S_A;

  ? state == A & Bc-A_B >= 0 & truck - (B_MAX - (Bc - A_B)) >= 0
  -> state, truck, A, B
  = B, truck - (B_MAX - (Bc - A_B)), A-A_B, B_MAX;
  ? state == A & Bc-A_B >= 0 & truck - (B_MAX - (Bc - A_B)) < 0
  -> state, truck, A, B
  = B, 0, (Bc - A_B) + truck, B_MAX;

  ? state == B
  -> state, truck, Ac, Bc
  = S, TRUCK_MAX, Ac-B_S, Bc-B_S;
```

A non-deterministic transition would introduce a new non-deterministic value:

```
  nondet int amount;
```

The non-deterministic amount simplifies the calculations:

```
TRANS:
  ? state == S & Ac-S_A >= 0
```

```
  -> state, truck, A, B
  = A, truck - amount, (Ac - S_A) + amount, Bc-S_A;
  ? state == A & Bc-A_B >= 0
  -> state, truck, A, B =
    B, truck - amount, Ac-A_B, (Bc - A_B) + amount;
  ? state == B
  -> state, truck, A, B
  = S, TRUCK_MAX, Ac-B_S, Bc-B_S;
```

A property that we want to verify asks the question whether it is possible to supply each state with a constant stream of packages such that they never starve. We know that this is the case if a loop is possible:

```
INF:
```

A full example of the delivery problem with a deterministic unloading scheme is given in Appendix E. A full example of the delivery problem with a non-deterministic unloading scheme is given in Appendix F.

6.2 Sequential programs

Another important use case is the verification of programs. A program consists of a sequence of statements. The execution of a statement can be interpreted as a state transition and also implemented as one. Take for example the following pseudo-code of a program that contains a loop:

```
1. while(condition) {
2.  stmt1;
3.  stmt2;
4. }
```

A pseudo-implementation of this program in the script language would have a global variable that is the program counter. After each instruction the program counter is updated such that the correct instruction is executed. The program counter is represented by a variable `PC`. After each transition the state updates are applied and the program counter is updated appropriately:

```
VARS:
  int PC;
  ...
INIT:
  PC = 1;
  ...
TRANS:
  //while loop condition:
  ? PC == 1 & condition -> PC = PC + 1;
  ? PC == 1 & !condition -> PC = 4;
```

```
  //while loop body:
  ? PC == 2 -> ..., PC = ..., PC+1; //stmt1
  ? PC == 3 -> ..., PC = ..., 1; //stmt2
```

Possibly one could the replace the program counter and use state variables that might introduce an optimization in model checking tools like UPPAAL. In the discussion there is more on the value of state variables that would possibly optimize these kinds of problems and make scripts easier to read.

6.2.1 Shared variables

Another script that was inspired by examples that can be found at the website of (Barnat et al., 2013) introduces the usage of the process model. This problem introduces two program instances running in parallel that manipulate a shared variable.

Let us consider two processes P and Q running in parallel:

```
P = loop { x=c; x=x+c; c=x;}
Q = loop { y=c; y=y+c; c=y;}
```

The variable c is shared between processes and its initial value 1. The claim is that c can contain any natural value. With the model checking tools that are used we cannot prove this, but for any value, we can try to find it.

Both processes consist of the same program. The program is translated to a `PROCESS` using the guidelines of section 6.2:

```
PROCESS addingProcess:
  VARS:
    int state : 0..2;
    int x : 0..256;
  INIT:
    x = 0;
    state = 0;
  TRANS:
    ? (tx == 0) -> x,state = c,1;
    ? (tx == 1) -> x,state = x+c,2;
    ? (tx == 2) -> c,state = x,0;
```

The shared variable c is defined globally. To test whether two instances can reach any value, two instantiations of the process are run in parallel:

```
VARS:
//Global shared variable that can reach any value
  int c : 0..256;
//Several processes that add to c in parallel
  proc P = addingProcess;
  proc Q = addingProcess;
```

```
INIT:
  c = 1;
PROGRAM:
//Run P in parallel with Q
  P || Q
```

Finally a reachability property for any value of c can be defined. A complete implementation of this model is given in Appendix C.

6.3 Alternating Bit Protocol

This problem introduces the implementation of a simple variant of the alternating bit protocol as described below.

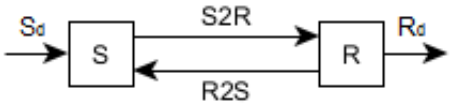

Figure 10: Alternating Bit Protocol

A sequence of data packages S_d of length n is sent from a sender S over an unreliable data channel $S2R$ to a receiver R. This means that during the transmission a data package can be lost and has to be sent again in order for R to receive it. The receiver stores each received package in its own sequence R_d, which is initially empty. The receiver can also send information back to the sender over an unreliable data channel $R2S$. The sender and receiver each control a so-called *alternating bit* S_b and R_b respectively. Initially the sender and receiver start with their alternating bit set to 0. The receiver constantly broadcasts the negation of R_b. It is flipped only when a package is received that carries a bit that is equal to R_b. The sender includes a copy of S_b in each package that it sends. The sender only flips S_b if it receives a bit that is equal to S_b.
To represent whether the data channels are currently filled we can use boolean variables $S2R_f$ and $R2S_f$ for the $S2R$ and $R2S$ channel respectively. If the variable is set to `true` it means the data channel is filled, it will be `false` otherwise. The alternating bits that are currently in the data channels are represented by variables $S2R_b$ and $R2S_b$. The value in these variables only has a meaning if the corresponding data channel is filled. The same principle goes for other data that is in the data channels, which is represented by variables $S2R_d$ and $R2S_d$.

Summed up, the rules of the protocol are as follows:

- At any moment the information that is carried in $S2R$ can be lost: $S2R_f$ `:= false`.
- At any moment the information that is carried in $R2S$ can be lost: $R2S_f$ `:= false`.
- At any moment $S2R$ can be filled by the first element of S_d, carrying S_b: $S2R_b$ `:= S_b;` $S2R_d$ `:= head(`S_d`);` $S2R_f$ `:= true`.
- At any moment $R2S$ can be filled by the negation of R_b: $R2S_b$ `:= not(`R_b`);` $R2S_f$ `:= true`.
- If $R2S$ carries a bit that has the same value as S_b, then the first element of S_d may be removed while S_b is flipped: `if` $R2S_f$ `and` $R2S_b = S_b$ { `remove head(`S_d`);` S_b `:= not(`S_b`)` }.
- If $S2R$ carries a data package with a bit that has the same value as R_b, then the package is added to R_d while R_b is flipped: `if` $S2R_f$ `and` $R2S_b = R_b$ { `add` B_d `to` R_d`;` R_b `:= not(`R_b`)` }.

There are two properties that must be satisfied:

1. The initial length of S_d must be equal to the final length of R_d.
2. If property 1 holds, then the initial content of S_d must be the same and in the same order as the final content R_d.

A correct implementation of the protocol guarentees both properties. In the script we can verify only one property at the time. To verify property 1 we can specify a reachability property that states that R_d has the same length as the initial length of S_d for a particular data size. To verify property 2 we specify that property 1 and the negation of property 2 must be unreachable if the protocol is implemented correctly. In other words, we try to find a state in which both lengths of the arrays are equal, but the content is at any place incorrect. Verifying just one of these properties is not a complete proof of the correctness of the protocol, but still meaningful to experiment with later on. An example implementation of the complete protocol that verifies property 2 can be found in Appendix G.

6.4 Robot vacuum cleaner

This problem was taken from (Vaandrager & Verbeek, 2014). It describes a robot vacuum cleaner that must make its way across an n x n grid such that it visits all cells and returns in the same state as it started. The robot has no memory of where it has been, it merely knows its current position and orientation. The robot is always facing either north, east, south or west and located in one of the cells of the grid. Initially, the robot starts in cell (0,0) and is facing north. It is allowed to take two of the following actions in each cell:

- Turn 90 degrees

- Move forward if not facing a wall

Hereby it is assumed that all the actions always have the intended effect. The question asked is whether there exists a strategy for an n x n grid such that the robot always knows what to do and that it visits all cells infinitely often. Figure 11 that was taken directly from (Vaandrager & Verbeek, 2014) shows two possible strategies for a 3 x 3 grid. An example implementation for a 2 x 2 grid is given in Appendix H.

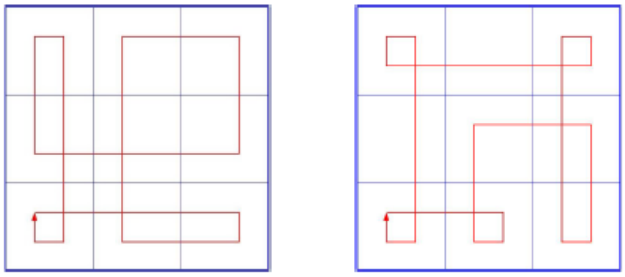

Figure 11: Two strategies of 24 (left) and 26 (right) actions

Results from Vaandrager et al. show that Uppaal runs out of memory for a 5 x 5 grid and the largest instance they could solve using a SAT-solver called zChaff was a 7 x 7 grid.

7 Experiments

In this section, each model checker or solver was benchmarked using the new tool with custom instances of the example problems of the previous section. All benchmarks were performed on a Core i7 2700K processor running at a clock frequency of 4.8GHz with 8GB of RAM running at 2133MHz using Windows 7 Professional as operating system. All the scripts for problem instances were automatically generated and available online[8]. The results outlined in this section were obtained using the following external tools: Yices 1.0.40, NuSMV 2.5.4 and UPPAAL 4.0.13. These versions were common to use during the writing of this paper, but the tools are still active in development. Using any other or newer versions of these tools than the ones mentioned may lead to significantly different results and conclusions.

In these synthetic benchmarks the solutions to the problems and when applicable the number of steps that are needed to reach the solution were determined beforehand by custom experiments. The maximum amount of steps needed is always found by trying different maximum steps values and determining the lowest value for which the property is still verified. Specifying the maximum number of steps can have a great impact on solving performance both in negative and positive way. For each separate problem the choice of whether or not to specify the maximum steps will be explained. For Yices this value is always specified, because its implementation depends on it.

What makes it hard to do meaningful and fair benchmarks is that each tool has its own limitations and perculiarities as was described in section 4.3. Yices is known to rapidly decrease in performance if the step counter is very high and NuSMV is known decrease in performance if the variable range is very wide. Finally UPPAAL has great difficulties with non-deterministic variables and runs out of memory quickly. This has all been taken into account when designing experiments.
If a tool takes a very long time to solve a problem, this is usually a significant while longer due to the exponential and combinatorial complexity of the problems and solvers. Therefore a time-out of fifteen minutes is used. Usually within this time frame at least one of the tools returns an answer for each of the benchmark problems.

7.1 Weighted graphs

Here a deterministic example of the truck delivery problem was used that is represented by the following graph:

[8] `https://bitbucket.org/VultureX/unifiedmodelcheckingprototype`

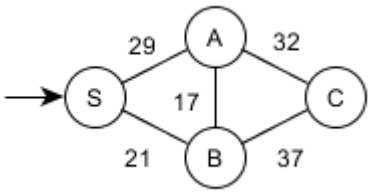

Figure 12: Truck Delivery Benchmark Instance

The initial parameters for the problem are specified in the following table:

City	A	B	C
Initial value	50	40	150
Maximum value	120	120	200

Table 1: Truck Delivery Benchmark Parameters

Custom testing reveiled that an infinite cycle becomes possible at a maximum truck capacity of 319 and above and it always takes at least 17 steps to show this. Since loop checking disallows the specification of the maximum steps variable it was omitted for NuSMV and UPPAAL. The amount of steps for Yices was tested with 17, 22 and 27 to gain insight in the performance impact for overestimating the maximum number of steps for bounded model checking. The maximum capacity of the truck was varied from 315 to 322.

Truck capacity	NuSMV	UPPAAL	Yices 17	Yices 22	Yices 27	Sat
315	1.03	0.04	0.86	2.46	3.22	No
316	1.10	0.06	1.09	2.30	2.84	No
317	1.24	0.05	1.02	2.20	9.36	No
318	1.12	0.08	1.01	1.34	3.10	No
319	1.10	0.11	0.97	3.32	3.23	Yes
320	1.11	0.08	1.20	3.05	7.02	Yes
321	1.25	0.09	1.35	1.68	4.89	Yes
322	1.17	0.11	1.33	3.40	9.79	Yes
Avg time	1.14	0.08	1.08	2.47	5.43	

Table 2: Deterministic Truck results

NuSMV and UPPAAL give a definitive conclusion on the absence of a loop, since they are not bounded by a number of steps. For Yices, unsatisfiability of the formula may not be regarded as definitive proof of the absence of a loop.

Additional results between Yices and NuSMV were obtained with the same instance where each time a non-deterministically chosen amount of packages

could be unloaded from the truck. The least amount of steps needed for a cycle was 11 for an instance with a truck with capacity 318 or higher. The amount of steps for Yices was tested with 11, 16 and 21.

Truck capacity	NuSMV	Yices 11	Yices 16	Yices 21	Sat
315	T/O	1.7	19.38	83.92	No
316	T/O	1.89	25.40	63.41	No
317	T/O	2.19	25.90	81.05	No
318	T/O	1.49	25.46	38.93	Yes
319	T/O	1.02	12.19	27.69	Yes
320	T/O	2.26	10.54	58.64	Yes
321	T/O	1.81	20.43	63.53	Yes
322	T/O	2.06	13.35	35.13	Yes
Avg time	N/A	1.80	19.08	56.53	

Table 3: Non-Deterministic Truck results

7.2 Alternating Bit Protocol

In this experiment instances of the Alternating Bit Protocol (ABP) as described in section 6.3 were used. To verify the correctness of the protocol, the received array must be equal to the sent array in all possible configurations and executions. However this requires a non-deterministic initialization of the array. Futhermore, an execution trace may be infinitely long. For full correctness of the protocol both properties specified in section 6.3 must be verified. During the experiments we will only verify the correctness of property 2 (see section 6.3) to keep the number of benchmarks to a minimum. The first property is assumed to be verified earlier. Due to several limitations of the tools, only NuSMV is able to give us a definitive answer regarding the verification of property 2 for Booleans and to some degree for integers, since NuSMV is able to deal with non-deterministic initialization and is not bounded by a maximum number of steps.

In order to be able to benchmark each tool, several tests were run using altered properties such that still a comparison can be made. To be able to run benchmarks for UPPAAL correctness was only verified for a sender's array of which the variables are set to 0 or false and a receiver's array of which the variables are initialized to 1 or true depending on the data-type used.
The protocol can only be verified for finite execution traces by Yices. Therefore we determine via custom testing the amount of steps needed to get send the array to the receiver. In order to be fair we use this number + 10 to allow slightly longer execution paths in which errors may occur. This maximum number of steps is specified for all tools. Finally, since NuSMV works badly with wide integer ranges, the maximum range of integer values is set to 0..1023.

This results in the first set of experiments that benchmark the ABP using various queue lengths and maximum steps values, two different data-types and the fixed starting state. The results are shown in Table 4 and 5.

Queue size	Max Steps	Yices	NuSMV	UPPAAL
1	14	0.06	0.04	0.03
2	18	0.17	0.04	0.04
3	22	0.57	0.04	0.03
4	26	1.34	0.06	0.04
5	30	2.79	0.09	0.06
6	34	5.67	0.13	0.07
7	38	36.76	0.21	0.08
8	42	81.70	0.34	0.10
9	46	227.29	0.68	0.12
10	50	335.55	1.01	0.14
11	54	217.49	2.26	0.12
12	58	488.43	7.14	0.16
13	62	T/O	17.14	0.15
14	66	T/O	40.34	0.21
15	70	T/O	89.93	0.26
16	74	T/O	237.68	0.24
17	78	T/O	650.43	0.31
18	82	T/O	T/O	0.34

Table 4: Results for a fixed initial Boolean array, limited steps

Queue size	Max Steps	Yices	NuSMV	UPPAAL
1	14	0.12	1.34	0.03
2	18	0.30	3.52	0.03
3	22	0.64	5.88	0.04
4	26	2.58	9.02	0.05
5	30	7.78	13.18	0.06
6	34	18.91	19.57	0.06
7	38	41.47	26.51	0.12
8	42	81.36	37.68	0.11
9	46	282.57	55.62	0.11
10	50	256.07	89.44	0.13
11	54	489.57	T/O	0.14
12	58	647.37	T/O	0.16
13	62	632.45	T/O	0.19
14	66	T/O	T/O	0.22
15	70	T/O	T/O	0.24

Table 5: Results for a fixed initial integer array, limited steps

The second set of benchmarks shows the results for a fixed array with an unlimited amount of steps. As such only NuSMV and UPPAAL could be tested. The results are shown in Table 6 and 7.

Queue size	NuSMV	UPPAAL
1	0.06	0.02
2	0.03	0.03
3	0.03	0.03
4	0.04	0.03
5	0.04	0.03
6	0.05	0.03
7	0.06	0.03
8	0.07	0.03
9	0.12	0.03
10	0.19	0.03
11	0.38	0.04
12	0.79	0.03
13	2.32	0.07
14	5.31	0.05
15	11.83	0.04
16	33.38	0.06
17	95.14	0.04
18	209.14	0.06
19	560.95	0.06
20	T/O (mem)	0.04

Table 6: Results for a fixed initial Boolean array, unlimited steps

Queue size	NuSMV	UPPAAL
1	1.78	0.03
2	3.93	0.03
3	6.54	0.03
4	9.86	0.03
5	13.68	0.03
6	18.29	0.04
7	23.91	0.03
8	32.14	0.06
9	40.31	0.05
10	49.55	0.03
11	T/O	0.03
12	T/O	0.06

Table 7: Results for a fixed initial integer array, unlimited steps

The last set of benchmarks uses a non-deterministic starting state, which is only possible with Yices and NuSMV due to limitations of UPPAAL as discussed in section 4.3. Again we use the maximum steps value for both tools in order to remain fair. The results are shown in Table 8 and 9. The last column shows NuSMV with an unlimited amount of steps.

Queue size	Max Steps	Yices	NuSMV	NuSMV (unlim steps)
1	14	0.05	0.03	0.02
2	18	0.23	0.04	0.03
3	22	0.44	0.06	0.02
4	26	1.11	0.13	0.03
5	30	2.49	0.38	0.04
6	34	16.53	1.23	0.08
7	38	40.69	3.69	0.21
8	42	82.14	15.76	0.73
9	46	179.88	70.65	3.91
10	50	311.29	360.48	24.22
11	54	407.93	T/O	199.25
12	58	857.62	T/O	T/O

Table 8: Results for a any initial Boolean array

Queue size	Max Steps	Yices	NuSMV	NuSMV (unlim)
1	14	0.06	1.82	1.67
2	18	0.32	4.28	3.75
3	22	1.08	7.71	6.27
4	26	3.72	13.55	10.04
5	30	15.9	28.57	14.21
6	34	59.64	74.42	20.58
7	38	89.65	322.79	37.37
8	42	281.91	T/O	435.66
9	46	340.06	T/O	T/O

Table 9: Results for any initial integer array

7.3 Shared variables

In this experiment the problem described in section 6.2.1 was benchmarked with values for the shared variable c between 0 and 1023. During experimenting we found that specifying a maximum steps value and whether or not the goal value was a power of 2 had a great impact on performance. Therefore we show results divided in several categories.

The first set of experiments was performed with a maximum steps value supplied to all tools. Custom tests reveiled the minimum amount of steps needed to reach

the target value. This amount was inserted as maximum steps value. Table 10 shows results where $c = 2^k$ for $k \in \{1, \ldots, 9\}$. Table 11 shows results for other random values.

Shared variable	Max Steps	Yices	NuSMV	UPPAAL
2	3	0.02	3.1	0.03
4	6	0.03	2.95	0.02
8	9	0.12	3.05	0.03
16	12	0.41	3.15	0.03
32	15	0.22	3.73	0.04
64	18	8.92	4.16	0.09
128	21	28.16	4.38	0.19
256	24	737.7	4.20	0.29
512	27	T/O	4.00	0.59

Table 10: Results for $c = 2^k$ with $k \in \{0, \ldots, 9\}$, limited steps

Shared Variable	Max Steps	Yices	NuSMV	UPPAAL
6	9	0.06	4.02	0.02
15	15	0.43	4.04	0.03
31	21	85.96	4.26	0.11
63	24	685.27	4.40	0.23
127	30	T/O	5.41	1.50
156	27	43.61	4.87	0.55
222	30	826.37	5.54	1.48
355	33	T/O	7.75	3.72
404	33	T/O	7.80	3.64
543	36	T/O	13.16	9.59
689	36	T/O	12.96	9.75
713	39	T/O	23.96	24.03
860	36	T/O	13.24	10.6
917	39	T/O	25.35	25.3
1000	36	T/O	13.13	9.22

Table 11: Results for random values of c, limited steps

The second set of experiments was performed without a maximum steps value. For c, the same values as in the first set were used.

c	NuSMV	UPPAAL
2	T/O	0.02
4	T/O	0.03
8	T/O	0.03
16	T/O	0.03
32	T/O	0.04
64	T/O	0.06
128	T/O	0.10
256	T/O	0.16
512	T/O	0.36

Table 12: Results for $c = 2^k$ with $k \in \{0, \ldots, 9\}$, unlimited steps

c	NuSMV	UPPAAL
6	T/O	0.02
15	266.6	0.04
31	375.24	0.08
63	193.44	0.20
127	110.04	1.11
156	482.46	0.44
222	146.51	1.00
355	76.87	2.57
404	146.52	2.44
543	79.31	6.36
689	78.66	6.05
713	77.41	13.9
860	109.44	6.1
917	78.73	13.97
1000	126.18	5.54

Table 13: Results for random values of c, unlimited steps

Finally some benchmarks were run to see how the maximum steps value impacts Yices' performance. The maximum steps value was increased with 3, 6, 9 and 12 compared to values in previous tables.

c	Max Steps	Yices +0	Yices +3	Yices +6	Yices +9	Yices +12
2	3	0.02	0.02	0.03	0.03	0.07
4	6	0.03	0.03	0.04	0.09	0.12
8	9	0.12	0.02	0.03	0.16	0.56
16	12	0.41	0.12	0.41	0.48	1.07
32	15	0.22	0.69	0.45	0.52	0.9
64	18	8.92	1.37	2.1	3.56	5.52
128	21	28.16	11.77	3.95	4.18	17.65
256	24	737.7	44.56	6.54	5.03	15.03
512	27	T/O	362.17	73.55	64.78	64.58

Table 14: Results for $c = 2^k$ with $k \in \{0, \ldots, 9\}$ for Yices with variable maximum steps

c	Max Steps	Yices +0	Yices +3	Yices +6	Yices +9	Yices +12
6	9	0.06	0.03	0.19	0.32	0.5
15	15	0.43	0.88	12.62	19.51	12.51
31	21	85.96	T/O	27.94	611.55	127.57
63	24	685.27	86.95	32.56	75.88	33.09
127	30	T/O	T/O	T/O	T/O	T/O
156	27	43.61	45.22	71.24	79.40	697.87
222	30	826.37	782.08	760.88	T/O	T/O
355	33	T/O	T/O	T/O	T/O	T/O

Table 15: Results for random values of c, for Yices with variable maximum steps

7.4 Robot vacuum cleaner

In this experiment the problem described in section 6.4 was benchmarked with values for the grid size n between 2 and 5. Custom testing reveiled the minimum amount of actions required to create a looping strategy. This amount was inserted as maximum steps value for all tools. We were able to verify that a strategy for a 5 x 5 grid needs at least 40 steps. In Table 16 it is shown that at 36 steps UPPAAL ran out of memory address space due to its 32 bit architecture, but it is interesting to note that NuSMV needed twice as much memory as UPPAAL's limit to find the solution.

Max Steps	Yices	NuSMV	UPPAAL	Satisfiable
34	T/O	199.47	794.75	No
36	T/O	312.08	N/A (mem 705.80)	No
38	T/O	437.65	N/A (mem 705.41)	No
40	T/O	655.70	N/A (mem 705.19)	Yes

Table 16: Results of experiments to determine the shortest strategy in a 5 x 5 grid

Table 17 shows the results for n x n grids with a maximum steps restriction.

n	Max Steps	Yices	NuSMV	UPPAAL
2	8	0.08	0.07	0.21
3	24	1.41	0.72	0.55
4	28	6.33	9.99	7.34
5	40	T/O	655.70	N/A (mem)

Table 17: Results for an n x n grid, limited steps

Finally Table 18 shows the results for n x n grids without a maximum steps restriction.

n	NuSMV	UPPAAL
2	0.09	0.21
3	1.26	0.39
4	T/O	6.42
5	T/O	N/A (mem)

Table 18: Results for n x n grid, unlimited steps

7.5 Conclusions

In general we can conclude that non-determinism greatly increases the solving time that is needed. This is seen in both the non-deterministic truck delivery problem and the ABP for any array.

Furthermore we see that whether or not the maximum amount of steps is specified matters for performance. In general NuSMV and UPPAAL tend to perform better when they are not bounded by a maximum number of steps, but the opposite is true for the shared variable problem. This may have to do with the fact that the update rules allow one process to update infinitely long while the other process is not updated at all. This is also known as starvation. In this case we see that it helps to specify a maximum amount of steps to minimize depth of the search tree.

Yices ultimately starts performing worse if the maximum number of steps is too high due to huge amounts of variables that are created while unfolding the transition system. However Yices may benefit from more steps if that means that more desired states become available. This is very apparent in Table 14, where Yices performs better if the maximum steps value is higher than the minimum required for a solution. Also in Table 15 we can see that trying out different values may benefit performance in a way that is not obvious. Some times increasing the maximum amount of steps increases performance, but many times the opposite is true.

Table 19 gives a performance ranking for each benchmarking problem. The tools are rated from 1 to 3, where a rating of 1 is the fastest and 3 is the slowest.

Problem	Yices	NuSMV	UPPAAL
Truck Delivery, deterministic unload	2	3	1
Truck Delivery, non-deterministic unload	1	2	N/A
ABP Fixed Array, limited Steps	2/3	2/3	1
ABP Fixed Array, unlimited Steps	N/A	2	1
ABP Any Array, limited Steps	1	2	N/A
ABP Any Array, unlimited Steps	N/A	1	N/A
Shared Var $c = 2^k$	3	2	1
Shared Var c = random	3	2	1
Robot Vacuum Cleaner	2	1	3

Table 19: General Performance Rating

In case of the deterministic ABP with limited steps the result for Yices and NuSMV was a tie, because Yices performed better with integer arrays while NuSMV was better with Boolean arrays. Overall it appears that UPPAAL is the great winner if no non-determinism is involved, but exceptions in the robot vacuum cleaner problem show that this does not always have to be the case. With the new unified approach the chances of infeasibility that are inherent to one approach are decreased. This emphasizes the benefit of the inclusion of multiple tools, since for all approaches, there are different problems that are infeasible for one, but very feasible for the other.

8 Discussion

A universal scripting language as proposed in this paper can provide an easy way to expose model checking problems to a whole range of different model checking tools. It reduces the time needed to implement problem instances, but in its current form it still misses the full expressiveness of for example the LTL formal language. Future additions to the language could overcome these limitations. For now, a great deal of problems can be translated and the most frequently occurring properties can be verified. A few additions that came to mind while developing this tool are discussed.

A special state variable could make the description of some problems easier to read, because it eliminates the need for a variable that keeps track of the current state. The implementation script for a weighted graph like the one in Figure 9 could be written like this:

```
VARS:
  int value;
  state S, A, B; //New state definitions using a state keyword
  const int S_A = 10;
  const int A_B = 5;
  const int B_S = 10;
INIT:
  state = S; //New state initialization
  value = 0;
TRANS:
//New state transitions. The original update rules are moved
//between curly brackets.
  S -> A { value = value+S_A };
  A -> B { value = value+A_B };
  B -> S { value = value+B_S };
```

The introduction of the state variable reduces some syntax clutter that is needed to define constant integers. Furthermore the state transition is immediately clear, while the rest of the transition rules can remain unaltered and placed between the curly brackets. Occasional guards can still be used inside the curly brackets or they can be placed in front of the entire statement according to one's preference. A similar syntax is already used in the DIVINE tool (Barnat et al., 2013).

```
TRANS:
  //Example of state transition from S to A. The transition
  //is only taken if the condition holds. The condition is
  //specified up front and a new ':' character denotes what
  //the state transition should do.
  ? condition : S -> A { value = value+S_A };
```

```
//Example with the same meaning as above, but that retains
//the original update rule within the curly brackets.
S -> A { ? condition -> value = value+S_A };
```

Internally the new syntax would be translated using an integer state variable for Yices and NuSMV, but the translation to UPPAAL may benefit from this new syntax, since its own native implementation of states can be used instead of introducing a superfluous integer that keeps track of the state.

Furthermore the script language is limited in terms of process algebra and verifyable properties. Now it is possible to run separate process instances, but merely in parallel. An extension to the language could allow to also sequence processes or make a non-deterministic choice between processes. For example a single trace of the following program would either return a trace of Q or a trace of P followed by R.

```
PROGRAM:
  (P >: R) | Q
```

In the future it might also be useful to make the language richer in terms of property verification, i.e. it should be possible to verify more classes of safety and liveness properties. However the original goal was to avoid a full implementation of LTL to keep the language easy to read and intuitive to understand. Perhaps a golden medium can be found in UPPAAL's syntax that supports a few more expressions than just reachability and liveness properties as were defined here. For example in UPPAAL it is possible to define an expression $p \rightsquigarrow r$ that means that if p holds, then eventually r must also hold. Another example is `E[] p`, which means that there must exists a maximal path where p holds. A maximal path is defined as an infinite path or a path that leads to a state with no outgoing edges (Behrmann et al., 2004).

Finally an improved constant propagation algorithm may improve results. The constant propagation that is currently used is very simple and is not applied to more complicated expressions. In (Tzoref, Matusevich, Berger, & Beer, 2003) they use constant propagation in the translation of a finite state machine (FSM) to SMT. It identifies constant signals that are automatically propagated throughout the unfolded FSM, which generally results in a performance optimization. The same principal could apply to this scripting language.

A Example transition system script

```
MAXSTEPS:
  100
VARS:
  int x;
INIT:
  x = 1;
TRANS:
  x = x-1;
  x = x*2;
REACH:
  x == 99
```

B Example process model script

```
MAXSTEPS:
  100
VARS:
  proc P = example;
PROGRAM:
  P
REACH:
  P.x == 99

PROCESS example:
  VARS:
    int x;
  INIT:
    x = 1;
  TRANS:
    x = x-1;
    x = x*2;
```

C Parallel adding script

The following is an example of a reachability problem taken from a set of example problems of the DIVINE tool by (Barnat et al., 2013).

Two processes P and Q are running in parallel:

```
  P = loop {x=c; x=x+c; c=x;}
  Q = loop {y=c; y=y+c; c=y;}
```

The initial value of global variable c is 1 and x and y are local process variables. The claim is that c can possibly contain any natural value. With model checkers we cannot prove this, but for any value, we can try to find it. The following is an example instance searching for $c = 245$ encoded as process model.

```
//Parallel adding problem
MAXSTEPS:
  30
VARS:
//Global variable that can reach any value
  int c : 0..256;
//Several processes that add to c in parallel
  proc P = addingProcess;
  proc Q = addingProcess;
INIT:
  c = 1;
PROGRAM:
//Run P in parallel with Q
  P || Q
REACH:
//Can c reach the value 245?
  c == 245

PROCESS addingProcess:
  VARS:
    int theState : 0..2;
    int x : 0..256;
  INIT:
    x = 0;
    theState = 0;
  TRANS:
    ? (theState == 0) -> x,theState = c,1;
    ? (theState == 1) -> x,theState = x+c,2;
    ? (theState == 2) -> c,theState = x,0;
```

D Weighted Graph Example Script

```
//This script represents a weighted graph:
// S -10-> A -5-> B -10-> S
VARS:
  int sum; //sum of the edges
  int currentState 0..2; //state variable
  const int S = 0; //state S
  const int A = 1; //state A
```

```
  const int B = 2; //state B
  const int S_A = 10; //edge from S to A with weight 10
  const int A_B = 5; //edge from A to B with weight 5
  const int B_S = 10; //edge from B to S with weight 10
INIT:
  currentState = S; //Start in S
  sum = 0;
TRANS:
//For each state transition: update the sum with the edge weight
  currentState == S -> currentState,value = A,value+S_A;
  currentState == A -> currentState,value = B,value+A_B;
  currentState == B -> currentState,value = S,value+B_S;
REACH:
//Is there a path starting from S of total weight 25?
  sum == 25
```

E Truck Delivery Example Script 1

```
//A script that describes a food truck, traveling from state to
//state. It always unloading as many food packages as possible in
//each state.
VARS:
  int truck : 0.. TRUCK_MAX; //Amount of packages in truck
  int state : 0..2; //The state the truck is in
  int Ac : 0..A_MAX; //Amount of packages in A
  int Bc : 0..B_MAX; //Amount of packages in B
  const int S = 0; //depot
  const int A = 1; //state A
  const int B = 2; //state B
  const int S_A = 10; //edge from S to A with weight 10
  const int A_B = 5; //edge from A to B with weight 5
  const int B_S = 10; //edge from B to S with weight 10
  const int TRUCK_MAX = 20; //maximum truck capacity
  const int A_MAX = 30; //maximum capacity of state A
  const int B_MAX = 30; //maximum capacity of state B
INIT:
  //The truck begins in S and is fully loaded
  currentState = S;
  truckLoad = TRUCK_MAX;
  Ac = 20;
  Bc = 20;
TRANS:
//Transition from S to A
  ? state == S & Ac-S_A >= 0 & truck - (A_MAX - (Ac - S_A)) >= 0
```

```
  -> state, truck, Ac, Bc
  = A, truck - (A_MAX - (Ac - S_A)), A_MAX, Bc-S_A;
  ? state == S & A-S_A >= 0 & truck - (A_MAX - (Ac - S_A)) < 0
  -> state, truck, Ac, Bc
  = A, 0, (Ac - S_A) + truck, Bc-S_A;

//Transition from A to B
  ? state == A & Bc-A_B >= 0 & truck - (B_MAX - (Bc - A_B)) >= 0
  -> state, truck, A, B
  = B, truck - (B_MAX - (Bc - A_B)), A-A_B, B_MAX;
  ? state == A & Bc-A_B >= 0 & truck - (B_MAX - (Bc - A_B)) < 0
  -> state, truck, A, B
  = B, 0, (Bc - A_B) + truck, B_MAX;

//Transition from B to S
  ? state == B
  -> state, truck, Ac, Bc
  = S, TRUCK_MAX, Ac-B_S, Bc-B_S;
INF:
```

F Weighted Graph Example Script 2

```
//A script that describes a food truck, traveling from state to
//state. It unloads a non-determinstic amount of food packages in
//each state.
VARS:
  int truck : 0.. TRUCK_MAX; //Amount of packages in truck
  int state : 0..2; //The state the truck is in
  int Ac : 0..A_MAX; //Amount of packages in A
  int Bc : 0..B_MAX; //Amount of packages in B
  const int S = 0; //depot
  const int A = 1; //state A
  const int B = 2; //state B
  const int S_A = 10; //edge from S to A with weight 10
  const int A_B = 5; //edge from A to B with weight 5
  const int B_S = 10; //edge from B to S with weight 10
  const int TRUCK_MAX = 20; //maximum truck capacity
  const int A_MAX = 30; //maximum capacity of state A
  const int B_MAX = 30; //maximum capacity of state B
  nondet int amount; //an amount that the truck unloads
INIT:
  //The truck begins in S and is fully loaded
  currentState = S;
  truckLoad = TRUCK_MAX;
```

```
  Ac = 20;
  Bc = 20;
TRANS:
//Transition from S to A
  ? state == S & Ac-S_A >= 0
  -> state, truck, A, B
  = A, truck - amount, (Ac - S_A) + amount, Bc-S_A;

//Transition from A to B
  ? state == A & Bc-A_B >= 0
  -> state, truck, A, B =
    B, truck - amount, Ac-A_B, (Bc - A_B) + amount;

//Transition from B to S
  ? state == B
  -> state, truck, A, B
  = S, TRUCK_MAX, Ac-B_S, Bc-B_S;
INF:
```

G Alternating Bit Protocol Script

```
//Alternating bit protocol example with queue length = 3 and
//boolean data type:
VARS:
bool Sb;  //Sender bit
bool S2Rb;  //Sender to receiver channel bit
bool Rb;  //Receiver bit
bool R2Sb;  //Receiver to sender channel bit
bool S2Rf; //S2R channel is filled
bool R2Sf; //R2S channel is filled
bool S2Rd; //data currently in S2R
//Sender sequence data:
bool Sd0;
bool Sd1;
bool Sd2;
//Receiver sequence data and whether or not it's received:
bool Rd0;
bool Rd1;
bool Rd2;
int Snum : 0..3; //Current sender sequence number
int Rnum : 0..3; //Current receiver sequence number
INIT:
//Sender and receiver start at the beginning of the sequence
Snum = 0;
```

```
Rnum = 0;
//Channels are not filled:
S2Rf = false;
R2Sf = false;
//Initial values of sender and receiver bits:
Sb = false;
Rb = false;
TRANS:
//At any moment data can be lost:
S2Rf = false;
R2Sf = false;
//At any moment the sender can fill S2R with its current
//element, carrying bit Sb:
? Snum == 0 -> S2Rb,S2Rd,S2Rf = Sb,Sd0,true;
? Snum == 1 -> S2Rb,S2Rd,S2Rf = Sb,Sd1,true;
? Snum == 2 -> S2Rb,S2Rd,S2Rf = Sb,Sd2,true;
//At any moment the receiver can fill R2S with not(Rb)
R2Sb,R2Sf = !Rb,true;
//Sender alternating bit and progressing sequence:
? R2Sf & R2Sb == Sb -> Snum,Sb = Snum+1,!Sb;
//Receiver alternating bit:
? S2Rf & S2Rb == Rb & Snum == 0
-> Rd0,Rb,Rnum = S2Rd,!Rb,Rnum+1;
? S2Rf & S2Rb == Rb & Snum == 1
-> Rd1,Rb,Rnum = S2Rd,!Rb,Rnum+1;
? S2Rf & S2Rb == Rb & Snum == 2
-> Rd2,Rb,Rnum = S2Rd,!Rb,Rnum+1;
REACH:
Snum == 3 & Rnum == 3 &
(Sd0 == !Rd0 | Sd1 == !Rd1 | Sd2 == !Rd2)
```

H Robot Vacuum Cleaner script

```
//Robot vacuum problem
MAXSTEPS:
  8
VARS:
  //Room size
  const int n = 2;

  //Strategy values
  const int UNDEF = 0;
  const int TURN = 1;
  const int FORWARD = 2;
```

```
  //Directions
  const int N = 0;
  const int E = 1;
  const int S = 2;
  const int W = 3;

  //Strategy as defined so far for each cell and orientation:
  int x0y0N : UNDEF..FORWARD;  int x0y0E : UNDEF..FORWARD;
  int x0y0S : UNDEF..FORWARD;  int x0y0W : UNDEF..FORWARD;

  int x1y0N : UNDEF..FORWARD;  int x1y0E : UNDEF..FORWARD;
  int x1y0S : UNDEF..FORWARD;  int x1y0W : UNDEF..FORWARD;

  int x0y1N : UNDEF..FORWARD;  int x0y1E : UNDEF..FORWARD;
  int x0y1S : UNDEF..FORWARD;  int x0y1W : UNDEF..FORWARD;

  int x1y1N : UNDEF..FORWARD;  int x1y1E : UNDEF..FORWARD;
  int x1y1S : UNDEF..FORWARD;  int x1y1W : UNDEF..FORWARD;

  //Current coordinates and direction:
  const int MAX = 1; //n-1
  int x : 0..MAX;
  int y : 0..MAX;
  int d : N..W;

INIT:
  //Begin in cell (0,0) facing north:
  x = 0;
  y = 0;
  d = N;

  x0y0N = UNDEF;  x0y0E = UNDEF;  x0y0S = UNDEF;  x0y0W = UNDEF;
  x1y0N = UNDEF;  x1y0E = UNDEF;  x1y0S = UNDEF;  x1y0W = UNDEF;
  x0y1N = UNDEF;  x0y1E = UNDEF;  x0y1S = UNDEF;  x0y1W = UNDEF;
  x1y1N = UNDEF;  x1y1E = UNDEF;  x1y1S = UNDEF;  x1y1W = UNDEF;

TRANS:
//Forward:
  ? x == 0 & y == 0 & d == N & !(x0y0N == TURN)
  -> x,y,d,x0y0N = x,1,d,FORWARD;
  ? x == 0 & y == 0 & d == E & !(x0y0E == TURN)
  -> x,y,d,x0y0E = 1,y,d,FORWARD;

  ? x == 1 & y == 0 & d == N & !(x1y0N == TURN)
  -> x,y,d,x1y0N = x,1,d,FORWARD;
```

```
  ? x == 1 & y == 0 & d == W & !(x1y0W == TURN)
  -> x,y,d,x1y0W = 0,y,d,FORWARD;

  ? x == 0 & y == 1 & d == E & !(x0y1E == TURN)
  -> x,y,d,x0y1E = 1,y,d,FORWARD;
  ? x == 0 & y == 1 & d == S & !(x0y1S == TURN)
  -> x,y,d,x0y1S = x,0,d,FORWARD;

  ? x == 1 & y == 1 & d == S & !(x1y1S == TURN)
  -> x,y,d,x1y1S = x,0,d,FORWARD;
  ? x == 1 & y == 1 & d == W & !(x1y1W == TURN)
  -> x,y,d,x1y1W = 0,y,d,FORWARD;

//Turn:
  ? x == 0 & y == 0 & d == N & !(x0y0N == FORWARD)
  -> x,y,d,x0y0N = x,y,E,TURN;
  ? x == 0 & y == 0 & d == E & !(x0y0E == FORWARD)
  -> x,y,d,x0y0E = x,y,S,TURN;
  ? x == 0 & y == 0 & d == S & !(x0y0S == FORWARD)
  -> x,y,d,x0y0S = x,y,W,TURN;
  ? x == 0 & y == 0 & d == W & !(x0y0W == FORWARD)
  -> x,y,d,x0y0W = x,y,N,TURN;

  ? x == 1 & y == 0 & d == N & !(x1y0N == FORWARD)
  -> x,y,d,x1y0N = x,y,E,TURN;
  ? x == 1 & y == 0 & d == E & !(x1y0E == FORWARD)
  -> x,y,d,x1y0E = x,y,S,TURN;
  ? x == 1 & y == 0 & d == S & !(x1y0S == FORWARD)
  -> x,y,d,x1y0S = x,y,W,TURN;
  ? x == 1 & y == 0 & d == W & !(x1y0W == FORWARD)
  -> x,y,d,x1y0W = x,y,N,TURN;

  ? x == 0 & y == 1 & d == N & !(x0y1N == FORWARD)
  -> x,y,d,x0y1N = x,y,E,TURN;
  ? x == 0 & y == 1 & d == E & !(x0y1E == FORWARD)
  -> x,y,d,x0y1E = x,y,S,TURN;
  ? x == 0 & y == 1 & d == S & !(x0y1S == FORWARD)
  -> x,y,d,x0y1S = x,y,W,TURN;
  ? x == 0 & y == 1 & d == W & !(x0y1W == FORWARD)
  -> x,y,d,x0y1W = x,y,N,TURN;

  ? x == 1 & y == 1 & d == N & !(x1y1N == FORWARD)
  -> x,y,d,x1y1N = x,y,E,TURN;
  ? x == 1 & y == 1 & d == E & !(x1y1E == FORWARD)
  -> x,y,d,x1y1E = x,y,S,TURN;
  ? x == 1 & y == 1 & d == S & !(x1y1S == FORWARD)
```

```
  -> x,y,d,x1y1S = x,y,W,TURN;
  ? x == 1 & y == 1 & d == W & !(x1y1W == FORWARD)
  -> x,y,d,x1y1W = x,y,N,TURN;

REACH:
  !(x0y0N == UNDEF & x0y0E == UNDEF
  & x0y0S == UNDEF & x0y0W == UNDEF) &
  !(x1y0N == UNDEF & x1y0E == UNDEF
  & x1y0S == UNDEF & x1y0W == UNDEF) &
  !(x0y1N == UNDEF & x0y1E == UNDEF
  & x0y1S == UNDEF & x0y1W == UNDEF) &
  !(x1y1N == UNDEF & x1y1E == UNDEF
  & x1y1S == UNDEF & x1y1W == UNDEF) &
  x == 0 & y == 0 & d == N
```

I Syntax

The formal description of the language syntax is as follows.

⟨*Formula*⟩ ::= ⟨*TransitionSystem*⟩ | ⟨*ProcessModel*⟩

⟨*TransitionSystem*⟩ ::= [‘`MAXSTEPS:`’ ⟨*Constant*⟩]
‘`VARS:`’ ⟨*VarDeclTS*⟩$^+$
[‘`INIT:`’ ⟨*VarInit*⟩*]
‘`TRANS:`’ ⟨*Stmt*⟩$^+$
⟨*Goal*⟩

⟨*ProcessModel*⟩ ::= [‘`MAXSTEPS:`’ ⟨*Constant*⟩]
‘`VARS:`’ ⟨*VarDeclPM*⟩$^+$
[‘`INIT:`’ ⟨*VarInit*⟩*]
‘`PROGRAM:`’ ⟨*ProcExp*⟩
⟨*Goal*⟩
⟨*Process*⟩$^+$

⟨*Process*⟩ ::= ‘`PROCESS`’ ⟨*ID*⟩ ‘`:`’
‘`VARS:`’ ⟨*VarDeclTS*⟩$^+$
[‘`INIT:`’ ⟨*VarInit*⟩*]
‘`TRANS:`’ *Stmt*$^+$

⟨*VarDeclTS*⟩ ::= ‘`const`’ ⟨*Type*⟩ ⟨*ID*⟩ ‘`=`’ ⟨*Constant*⟩ ‘`;`’
| [‘`nondet`’] ⟨*Type*⟩ ⟨*ID*⟩ [‘`:`’ ⟨*Range*⟩] ‘`;`’

⟨*VarDeclPM*⟩ ::= ‘`const`’ ⟨*Type*⟩ ⟨*ID*⟩ ‘`=`’ ⟨*Constant*⟩ ‘`;`’
| [‘`nondet`’] ⟨*Type*⟩ ⟨*ID*⟩ [‘`:`’ ⟨*Range*⟩] ‘`;`’
| ‘`proc`’ ⟨*ID*⟩ ‘`=`’ ⟨*ID*⟩ ‘`;`’

⟨*VarInit*⟩ ::= ⟨*BoolExp*⟩ ';' ⟨*VarInit*⟩
  | ⟨*BoolExp*⟩ ';'
  | ⟨*ID*⟩ '`=`' ⟨*Exp*⟩ '`;`' ⟨*VarInit*⟩
  | ⟨*ID*⟩ '`=`' ⟨*Exp*⟩ '`;`'

⟨*Exp*⟩ ::= ⟨*BoolExp*⟩ | ⟨*NumExp*⟩

⟨*BoolExp*⟩ ::= ⟨*UnOpBool*⟩ ⟨*BoolExp*⟩
  | ⟨*BoolExp*⟩ ⟨*ConnectorOp*⟩ ⟨*BoolExp*⟩
  | ⟨*NumExp*⟩ ⟨*ComparisonOpArith*⟩ ⟨*NumExp*⟩
  | ⟨*BoolExp*⟩ ⟨*ComparisonOp*⟩ ⟨*BoolExp*⟩
  | ⟨*NumExp*⟩ ⟨*ComparisonOp*⟩ ⟨*NumExp*⟩
  | '`(`' ⟨*BoolExp*⟩ '`)`'
  | ⟨*BoolTerm*⟩

⟨*NumExp*⟩ ::= ⟨*NumExp*⟩ ⟨*ArithOp*⟩ ⟨*NumExp*⟩
  | '`(`' ⟨*NumExp*⟩ '`)`'
  | ⟨*ArithTerm*⟩

⟨*ProcExp*⟩ ::= ⟨*ID*⟩ ⟨*ProcOp*⟩ ⟨*ProcExp*⟩ | ⟨*ID*⟩

⟨*Stmt*⟩ ::= ['`?`' ⟨*BoolExp*⟩ '`->`'] ⟨*IDEnum*⟩$^+$ '`=`' ⟨*ExpEnum*⟩$^+$

⟨*Goal*⟩ ::= '`REACH:`' ⟨*BoolExp*⟩
  | '`INF:`'
  | '`SAFE:`' ⟨*BoolExp*⟩
  | '`LIVENESS:`' ⟨*BoolExp*⟩
  | '`DEADLOCK:`'

⟨*BoolTerm*⟩ ::= '`false`' | '`true`' | ⟨*ID*⟩

⟨*ArithTerm*⟩ ::= ⟨*ID*⟩ | ⟨*Num*⟩

⟨*IDEnum*⟩ ::= ⟨*ID*⟩ | ⟨*ID*⟩ '`,`' ⟨*IDEnum*⟩

⟨*ExpEnum*⟩ ::= ⟨*Exp*⟩ | ⟨*Exp*⟩ '`,`' ⟨*ExpEnum*⟩

⟨*UnOpBool*⟩ ::= '`!`'

⟨*ConnectorOp*⟩ ::= '`&`' | '`|`'

⟨*ComparisonOpArith*⟩ ::= '`<`' | '`<=`' | '`>`' | '`>=`'

⟨*ComparisonOp*⟩ ::= '`==`'

⟨*ArithOp*⟩ ::= '`+`' | '`-`' | '`*`' | '`/`'

⟨*ProcOp*⟩ ::= '`||`'

⟨*Constant*⟩ ::= ⟨*Num*⟩ | ⟨*ID*⟩

⟨*Range*⟩ ::= ⟨*Constant*⟩ '`..`' ⟨*Constant*⟩

⟨*Num*⟩ ::= ['`-`'] *digit*$^+$

⟨*Type*⟩ ::= '`int`' | '`bool`'

⟨*ID*⟩ ::= *alpha*$^+$

⟨*AccessorID*⟩ ::= ⟨*ID*⟩ '`.`' ⟨*AccessorID*⟩ | ⟨*ID*⟩

References

Alpern, B., & Schneider, F. B. (1985). Defining liveness. *Information processing letters*, *21*(4), 181–185.

Barnat, J., Brim, L., Havel, V., Havlek, J., Kriho, J., Leno, M., ... Weiser, J. (2013). DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs. In *Computer Aided Verification (CAV 2013)* (Vol. 8044, pp. 863–868). Springer.

Barrett, C., Stump, A., & Tinelli, C. (2010). *The satisfiability modulo theories library (smt-lib).* Retrieved from `www.SMT-LIB.org`

Behrmann, G., David, A., & Larsen, K. G. (2004). A tutorial on uppaal. In *Formal methods for the design of real-time systems* (pp. 200–236). Springer.

Behrmann, G., David, A., Larsen, K. G., Hakansson, J., Petterson, P., Yi, W., & Hendriks, M. (2006). Uppaal 4.0. In *Quantitative evaluation of systems, 2006. qest 2006. third international conference on* (pp. 125–126).

Biere, A., Cimatti, A., Clarke, E. M., Strichman, O., & Zhu, Y. (2003). Bounded model checking. *Advances in computers*, *58*, 117–148.

Cimatti, A., Clarke, E., Giunchiglia, F., & Roveri, M. (1999). Nusmv: A new symbolic model verifier. In *Computer aided verification* (pp. 495–499).

Clarke, E., Biere, A., Raimi, R., & Zhu, Y. (2001). Bounded model checking using satisfiability solving. *Formal Methods in System Design*, *19*(1), 7–34.

Dutertre, B., & De Moura, L. (2006). *The yices smt solver.* `http://yices.csl.sri.com/tool-paper.pdf`.

McMillan, K. L. (1993). *Symbolic model checking.* Springer.

Mishchenko, A. (2001). An introduction to zero-suppressed binary decision diagrams. *Jun*, *8*, 1–15.

Tzoref, R., Matusevich, M., Berger, E., & Beer, I. (2003). An optimized symbolic bounded model checking engine. In *Correct hardware design and verification methods* (pp. 141–149). Springer.

Vaandrager, F., & Verbeek, F. (2014, June). Recreational formal methods: Designing vacuum cleaning trajectories.

Yin, L., He, F., & Gu, M. (2013). Optimizing the sat decision ordering of bounded model checking by structural information. In *Theoretical aspects of software engineering (tase), 2013 international symposium on* (pp. 23–26).