TKK Reports in Information and Computer Science
Espoo 2009 TKK-ICS-R17

# MODEL CHECKING PSL SAFETY PROPERTIES

Tuomas Launiainen

TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY
TECHNISCHE UNIVERSITÄT HELSINKI
UNIVERSITE DE TECHNOLOGIE D'HELSINKI

TKK Reports in Information and Computer Science
Espoo 2009 TKK-ICS-R17

# MODEL CHECKING PSL SAFETY PROPERTIES

Tuomas Launiainen

Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Information and Computer Science

Teknillinen korkeakoulu
Informaatio- ja luonnontieteiden tiedekunta
Tietojenkäsittelytieteen laitos

Distribution:
Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Information and Computer Science
P.O.Box 5400
FI-02015 TKK
FINLAND
URL: http://ics.tkk.fi
Tel. +358 9 451 1
Fax +358 9 451 3369
E-mail: series@ics.tkk.fi



ISBN 978-952-248-041-5 (Print)
ISBN 978-952-248-042-2 (Online)
ISSN 1797-5034 (Print)
ISSN 1797-5042 (Online)
URL: http://lib.tkk.fi/Reports/2009/isbn9789522480422.pdf

TKK ICS
Espoo 2009

**ABSTRACT:** Model checking is a modern, efficient approach to gaining confidence of the correctness of complex systems. It outperforms conventional testing methods especially in cases where a high degree of confidence in the correctness of the system is required, or when the test runs of the system are difficult to reproduce accurately. In model checking the system is verified against a specification that is expressed in a formal specification language. The main challenges are that the process requires quite a lot of training, experience, and computing power. Recent developments in the field of model checking address all of these issues.

Safety properties are a subset of formal specifications that are simpler to verify than formal specifications in the general case. Additionally, safety properties can be used to improve conventional testing methods by observing the behaviour of the system at runtime and reporting the detected violations of the safety properties, which are more expressive than the properties used with conventional testing. In model checking, recognising and separately verifying safety properties can give faster verification times than just processing all properties without a specialised algorithm for safety properties.

One of the problems related to model checking is creating specifications that are meaningful to both humans and to model checking tools. One specification language that focuses on this problem is the IEEE 1850 standard Property Specification Language (PSL). It is not as widely supported by academic model checking tools as linear temporal logic (LTL) or computation tree logic (CTL), but it has many features that make writing specifications easier for engineers.

This work describes a method for verifying PSL safety properties by converting them to transducers, a variant of symbolic finite automata. The semantics in the most current proposal for the revised PSL standard is reviewed, and additional operators are introduced for formula rewriting. The main contributions of this work are the PSL translation and its proof of correctness with respect to the presented semantics, and a prototype implementation of an algorithm for model checking PSL safety properties. The implementation is built on top of the NuSMV model checker, a modern, open-source tool that previously had little support for PSL. Experiment results are presented to show the feasibility of the implemented approach.

**KEYWORDS:** model checking, PSL, safety properties, NuSMV

# CONTENTS

# LIST OF FIGURES

## LIST OF ABBREVIATIONS

| | |
|---|---|
| PSL | Property Specification Language |
| IEEE | Institute of Electrical and Electronics Engineers |
| LTL | Linear Temporal Logic |
| NuSMV | A free, open source model checker |
| $AP$ | Atomic Proposition |
| CTL | Computation Tree Logic |
| BDD | Binary Decision Diagram |
| DAG | Directed, Acyclic Graph |
| $\wedge$ | Logical and |
| $\vee$ | Logical or |
| $\neg$ | Logical negation |
| $\Leftrightarrow$ | Equivalence |
| $\Rightarrow$ | Implies |
| $\subset$ | Proper subset of |
| $\subseteq$ | Subset of |
| $\times$ | Cartesian product |
| $\in$ | Member of |
| $\cup$ | Union |
| $\cap$ | Intersection |
| $\setminus$ | Set subtraction |
| $\exists$ | There exists |
| $\forall$ | For all |
| $\top$ | True |
| $\bot$ | False |
| SERE | Sequential Regular Expression |
| **X** | Next |
| **G** | Globally |
| **F** | Finally |
| **U** | Until |
| **R** | Releases |
| $\mapsto$ | Tail implication |
| $\diamond\!\!\rightarrow$ | Tail conjunction |
| $*$ | Kleene closure |
| $\cdot$ | Concatenation |
| $\circ$ | Concatenation with an overlap |

# 1 INTRODUCTION

A crucial part of engineering complex systems, such as computer programs, electrical circuits, and embedded electronic systems, is verifying that the systems works as intended. For example, modern computer programs often have to deal with an environment where many processes are run concurrently and where they communicate and co-operate with each other. In such cases the traditional approach of testing the system, where inputs are given to the system and then the observed behaviour is compared to the intended behaviour, can be insufficient to gain enough confidence in the correctness of the tested system. This is because the order in which the program steps of different processes are executed can vary greatly between test runs, and this can affect the behaviour of the system as a whole. Thus, a fault that only occurs with a certain scheduling of the processes can be next to impossible to detect with traditional testing.

Model checking [12] is a tool that can cover some of the cases where traditional testing is insufficient. Like testing, its primary function is to gain confidence that the system works as intended. As such, it can partially replace testing, but mostly the two approaches complement each other. The term model checking refers to a collection of techniques but the common factor in them is that the system is represented using a formal mathematical model. The model is combined with properties that are created from the intended behaviour of the system. The specified properties must also be expressed formally. The model and properties are then given to a model checking tool that goes through every possible behaviour of the model and verifies that none of them break the specified properties.

Model checking can be an expensive procedure because of its nature. Expressing both the model of the system and the properties formally requires great care and expertise. Mistakes in either can obviously produce incorrect results. Moreover, the verification of models with model checking tools requires a lot of computing resources. More specifically, model checking suffers from the so called state explosion [30]: when the model is constructed from many concurrent components, as is often the case with concurrently executed programs, the resulting model can have an exponential number of reachable states with respect to the number of components. Since the tool must go through every possible behaviour of the model, a lot of computing power (time) and memory are required.

Model checking has many advantages as well. It can detect errors that would be very difficult to notice with other methods, for example in concurrent software. The properties that can be verified are more expressive than with traditional testing, depending on the formalism used to express them. For example properties that require something to happen infinitely often, or properties that require that some alternative is always available. Model checking can also be used for prototyping a design in cases where creating a model is easier than creating an actual implementation. Additionally, because every possible behaviour of the model is checked, the result is absolutely certain, unless the model checking tool itself has serious errors. Thus, model checking can be used in making formal proofs of the system: if the model an prop-

erties are proven correct, a very high degree of confidence can be gained that the system itself is correct.

For model checking to be applicable and easier to deploy in more cases the model checking tools need to be developed further. The running time and memory requirements can be lowered with better algorithms, tools can be made to automatically extract models from programs, and modelling and specification languages can be made easier to use for engineers with different backgrounds. This work introduces a new translation from the Property Specification Language (PSL) to symbolic finite state automata. PSL was developed from IBM's Sugar [4], a specification language that was created to be easier to use by engineers than the specification languages that precede it. More recently it has been made into an IEEE standard [3]. The correctness of the developed translation is proven, and with it a model checking algorithm for PSL is implemented to the NuSMV [8] model checker: a fast, free, and open-source tool. Some experiments are performed with the modified NuSMV model checker, and the results are analysed. Unfortunately, a direct comparison of the algorithm to other PSL model checking algorithms could not be done, because of the limited support for PSL in the latest version of NuSMV, v.2.4.3 at the time of writing.

## 1.1 RELATED WORK

Safety properties have been singled out for separate inspection in several papers. Complexity issues of model checking safety properties with finite state automata are covered in [18], which also contains a classification of safety properties into intentionally safe, accidentally safe, and pathologically safe. The paper also describes a way to construct finite state automata that can be used to observe safety properties, much like in this work. The construction is further studied and refined in [17]. The approach to constructing the automata in these papers differs from the one presented here, however, and only LTL formulae are considered in them.

A translation of LTL safety properties into finite state automata, following the approach of [18] is covered in [20] and [21]. Evaluation of LTL on finite paths in the context of bounded model checking is done in [6]. The approach in that paper is to encode the model checking problem to a propositional satisfiability (SAT) problem and then use a SAT-solver to solve the encoded instance. That approach needs a semantics for LTL formulae with finite paths, which is similar to the semantics for finite paths in this work. A translation from LTL to SMV code is done in [11]. The implementation technique in that paper is similar to the one used in this work but they only consider infinite paths unlike this work. Monitoring safety properties expressed in the past time fragment of LTL are used as a testing aid in [15].

Other work has also been done to translate PSL properties into other forms of specifications that are easier to model check. The one that is probably most similar to the approach in this work is presented in [26]. That paper also translates PSL to symbolic automata that are similar to the ones in this work. We are not aware of an implementation of the approach of [26]. Those automata are for infinite words, however, and the translation is for the full

PSL, without special consideration for safety properties. A translation from the full PSL to non-deterministic symbolic Büchi-automata for NuSMV is presented in [9], where PSL formulae are divided into an LTL part and a part encoded as symbolic Büchi-automata. An automata translation for the safety simple subset of PSL is presented in [27]. The idea behind that paper is also quite similar to the one in this work but that paper restricts the supported subset of PSL syntactically to rule out non-safety formulae. Moreover, their algorithm does not seem to be implemented in a freely available model checking tool. This work, on the other hand, supports the full PSL language, but is only able to detect finite counterexamples to the properties. A method of translating a core set of PSL properties to Büchi-automata is presented in [5], but safety properties are not considered separately. A translation of PSL into the $\mu$-calculus is presented in [23]. The expressiveness of fragments of PSL is studied in [19], along with the complexity of model checking these fragments.

PSL safety properties, the semantics of PSL with finite paths, and model checking PSL safety properties is also discussed in [10]. That paper also restricts the syntax of PSL to rule out non-safety formulae. Also, we are not aware of a freely available implementation of its semantics. An alternative way of defining the semantics of PSL for finite paths is also discussed in [14]. The concerns of these two papers are addressed in a recent proposal for a revised PSL standard [1], the semantics of which is used in this work. Automata formalisms similar to the one in this work are presented in e.g. [25] and [28].

The rest of this work is structured as follows. Chapter 2 goes over the basic concepts of model checking, Chapter 4 introduces the basics of the PSL language, and Chapter 5 formalises the semantics of the language. Chapter 6 discusses the use of the semantics in Chapter 5 for finite paths. In Chapter 7 transducers, a formalism similar to symbolic finite state automata is introduced, and a method for compositionally constructing transducers from PSL formulae is described, along with proofs of its correctness. Chapter 8 describes the prototype implementation of the translation described in Chapter 7 into the open-source model checker NuSMV. Chapter 9 describes the test setup that was used to run experiments on the implementation, and presents the results and their interpretation. Finally, Chapter 10 concludes.

# 2 MODEL CHECKING

The process of model checking usually starts with a decision of what part of the system will be checked, and what properties that part should satisfy. Then, an abstract model is created from that part of the system. The properties that are checked for in the system determine what parts should be abstracted away, and what details must be included in the model.

A *model* in the context of model checking is a precise formal description of the behaviour of the system. Usually the design of the system is what needs to be validated, and the model is built by hand from the high level design. Some small critical parts of the actual implementation of the system can also be validated but it is rarely feasible to validate the whole implementation of the system. The model can also be automatically generated from the design or the implementation but usually that requires that the description of the design or implementation is done with that purpose in mind.

In this work we use a common formalism for models: the Kripke structure [12]. A Kripke structure is a system that consists of finite number of states, some of which are *initial*, a *transition relation*, and a function that labels each state with the *atomic propositions* that hold in the state. When the system starts, it is in one of its initial states. The next state is one of the states to which there is a transition from the current state in the transition relation. The transition relation is required to be total, meaning that from every state there is at least one transition originating from it. Formally, a Kripke structure $M$ over a set of atomic propositions, $AP$, is a quadruple $M = (S, S_0, R, L)$, where:

- $S$ is a finite set of states,
- $S_0 \subseteq S$ is the set of initial states,
- $R \subseteq S \times S$ is the transition relation, and
- $L : S \to 2^{AP}$ is the labelling function, that associates each state in $S$ with the atomic propositions that hold in it.

A *path* of the Kripke structure is a finite or infinite sequence of states, $\pi = s_0 s_1 s_2 \ldots$ such that for all $i \geq 0$, $(s_i, s_{i+1}) \in R$ holds. If $s_0 \in S_0$, then $\pi$ is an *initialised path*.

This kind of formalism for models is extremely simplistic, and on purpose. Creating models directly as Kripke structures is only seldom feasible but creating model checking algorithms for them is convenient. Consequently the modelling work is often done in a more expressive language that is then converted to a Kripke structure by the model checking tool.

*Properties* are the other part of the model checking process. They must also be represented formally, usually in some formal specification language. The purpose of a property is to tell what the model is not allowed to do, and what it must be able to do. Typical formal specification languages are the Computation Tree Logic (CTL), and the Linear Temporal Logic (LTL) [12]. Property Specification Language (PSL), is a more modern addition and an IEEE 1850 standard [3].

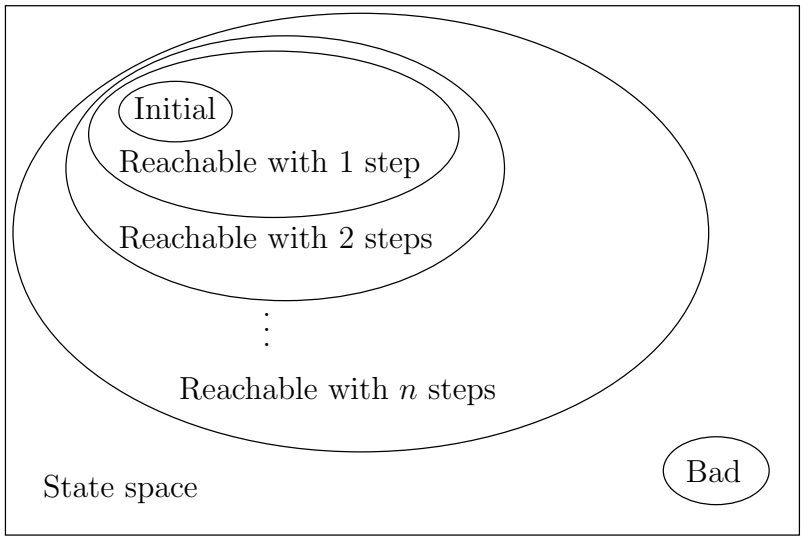



Figure 2.1: An illustration of invariant model checking. The outermost box represents all the possible states of the model.

## 2.1 INVARIANT MODEL CHECKING

A simple version of the model checking process is checking whether *invariant properties* hold. Intuitively, an invariant is something that must remain true in all reachable states of the model. An example of an invariant property is that in a bank account the amount of money deposited minus the amount withdrawn must equal the balance. From the invariant the set of bad states, i.e. those in which the invariant is broken, can be computed. Then, the model is inspected to see if any bad state is reachable from the initial state. The following process is used to do this check:

1. Let $X$ be a set of states, initially set to $S_0$, the set of initial states of the Kripke structure.
2. Let $succ(X)$ be the set of states that are reachable from $X$ with a single transition, i.e. $succ(X) = \{s \mid \exists t \in X : (t, s) \in R\}$.
3. Let $B$ be the set of bad states, i.e. the set of states in which the invariant is broken.
4. If $B \cap X \neq \emptyset$, report failure and terminate the algorithm.
5. If $X \cup succ(X) = X$, report success and terminate the algorithm.
6. Set $X$ to $X \cup succ(X)$ and proceed with step 4.

Figure 2.1 illustrates this process.

## 2.2 SYMBOLIC MODEL CHECKING

The straightforward way of handling states in model checking algorithms is by storing each of them explicitly in memory. Doing this often leads to a

problem called *state explosion* [30], however, even with moderately complex models. State explosion happens when the state-space is exponential in size with respect to the description of the model. This is the case, for example, when the model consists of multiple concurrently executing components, and there can be exponentially many combinations of the local states of the components. With state explosion, the main memory of computers quickly becomes too small to accommodate the state space, and the hard drives are too slow in practice to be used in storing the state space. The solution to the problem is to have some compact way of representing the sets of states that arise in model checking. A popular technique that achieves this is called symbolic model checking, and is discussed in detail in [24] and [12].

### 2.2.1 Binary Decision Diagrams

A popular way of way of representing the state space in symbolic model checking is with Binary Decision Diagrams. Binary Decision Diagrams (BDD) [7], are a way of describing boolean functions. They are rooted, directed, acyclic graphs (DAGs), that have two kinds of nodes: non-terminals and terminals. The terminal nodes denote a boolean value, while non-terminal nodes denote variables with boolean domains. Terminal nodes have no children, while non-terminals have exactly two: a *low* and a *high* child. The order of variables in the BDD is fixed: if a variable comes after another in one path from the root to a terminal, they cannot appear in reversed order along another path.

Intuitively, a BDD is interpreted as a boolean function in the following way: start with the root, and go to the *high* child if the variable denoted by the node is true, or the *low* child otherwise. This choice is repeated on every non-terminal node encountered this way, until a terminal is reached. The value denoted by the terminal is the value of the function with the variable values for the followed path.

Let $V$ be a set of boolean variables. A *truth assignment* of $V$ is a mapping of the variables in $V$ to truth values, i.e. $a : V \to \{0, 1\}$. A boolean function $f$ over the set of variables $V$ maps the truth assignments of $V$ to truth values. One way to describe $f$ as a BDD is to have a the full tree with separate nodes for all the possible truth assignments. The size of such a tree is $2^{\mathcal{O}(|V|)}$. Figure 2.2 illustrates this set-up.

This way of representing boolean functions is obviously inefficient, since a simple truth table would be smaller. A very straightforward improvement is to use only one instance each of the "true"-node and the "false"-node. This removal of redundancy is extended to variable nodes. If the *low*-child and the *high*-child of a variable node are the same, the variable node can be removed, and all nodes that point to it are modified to point to its child instead. Also, sub-graphs that have identical structure are merged into one. This can be done efficiently in bottom-up order one node at a time: if two nodes denote the same variable and have the same *low* and *high* children, they are merged. When no new node merging or removal can be done, the BDD is in its reduced form. Figure 2.3 shows an unreduced BDD, and Figure 2.4 shows the same BDD in its reduced form.

BDDs that have a fixed variable ordering and are reduced as described

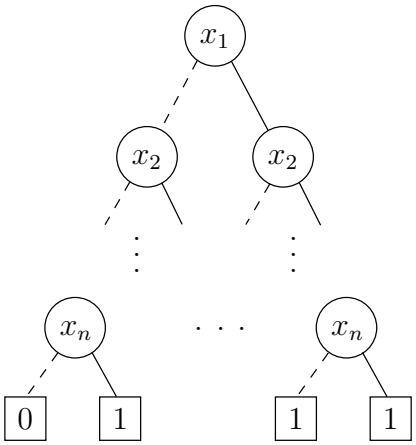



Figure 2.2: A Binary Decision Diagram representing some boolean function over a set of $n$ variables. A dashed line leads to the *low* child while a solid line leads to a *high* child.

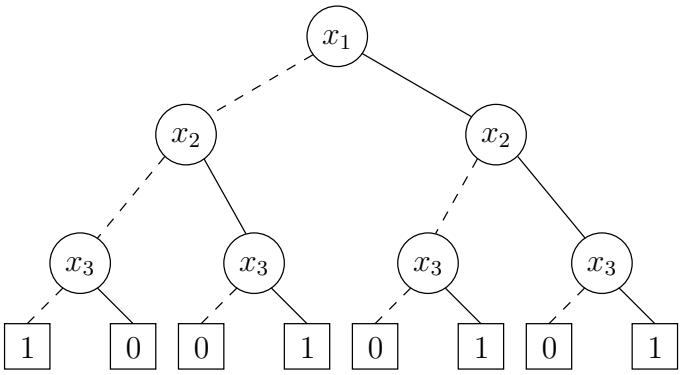



Figure 2.3: A BDD in its unreduced form.

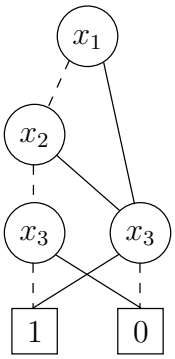



Figure 2.4: A BDD in its reduced form.

above are called Reduced, Ordered Binary Decision Diagrams (ROBDD). ROBDDs are a *canonical* way of representing boolean functions, meaning that if two ROBDDs represent the same function, they are isomorphic, i.e. they have the same structure. This in turn means that checking if a boolean function is valid or satisfiable can be done trivially by checking if the ROBDD that represents the function is a single "true" or "false" terminal node. In this text all BDDs from now on refer to ROBDDs. The properties of BDDs and algorithms for manipulating them are covered e.g. in [7].

Utilising BDDs in model checking is conceptually fairly straightforward. Since they are a convenient way to describe boolean functions, all that is needed is a way to use boolean functions to describe the parts of a Kripke structure.

- A state of the Kripke structure is encoded as a truth assignment of a set of variables $V$.
- A set of states $S'$ can be coded as a boolean function over the set of variables $V$. The binary function $f_{S'}$ describing $S'$ returns true if and only if the truth assignment represents a state in $S'$.
- The transition relation $R \subseteq S \times S$ can also be coded as a boolean function $f_R$ over a union of two sets of variables $V$ and $V'$. Truth assignments of $V$ are used to represent starting states for transitions and truth assignments of $V'$ are used to represent ending states. Let $a$ be a truth assignment of $V$ and $a'$ be a truth assignment of $V'$. Then $f_R$ returns true if and only if the pair of states represented by $a$ and $a'$ belongs to the transition relation $R$.

A few operations on boolean functions need to be defined before a model checking algorithm that utilises them can be described.

- The union of two sets is equivalent to the disjunction of the boolean functions that describe them. A disjunction of two boolean functions $f_1$ and $f_2$ over the sets of variables $V_1$ and $V_2$ is a boolean function $f$ over the set of variables $V_1 \cup V_2$, s.t. $f$ returns true for a truth assignment if and only if at least one of $f_1$ and $f_2$ would return true for that assignment.
- The intersection of two sets is equivalent to the conjunction of the boolean functions that describe them. A conjunction of two boolean functions $f_1$ and $f_2$ over the sets of variables $V_1$ and $V_2$ is a boolean function $f$ over the set of variables $V_1 \cup V_2$, s.t. $f$ returns true for a truth assignment if and only if both $f_1$ and $f_2$ would return true for that assignment.
- Using the transition relation requires two operations on boolean functions: existential quantification and renaming. Existential quantification is denoted by $\exists V' \,.\, f_1$, where $f_1$ is a boolean function over the set of variables $V_1$ and $V' \subseteq V_1$. The result of the operation is a boolean function $f$ over the set of variables $V_1 \setminus V'$, s.t. $f$ returns true for a truth assignment $a$ if and only if there exists a truth assignment $a'$ of $V'$ such

that $f_1$ would return true for the combined truth assignment of $a$ and $a'$. This can be thought of as saying the variables in $V'$ are not of importance: if there is a way to set them that makes the function true, then the value is true.

Renaming is denoted by $f_1[V'/V]$, where $f_1$ is a boolean function over the set of variables $V_1$, $|V| = |V'|$, and $V' \subseteq V_1$. The result of the operation is a boolean function $f$ over the set of variables $(V_1 \setminus V') \cup V$ s.t. the variables in $V'$ are replaced with the variables in $V$. $f$ returns true for a truth assignment $a$ if and only if replacing the variables from $V'$ with the variables from $V$ in $a$ would result in a truth assignment for which $f_1$ would return true.

The above operations can be used to compute the set of states that is reachable from some set of states $S'$ with a single step, for example. Let $f_{S'}$ be the boolean function over $V$ that represents $S'$, let $f_R$ be the boolean function over $V \cup V'$ that represents the transition relation. The states that are reachable from $S'$ with a single step can be computed with the operation $(\exists V\,.\,(f_R \wedge f_{S'}))[V'/V]$. First, the transition relation is intersected with the set of states $S'$: this restricts it to the transitions whose starting state is in $S'$. After that existential quantification is used to remove the variables that are used for encoding the starting state, and finally the remaining variables are renamed to get an encoding for the desired set of states.

For more details about the above operations, their implementations with BDDs, and about BDDs in general, see [7] or [24]. With these operations, an algorithm for symbolic invariant model checking can be described. The algorithm, shown in Figure 2.5, gets as parameters the boolean function encoding of the set of initial states of the Kripke structure, the boolean function encoding of the transition relation, and the boolean function encoding of the set of bad states. It returns FAILURE if some bad state is reachable from the initial states, and SUCCESS otherwise. The auxiliary function *succ* computes the states that are reachable with a single step from the set it gets as a parameter, using the transition relation that it gets as the other parameter. $V$ is the set of variables that is used for encoding states and the starting states of a transition, and $V'$ is the set of variables that is used for encoding the ending states of transitions.

## 2.3 SAFETY PROPERTIES

Safety properties are a special class of properties that state that require that some bad thing can never happen. Intuitively, the bad thing is something that can be immediately noticed from an observed behaviour. More formally, a *safety language* [18] is a set of finite and infinite paths, for which the following holds: for every path not in the safety language, there exists a *bad prefix*. A bad prefix is a finite path that cannot be extended in any way to produce a path that is in the language. A bad prefix can be thought of as a counterexample to a property. A safety property is a property that can be expressed as a safety language.

```
succ(f, f_R) {
   return (∃V . (f_R ∧ f))[V'/V]
}

CheckInvar(f_S0, f_R, f_B) {
   f ← false
   f' ← f_S0
   while (f' ≠ f) {
      f ← f'
      f' ← f' ∨ succ(f', f_R)
      if (f' ∧ f_B ≠ false) {
         return FAILURE
      }
   }
   return SUCCESS
}
```

Figure 2.5: An algorithm for symbolic invariant model checking

Safety properties are an important class of properties in a few ways. Firstly, model checking with safety properties is a simpler task than general model checking, as discussed e.g. in [18] and [21]. This makes developing specialised algorithms for handling them attractive. Secondly, safety properties can be monitored at runtime by adding instrumentation to a system that reports violations of the safety property. This follows from the finiteness of the counterexamples: when the property is broken, a finite execution is enough to report its violation, no knowledge about possible future behaviour is needed.

## 2.4 MODEL CHECKING WITH OBSERVERS

Observers in the context of model checking are finite state machines that are used to detect counterexamples to a property. For details, see [11]. They have a set of bad states, expressed as an invariant property, and are given an execution path as input. The observer enters a bad state if and only if the execution so far has broken the property that is being checked. This method of operation of course means that only safety properties can be exhaustively verified by observers, since the observer enters a bad state after a finite number of steps or not at all, and thus the counterexample for the property must be finite. Some finite counterexamples to non-safety properties can be detected as well. Figure 2.6 illustrates the components and their inputs in model checking with the observer method.

While the observer method does suffer from being limited to finite counterexamples, it has a reduced complexity of verification: it is sufficient to check for the reachability of bad states. Observers can also be used in testing to bring the expressiveness of properties from model checking to the testing methodology by running the observer together with the system and having it

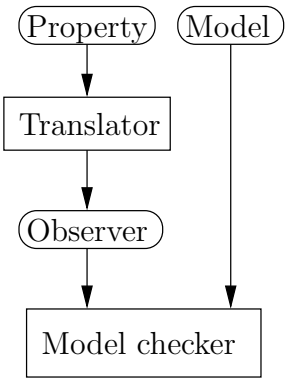



Figure 2.6: Model checking with the observer method. Ovals represent inputs to components, and boxes represent the components.

report a fault when a bad state is reached, as is done in [15]. The main advantage to conventional testing is that more complex properties can be tested for more easily.

# 3 NUSMV MODEL CHECKING TOOL

NuSMV [8] is a model checking tool and framework released under an open source license. It was originally designed as a BDD-based model checker, but has since been extended to incorporate propositional satisfiability (SAT) -based model checking techniques. In the process of adding the SAT-based back end, NuSMV has been redesigned to make adding new algorithms and model checking methods possible. The goal of the NuSMV project is to have a free tool with support for state of the art techniques, as well as a framework for adding and testing experimental techniques.

## 3.1 INPUT LANGUAGE

NuSMV uses an extension of the SMV language [16]. The basic building blocks of SMV models are *modules*. They are templates for finite state machines, and define a set of state variables, parameters, and restrictions on behaviour. The state of the whole model consists of instantiated modules and their internal states. Module instances can behave synchronously, updating their state all at once, or as asynchronous processes, in which case one instance at a time is non-deterministically chosen to update its state.

A module consists of a list of parameters, a list of variables, and a set of constraints for them. The variables can have domains of boolean values, arrays, integer ranges, and module instances, among others, but this work focuses only on variables with boolean domains. The constraints limit which state changes are possible, thus inducing a Kripke structure. Any unrestricted variables can take any value in their domain.

The parameters of a module are used to convey signals between module instances. A parameter can be used like a state variable in the constraints, but the effect carries on to other module instances, allowing them to communicate with each other.

Variables are declared in `VAR`-blocks in a module. A variable declaration consists of the name and type of the variable. Multiple variable declarations can be included in a `VAR`-block, and a module can contain multiple `VAR`-blocks. A typical `VAR`-block could look something like:

```
VAR
  a : boolean;
  b : boolean;
```

This block declares two boolean variables, `a` and `b`.

Constraints can be added with `INIT`-blocks, `INVAR`-blocks, `TRANS`-blocks, and `ASSIGN`-blocks. In this work we focus only on `INIT`-, `INVAR`-, and `TRANS`-blocs. The `INIT`-block contains a boolean expression that must be true in the initial state of the module. If multiple `INIT`-blocks are present, they all must be true in the initial state. A typical `INIT`-block could look like:

```
INIT a & !b
```

This block states that in every initial state, `a` must hold and `b` must not hold.

The `INVAR`-block contains a boolean expression that must be true in every state of the module. Again, multiple `INVAR`-blocks must all be true. For example, the block:

```
INVAR
  a | b | c
```

states that at least one of the variables must be true in every state.

The `TRANS`-block also contains a boolean expression, but such an expression is true in every state change of the system. Boolean expressions inside a `next`-expression must be true after the state change, whereas boolean expressions outside them are true before the state change. A module can contain multiple `TRANS`-blocks, in which case they must all be true in every state change. A typical `TRANS`-block could look like:

```
TRANS
  a <-> next(!a)
```

This block declares that `a` is an oscillating variable: it is true in exactly every other state. The constraints in `TRANS`-blocks do not hold in the case where there is no next state. `INVAR`-constraints must be used in such cases.

NuSMV modules can include many types of verifiable specifications, including CTL, LTL, PSL, and invariant properties. They are declared in `SPEC`-, `LTLSPEC`-, `PSLSPEC`-, and `INVARSPEC`-blocks, respectively. Each type of temporal logic specification block contains a formula written in the corresponding logic, and the invariant specification contains a boolean expression that must hold in every reachable state of the module.

The following example illustrates the use of `INIT` and `TRANS` blocks, as well as usage of modules:

```
MODULE main
VAR
  a : step_delay(0, b.out);
  b : step_delay(1, a.out);

MODULE step_delay(initial, signal)
VAR
  out : boolean;
INIT
  out = initial
TRANS
  next(out) = signal
```

The `step_delay`-module is a state machine that has two parameters: the initial value of the output and the signal that it gets its subsequent output values from. It introduces a delay of a single step between the signal and its output. The `main`-module instantiates two synchronous copies of the `step_delay`-module and sets the input signal of each to the output of the other. The end result is a system where `a.out` and `b.out` take turns in being true.

# 4 INTRODUCTION TO PSL

The Property Specification Language (PSL) [3], was developed from Sugar [4], a specification language designed by IBM. Sugar, and by extension PSL, is focused on creating a specification language that is easy to use by engineers. It has many features for circuit design and analysis, but can be used in any type of environment where formal specification languages are used. The previous versions of the standard suffer from slight inconsistencies as detailed for example in [10], but a new version of the PSL standard is under development. This work uses the semantics from [1], which is a proposal for the next version of PSL. A more thorough introduction to PSL can be found in [13].

PSL uses atomic propositions as basic elements of statements. The specification also includes the boolean layer, which includes more complex expressions that have boolean values: bit vector expressions, numerical expressions, and string expressions etc. Here, however, we omit the boolean layer for simplicity.

Basic PSL expressions can be formed and combined with the basic boolean logic operators: $\wedge$, $\vee$, and $\neg$, but the real power of PSL is with the temporal layer. The temporal layer contains operators that refer to the future or past. They are the heart of the language, and are used to specify behaviour with respect to subsequent states of the system.

The core of PSL, called the Foundation Language in the specification, contains temporal operators that make statements about a single path. A path is a sequence of states that represent one possible way in which the system can behave. This means that the temporal operators in the Foundation Language cannot make statements about what states are available as a choice for the next state. This is possible in the Optional Branching Extension of the language, which is not covered in this work. A system satisfies a property if and only if the property holds on every path of the system. The temporal operators in the Foundation Language can be roughly divided into two parts: Sequential Regular Expressions (SEREs), that are discussed in Section 4.1, and LTL-like operators, that are discussed in Section 4.2.

## 4.1 SEQUENTIAL REGULAR EXPRESSIONS

Sequential Regular Expressions (SERE), are very much like other forms of regular expressions that are used in various computer-related tasks. A common use for regular expressions is to describe patterns against which it is efficient to test a sequence of characters. SEREs, on the other hand, are used to describe a pattern that can be tested for in a path. They also contain some operators that are not available in typical forms of regular expressions.

The benefits of SEREs compared to the LTL-like temporal operators are that engineers that already know some form of regular expressions can use that previous knowledge with SEREs. Moreover, SEREs can be used to specify some things that can not be specified in LTL. They can only specify finite patterns, however, so specifying infinite behaviour must be done with the LTL-like operators.

When deciding if a path matches the pattern described by a SERE, the basic interpretation is to see if the beginning of the path matches the pattern. A SERE by itself cannot make statements about things that must happen infinitely often, or state that some action must always be followed by something else in an infinite execution. This limits the usability of SEREs by themselves, and therefore they are often used as parts of a larger property that uses the other temporal operators in PSL.

Paths in this chapter are presented as sequences of atomic proposition groups. The groups are surrounded by braces, and contain the atomic propositions that hold in them. Some groups also have negated atomic propositions, which means that the atomic proposition does not hold in that group, and that it is somehow significant. For example, the path

$$\{\mathbf{a}\}, \{\mathbf{a}, \neg b\}$$

consists of two states, where $a$ holds in both states, but the second state stresses that $b$ does not hold in it. The atomic propositions that hold are in **bold**, while those that do not hold are in *italics*.

### 4.1.1 Atomic propositions

The most simple Sequential Regular Expression is a single atomic proposition. The pattern it specifies is a single state in which the atomic proposition holds. Braces are used for grouping SEREs, so an example of a SERE with just the atomic proposition `a` would be:

$$\{\mathtt{a}\}$$

### 4.1.2 Concatenation

Concatenation of SEREs is another to construct simple patterns. The `;`-operator is used to concatenate two expressions, and the `:`-operator is used to concatenate two expressions such that they have an overlap of a single state. For example, the path

$$\{\mathbf{a}, \mathbf{b}, \neg c\}, \{\mathbf{b}, \mathbf{c}, \neg a\}, \{\mathbf{c}, \neg a, \neg b\}, \ldots$$

would match the pattern specified by the expression

$$\{\mathtt{a}; \mathtt{b}; \mathtt{c}\}$$

as well as the pattern specified by the expression

$$\{\{\mathtt{a}; \mathtt{b}\} : \{\mathtt{c}; \mathtt{c}\}\}.$$

### 4.1.3 Union

The union of two SEREs is another operation that is used with practically all types of regular expressions. The union operator is written as `|`, and it combines two SEREs into a new one that matches all the patterns that either of them would match. For example, the SERE

$$\{\{\mathtt{a}; \mathtt{b}\} \mid \{\mathtt{c}\}\}$$

specifies a pattern that would match with the following two paths:

$$\{\mathbf{a}\}, \{\mathbf{b}\}, \ldots$$
$$\{\mathbf{c}\}, \ldots$$

### 4.1.4 Intersection

The intersection of two SEREs is something that is missing from most forms of regular expressions, because it is a computationally costly operation. The intersection operator is written as `&&`, and it combines two SEREs into a SERE that matches every pattern that both of them match. Additionally, the length of the match must be the same with both SEREs. For example the following SERE:

$$\{\{\texttt{a}\} \mid \{\texttt{b}; \texttt{c}\}\} \;\&\&\; \{\{\texttt{b}\} \mid \{\texttt{d}; \texttt{a}\}\}$$

would match the following paths:

$$\{\mathbf{b}, \mathbf{d}, \neg a, \neg c\}, \{\mathbf{a}, \mathbf{c}, \neg b, \neg d\}, \ldots$$
$$\{\mathbf{a}, \mathbf{b}, \neg c, \neg d\}, \ldots$$

but not these:

$$\{\mathbf{b}, \neg a, \neg c, \neg d\}, \{\mathbf{c}, \neg a, \neg b, \neg d\}, \ldots$$
$$\{\mathbf{a}, \mathbf{d}, \neg b, \neg c\}, \{\mathbf{a}, \neg b, \neg c, \neg d\}, \ldots$$

because the length of the match must be the same on both sides of the SERE.

### 4.1.5 Repetition

Repetition of a SERE is again something that is present in practically every form of regular expressions. There are a few ways to make repeating versions of SEREs: adding `[*]` to the SERE matches patterns that match the original SERE zero or more times, adding `[+]` does the same one or more times, adding `[*`$n$`]` matches the pattern exactly $n$ times, adding `[*`$n : m$`]` matches from $n$ to $m$ times. For example the SERE

$$\{\{\texttt{a}\}[*]; \texttt{b}\}$$

matches the following paths:

$$\{\mathbf{b}\}, \ldots$$
$$\{\mathbf{a}\}, \{\mathbf{b}\}, \ldots$$
$$\{\mathbf{a}\}, \{\mathbf{a}\}, \{\mathbf{a}\}, \{\mathbf{b}\}, \ldots$$

## 4.2 TEMPORAL LOGIC OPERATORS

PSL has the usual set of operators from LTL, which function exactly as in LTL. Only operators that refer to future states are presented here. In addition to the operators here, PSL has lots of syntactic sugar (hence the original name Sugar), including operators that refer to the past. As with SEREs, a formula written with these operators holds in a path if and only if it holds in the first state. Both SEREs and atomic propositions can be used as sub-formulae.

### 4.2.1 Weak and strong operators

PSL divides operators into *weak* and *strong* ones. The exact meaning varies according to the individual operators, but the intuitive difference is how the operator is interpreted when finite paths are used with PSL, and the path ends before enough information is available to resolve the truth value of a formula. A strong version of an operator is interpreted as false if this happens, while a weak operator evaluates to true.

### 4.2.2 Next

The **X**!-operator states that the formula it is applied to holds in the next state. The exclamation mark indicates that this is a strong operator, meaning it is false if there is no next state. The corresponding weak operator is **X**. For example in the path

$$\{\neg a\}\{\mathbf{a}\}, \{\mathbf{a}\}, \{\neg a\}, \ldots$$

the following formulas would hold:

**X**! `a`

**X**!**X**! `a`

but the following would not:

**X**!**X**!**X**! `a`

PSL also defines more verbose versions of the next-operators: `next` and `next!`

### 4.2.3 Globally

The **G**-operator states that the formula it is applied to holds in every state of the path. By itself it can be used to specify invariant properties: if something must hold in every state of every path of a model, then it must hold in every reachable state of the model. The **G**-operator is not limited to specifying invariant properties, however, as it can be embedded into other properties to specify other interesting properties. For example the PSL property

**G**(`a -> G b`)

uses a logical implication to state that whenever the atomic proposition `a` is true, the atomic proposition `b` must hold then and always after that. A more verbose version of the **G**-operator is `always`. It does not have a strong and a weak version, since there is no ambiguity about the value it should have when a path ends. **G** can be thought of as a weak operator, because if the property always held up to the end of the path, then the operator is true.

### 4.2.4 Finally

The **F**-operator specifies that the formula it is applied to must hold in some future state of the path. A typical example that uses the **F**-operator is a property that states that whenever a request is sent, at some subsequent state a

response is received:

$$\mathbf{G}(\texttt{request -> } \mathbf{F}\ \texttt{response})$$

Here `request` and `response` are atomic propositions. Also note that this property does not say that *every* request must get a response: it just states that *after* every request a response must be made. The property would still be true in the following path, for example:

$$\{\texttt{request}\}, \{\texttt{request}\}, \{\texttt{response}\}$$

A verbose version of the **F**-operator is the `eventually!`-operator. Again, no separate strong or weak version exists: if a path ends and the property has not been true, the operator is false. Therefore, the **F** operator can be thought of as a strong operator.

### 4.2.5 Until

The **U**-operator combines two formulae and states that the first must hold from that point on up to, but not including the state where the second formula holds. The operator also requires that the second property must occur at some point. To specify, for example, that `heat` cannot be true before `water` is, and additionally `water` must be true at some point, the following property can be used:

$$\texttt{!heat}\ \mathbf{U}\ \texttt{water}$$

To further specify that `heat` must become true at some point as well, a finally-operator can be added:

$$(\texttt{!heat}\ \mathbf{U}\ \texttt{water})\ \&\&\ \mathbf{F}\ \texttt{heat}$$

The verbose version of **U** is the `until!`-operator. The weak version `until` does not require the second property to become true, but if it does not, then the first property must hold in every subsequent state. **W** can also be used to denote the weak until operator. The `until_` and `until_!` operators are variations that require that the first property must hold also in the state where the second one does.

### 4.2.6 Tail implication

Even though SEREs can be combined in various ways with other Foundation Language formulae, only the start of a SERE pattern match can be attached to other properties unless another operator is introduced. The `|=>`-operator can be used to link the end of a SERE match to other properties. The left operand of tail implication is a SERE and the right operand is any PSL formula. The `|=>`-operator is true in a state if and only if *every* match of the SERE that starts from the state is followed by a state where the other formula is true. For example, the property

$$\{\texttt{a}[+]\}\ \texttt{|=>}\ \texttt{b}$$

would hold in the path

$$\{\mathbf{a}\}, \{\mathbf{a}, \mathbf{b}\}, \{\mathbf{b}\}$$

but not for example in the path

$$\{\mathbf{a}, \neg b\}, \{\mathbf{a}, \neg b\}, \{\mathbf{b}, \neg a\}$$

because the SERE `{a[+]}` matches a single `a`, but `b` does not hold in the second state. `|->` is a variation of the operator that states that the SERE matches and the following property must overlap by a single state.

# 5 PSL SYNTAX AND SEMANTICS

This chapter formally defines the syntax and semantics of PSL. The presentation closely follows that of [1]. The suggested PSL extension with local variables as well as the Optional Branching Extension are not considered. Also note that the some of the operators in Chapter 4 are not considered. See [1] for a list of all PSL operators and information on how they can be expressed with the operators defined here.

## 5.1 SYNTAX

Let $AP$ be the set of atomic propositions, and $p \in AP$. Let $r_1$ and $r_2$ be Sequential Regular Expressions. The syntax of Sequential Regular Expressions (SERE), is defined inductively as follows:

$$\begin{array}{rccccccc} r ::= & [*0] & | & p & | & \neg p & | & \{r_1\} \quad | \quad r_1[+] \quad | \\ & r_1 \cdot r_2 & | & r_1 \circ r_2 & | & r_1 \cup r_2 & | & r_1 \cap r_2 \end{array}$$

Additionally, let $\phi_1$ and $\phi_2$ be PSL formulae and $k$ be a non-negative integer. The syntax of PSL formulas is defined inductively as follows:

$$\begin{array}{rccccccc} \phi ::= & p & | & r_1 & | & r_1! & | & \neg\phi_1 \quad | \quad (\phi_1) \quad | \\ & \mathbf{X}![k]\phi_1 & | & \phi_1 \wedge \phi_2 & | & \phi_1 \, \mathbf{U} \, \phi_2 & | & r_1 \mapsto \phi_1 \end{array}$$

The textual representation, i.e. `|->`, of the $\mapsto$ operator is used in Chapter 4. Some operators from that chapter are omitted here for brevity, since they can be rewritten with the operators of this chapter.

Additionally, the following operators are used in this work when converting formulae to the positive normal form. They appear also in [9].

$$\phi ::= \quad \phi_1 \, \mathbf{R} \, \phi_2 \quad | \quad \phi_1 \vee \phi_2 \quad | \quad r_1 \diamond\!\!\rightarrow \phi_1$$

The $\mathbf{R}$, i.e. releases-operator, is also used in LTL. The $\diamond\!\!\rightarrow$, i.e. tail conjunction-operator, is the dual of the $\mapsto$, it states that there exists some match of the SERE operand in the path, and that the formula holds in the state where the match ends.

## 5.2 SEMANTICS

The semantics of PSL formulae is defined with respect to *models* and their *computation paths*. A *model* over a set of atomic propositions $AP$ is a tuple $(S, S_0, R, L)$[1], where $S$ is a finite set of states, $S_0 \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is the transition relation, and $L : S \to 2^{AP}$ is a function that maps the states to the atomic propositions that hold in them.

[1] See the Kripke structure in Chapter 2.

A *path* $\pi = s_0, s_1, s_2, \ldots$ is a finite or infinite sequence of states. A *computation path* $\pi = s_0, s_1, s_2, \ldots$ of a model is a non-empty finite or infinite sequence of states where $s_0 \in S_0$ and each $(s_i, s_{i+1}) \in R$. The empty path is denoted by $\varepsilon$.

Note that unlike [1], here we use $s_0, s_1$, etc. to denote states in a path, and $\pi_0, \pi_1$, etc. to denote paths.

### 5.2.1 SERE semantics

For each SERE $r$, three languages are defined: $\mathcal{L}(r)$, $\mathcal{F}(r)$, and $\mathcal{I}(r)$. The first is the intuitive, commonly used language of finite words defined by other flavours of regular expressions, the second language contains exactly the words that are the proper prefixes of the words in the first, and the third consists of infinite words that are generated by making some repeating operator of the SERE repeat infinitely, or intuitively getting stuck in some repetition.

We define $S^*$ to mean the set of finite (possibly empty) sequences $s_0, s_1, \ldots, s_n$ where each $s_i \in S$, $S^+$ to mean the set of finite non-empty sequences $s_0, s_1, \ldots, s_n$ where each $s_i \in S$, and $S^\omega$ to mean the set of infinite sequences $s_0, s_1, \ldots$ where each $s_i \in S$. $|\pi|$ is the length of $\pi$ if $\pi \in S^*$, or $\omega$ otherwise. In the following definitions $\pi, \pi_1, \pi_2, \ldots \in S^* \cup S^\omega$ and $s \in S$.

**Base language**
The base language of a SERE is defined inductively as follows:

- $\mathcal{L}([*0]) = \{\varepsilon\}$.
- $\mathcal{L}(p) = \{s \mid p \in L(s)\}$, where $p \in AP$.
- $\mathcal{L}(\{r\}) = \mathcal{L}(r)$.
- $\mathcal{L}(r+) = \{\pi \mid \exists n \in \mathbb{Z}_+ : \pi = \pi_1\pi_2 \ldots \pi_n$ and $\forall i, 1 \leq i \leq n : \pi_i \in \mathcal{L}(r)\}$, where $\mathbb{Z}_+$ is the set of positive integers.
- $\mathcal{L}(r_1 \cdot r_2) = \{\pi_1\pi_2 \mid \pi_1 \in \mathcal{L}(r_1)$ and $\pi_2 \in \mathcal{L}(r_2)\}$.
- $\mathcal{L}(r_1 \circ r_2) = \{\pi_1 s \pi_2 \mid \pi_1 s \in \mathcal{L}(r_1)$ and $s\pi_2 \in \mathcal{L}(r_2)\}$.
- $\mathcal{L}(r_1 \cup r_2) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$.
- $\mathcal{L}(r_1 \cap r_2) = \mathcal{L}(r_1) \cap \mathcal{L}(r_2)$.

For example the language $\mathcal{L}(a+ \cdot \{b \cup c\})$ consists of the words $ab, ac, aab, aac, \ldots$

**Prefix language**
The prefix language is defined as:

$$\mathcal{F}(r) = \{\pi \mid \exists \pi' \in S^+ : \pi\pi' \in \mathcal{L}(r)\}.$$

For example, the language $\mathcal{F}(a+ \cdot \{b \cup c\})$ consists of the words $a, aa, aaa, \ldots$. Note that $b$ or $c$ never show up in the words of the language.

**Loop language**
The loop language is defined inductively as:

- $\mathcal{I}([*0]) = \emptyset$.
- $\mathcal{I}(p) = \emptyset$.
- $\mathcal{I}(\{r\}) = \mathcal{I}(r)$.
- $\mathcal{I}(r+) = \{\pi_1\pi_2 \mid \pi_1 \in \mathcal{L}(r+) \cup \{\varepsilon\} \text{ and } \pi_2 \in \mathcal{I}(r)\} \cup$
  $\{\pi \mid \pi \in S^\omega, \pi = \pi_1\pi_2\pi_3 \ldots, \forall i \in \mathbb{Z}_+ : \pi_i \in \mathcal{L}(r)\}$.
- $\mathcal{I}(r_1 \cdot r_2) = \mathcal{I}(r_1) \cup \{\pi_1\pi_2 \mid \pi_1 \in \mathcal{L}(r_1) \text{ and } \pi_2 \in \mathcal{I}(r_2)\}$.
- $\mathcal{I}(r_1 \circ r_2) = \mathcal{I}(r_1) \cup \{\pi_1 s \pi_2 \mid \pi_1 s \in \mathcal{L}(r_1) \text{ and } s\pi_2 \in \mathcal{I}(r_2)\}$.
- $\mathcal{I}(r_1 \cup r_2) = \mathcal{I}(r_1) \cup \mathcal{I}(r_2)$.
- $\mathcal{I}(r_1 \cap r_2) = \mathcal{I}(r_1) \cap \mathcal{I}(r_2)$.

For example, the language $\mathcal{I}(a+ \cdot \{b \cup c\})$ consists of only the single infinite word $a^\omega = aaa \ldots$. On the other hand, the language $\mathcal{I}(\{a+ \cdot \{b \cup c\}\}+)$ has the word $a^\omega$ but also the infinite words $(ab)^\omega, (ac)a^\omega, (abaac)^\omega$ among others, since the infinite looping can be done at either of the repetition operators.

### 5.2.2 PSL semantics

The PSL semantics itself is divided into three classes: the *strong*, *neutral*, and *weak* semantics. As with strong and weak operators in Chapter 4, the semantics differ when a path ends and the truth values of some operators are inconclusive in some way. With some operators the strong semantics demands that some additional conditions are met within the path, when the neutral and weak semantics do not. The weak semantics, on the other hand, relax some requirements, stating that some conditions need not be satisfied in a finite path. With infinite paths, the three semantics are identical. Intuitively, the strong semantics can be thought to require that all the conditions of the operator are met within the finite path: if it can be extended in such a way that the operator would become false, then the operator is false in the strong semantics. The weak semantics, on the other hand, works the other way around: if the path could be extended in such a way that the operator becomes true, then the operator is true in the weak semantics.

Here we again use a notation that is slightly different from [1]. We use $\pi \models_i^v \phi$ to denote that the formula $\phi$ holds with the semantics $v$ on the path $\pi$ at point $i$. Here $v$ is one of strong, neutral, or weak, and $i \in \mathbb{N}$.

In each case we assume that $\pi = s_0, s_1, s_2, \ldots \in S^* \cup S^\omega$. The semantics of PSL formulae is defined inductively as follows:

- $\pi \models_i^v p \Leftrightarrow p \in L(s_i)$, where $p \in AP$.
- $\pi \models_i^v r! \Leftrightarrow \begin{cases} \exists j \geq i : s_i, \ldots, s_j \in \mathcal{L}(r), \text{ or} \\ \varepsilon \in \mathcal{L}(r), \text{ or} \\ v = \text{weak and } \pi \in \mathcal{F}(r) \cup \{\varepsilon\}. \end{cases}$

- $\pi \models_i^v r \Leftrightarrow \begin{cases} v \neq \text{strong and } \pi \in \mathcal{F}(r) \cup \{\varepsilon\}, \text{ or} \\ \pi \in \mathcal{I}(r), \text{ or} \\ \pi \models_i^v r!. \end{cases}$
- $\pi \models_i^v r \mapsto \phi \Leftrightarrow \begin{cases} \forall j \geq i : \text{ if } s_i, \ldots, s_j \in \mathcal{L}(r), \text{ then } \pi \models_j^v \phi, \text{ and} \\ \text{if } \varepsilon \in \mathcal{L}(r), \text{ then } \pi \models_i^v \phi, \text{ and} \\ \text{if } v = \text{strong}, \text{ then } s_i, \ldots \notin \mathcal{F}(r) \cup \{\varepsilon\}. \end{cases}$
- $\pi \models_i^v r \diamond\!\!\rightarrow \phi \Leftrightarrow \begin{cases} \exists j \geq i : s_i, \ldots, s_j \in \mathcal{L}(r) \text{ and } \pi \models_j^v \phi, \text{ or} \\ \varepsilon \in \mathcal{L}(r) \text{ and } \pi \models_i^v \phi, \text{ or} \\ v = \text{weak and } s_i, \ldots \in \mathcal{F}(r) \cup \{\varepsilon\}. \end{cases}$
- $\pi \models_i^v (\phi) \Leftrightarrow \pi \models_i^v \phi.$
- $\pi \models_i^v \neg\phi \Leftrightarrow \begin{cases} v = \text{strong and } \pi \not\models_i^{\text{weak}} \phi, \text{ or} \\ v = \text{neutral and } \pi \not\models_i^{\text{neutral}} \phi, \text{ or} \\ v = \text{weak and } \pi \not\models_i^{\text{strong}} \phi. \end{cases}$
- $\pi \models_i^v \phi \wedge \psi \Leftrightarrow (\pi \models_i^v \phi \text{ and } \pi \models_i^v \psi).$
- $\pi \models_i^v \phi \vee \psi \Leftrightarrow (\pi \models_i^v \phi \text{ or } \pi \models_i^v \psi).$
- $\pi \models_i^v \mathbf{X}![k]\phi \Leftrightarrow \begin{cases} |\pi| \geq i + k \text{ and } \pi \models_{i+k}^v \phi, \text{ or} \\ v = \text{weak and } |\pi| < i + k. \end{cases}$
- $\pi \models_i^v \phi_1 \mathbf{U} \phi_2 \Leftrightarrow \begin{cases} \exists j \geq i : \pi \models_j^v \phi_2 \text{ and } \forall k, i \leq k < j : \pi \models_k^v \phi_1, \text{ or} \\ v = \text{weak and } \pi \in S^* \text{ and } \forall k, i \leq k \leq |\pi| : \pi \models_k^v \phi_1. \end{cases}$
- $\pi \models_i^v \phi_1 \mathbf{R} \phi_2 \Leftrightarrow \begin{cases} \forall j \geq i : \pi \models_j^v \phi_2 \text{ or } \exists k, i \leq k < j : \pi \models_k^v \phi_1, \text{ and} \\ v \neq \text{strong or } \pi \in S^\omega \text{ or } \exists k, i \leq k \leq |\pi| : \pi \models_k^v \phi_1. \end{cases}$

A formula $\phi$ holds with semantics $v$ for a computation path $\pi$, denoted $\pi \models^v \phi$, if and only if $\pi \models_0^v \phi$. A formula holds with a certain semantics in a model if and only if it holds in every computation path of the model with that semantics. A formula *fails* on a model if and only if it does not hold with the weak semantics.

# 6 EVALUATING FORMULAE ON FINITE EXECUTIONS

When building an observer for PSL properties, and in fact for any properties, only finite computation paths can be considered. In the case of properties where an infinite execution is needed to determine if an aspect of the property holds or not, some approximation needs to be done. Usually with model checking an under-approximation is appropriate: a property is reported to fail only if the approximation algorithm can be certain it fails. This is also the approach taken here.

With the PSL semantics described in the previous chapter, a convenient and intuitive approximation approach is to report when a property fails on the observed path, i.e. $\pi \not\models^{\text{weak}} \phi$. In that case the path can no longer satisfy the property, even if all possible extensions of the path are considered: a finite counterexample of the property has been found. This is equivalent to $\pi \models^{\text{strong}} \neg\phi$. An observer for a formula in the strong semantics can be constructed, because a finite path is always sufficient to determine if the formula holds. Negations in the formulae refer to other semantics, however:

$$\pi \models_i^v \neg\phi \Leftrightarrow \begin{cases} v = \text{strong and } \pi \not\models_i^{\text{weak}} \phi, \text{ or} \\ v = \text{neutral and } \pi \not\models_i^{\text{neutral}} \phi, \text{ or} \\ v = \text{weak and } \pi \not\models_i^{\text{strong}} \phi. \end{cases}$$

To avoid the need to create observers with other than strong semantics, the formula can be converted into positive normal form, which means allowing negations only in front of atomic propositions, where the three semantics are equivalent. The following rewriting rules, that are obtained by logically negating the semantics of the formulae, can be used to push negations next to atomic propositions:

- $\pi \models_i^v \neg\neg\phi \Leftrightarrow \pi \models_i^v \phi$,
- $\pi \models_i^v \neg(r \mapsto \phi) \Leftrightarrow \pi \models_i^v r \diamond\!\!\rightarrow \neg\phi$,
- $\pi \models_i^v \neg(\phi_1 \wedge \phi_2) \Leftrightarrow \pi \models_i^v (\neg\phi_1) \vee (\neg\phi_2)$, and
- $\pi \models_i^v \neg(\phi_1 \mathbf{U} \phi_2) \Leftrightarrow \pi \models_i^v (\neg\phi_1) \mathbf{R} (\neg\phi_2)$.

Additionally, because strong semantics is in use, the following rewriting rule can also be used:

- $\pi \models_i^{\text{strong}} \neg\mathbf{X}![k]\phi \Leftrightarrow \pi \models_i^{\text{strong}} \mathbf{X}![k]\neg\phi$.

Since the semantics of the $\diamond\!\!\rightarrow$ operator are not part of the PSL standard, the rewriting rule for it is proven here:

**Proposition 6.1.** $\pi \models_i^v \neg(r \mapsto \phi) \Leftrightarrow \pi \models_i^v r \diamond\!\!\rightarrow \neg\phi$.

*Proof.* For $v \in \{\text{strong}, \text{neutral}, \text{weak}\}$, we define

$$\overline{v} = \begin{cases} \text{weak, when } v = \text{strong} \\ \text{neutral, when } v = \text{neutral} \\ \text{strong, when } v = \text{weak} \end{cases}$$

Recall that the definition of $\pi \models_i^v r \mapsto \phi$ is:

$$\begin{cases} \forall j > i: \text{ if } s_i, \ldots, s_j \in \mathcal{L}(r), \text{ then } \pi \models_j^v \phi, \text{ and} \\ \text{if } \varepsilon \in \mathcal{L}(r), \text{ then } \pi \models_i^v \phi, \text{ and} \\ \text{if } v = \text{strong}, \text{ then } s_i, \ldots \notin \mathcal{F}(r) \cup \{\varepsilon\}. \end{cases}$$

The proposition can then be proven with:

$$\begin{aligned} & \pi \models_i^v \neg(r \mapsto \phi) \\ \Leftrightarrow \quad & \pi \not\models_i^{\overline{v}} r \mapsto \phi \\ \Leftrightarrow \quad & \begin{cases} \exists j > i : s_i, \ldots, s_j \in \mathcal{L}(r) \text{ and } \pi \not\models_j^{\overline{v}} \phi, \text{ or} \\ \varepsilon \in \mathcal{L}(r) \text{ and } \pi \not\models_i^{\overline{v}} \phi, \text{ or} \\ \overline{v} = \text{strong and } s_i, \ldots \in \mathcal{F}(r) \cup \{\varepsilon\} \end{cases} \\ \Leftrightarrow \quad & \begin{cases} \exists j > i : s_i, \ldots, s_j \in \mathcal{L}(r) \text{ and } \pi \models_j^v \neg\phi, \text{ or} \\ \varepsilon \in \mathcal{L}(r) \text{ and } \pi \models_i^v \neg\phi, \text{ or} \\ v = \text{weak and } s_i, \ldots \in \mathcal{F}(r) \cup \{\varepsilon\} \end{cases} \\ \Leftrightarrow \quad & \pi \models_i^v r \diamond\!\!\rightarrow \neg\phi \end{aligned}$$

□

# 7 TRANSDUCERS FOR PSL FORMULAE

*Transducers* (also called testers in e.g. [26]) are a symbolic variant of finite state automata. Intuitively a transducer is an automaton that has a finite number of input and state variables and one output variable, all with boolean domains, and a transition relation that maps values of the variables in a state to new values in the next state. Instead of accepting or rejecting the input at the end of the execution like regular finite automata, they can signal acceptance or rejection at each point. This can be interpreted as acceptance starting from that point, and since the transducer can refer to future input values, they are inherently non-deterministic.

In this work we use transducers to signal whether a PSL property holds starting at each point of execution. Translating transducers to NuSMV modules is very straightforward, as is building larger transducers from other, smaller component transducers. Additionally, proving properties about them is convenient. These are the main reasons for using this formalism here.

## 7.1 NON-DETERMINISTIC FINITE AUTOMATA

Some of the transducers in this chapter require the concept of a non-deterministic finite automaton (NFA). A non-deterministic finite automaton is a state machine that is given a finite sequence of input letters, and at the end of the sequence the automaton either accepts or rejects the input sequence. Formally, a non-deterministic finite automaton is a quintuple $(Q, \Sigma, \delta, q_0, F)$, where:

- $Q$ is a finite set of states,
- $\Sigma$ is a finite set of input letters, i.e. the alphabet,
- $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation,
- $q_0 \in Q$ is the initial state, and
- $F$ is the set of accepting states.

An input sequence $w$, called a *word*, is a finite sequence of input letters: $w = \sigma_1, \sigma_2, \ldots, \sigma_n$, where each $\sigma_i \in \Sigma$. An execution of an automaton $A$ for $w$ is a sequence of transitions $(q_0, \sigma_1, q'_1), (q'_1, \sigma_2, q'_2), \ldots, (q'_{n-1}, \sigma_n, q'_n)$, where $q_0$ is the initial state, and each triplet is a member of the transition relation, and therefore each $q'_i$ is a member of $Q$. The automaton is defined to accept the word if an execution exists for which $q'_n \in F$. The language of the automaton, denoted by $\mathcal{L}(A)$, is the set of words it accepts.

## 7.2 TRANSDUCERS

Formally, a transducer $T$ is a tuple $(Q, Q^{\text{in}}, q^{\text{out}}, I, F, \delta)$, where:

- $Q$ is a finite set of state variables.

- $Q^{\text{in}}$ is a finite set of input variables, disjoint from $Q$.
- $q^{\text{out}} \in Q$ is the output variable.
- Every subset of $Q \cup Q^{\text{in}}$ is a state of the transducer. A variable $v$ is true in a state $s \subseteq Q \cup Q^{\text{in}}$ if and only if $v \in s$.
- $I \subseteq 2^{Q \cup Q^{\text{in}}}$ is a set of initial transducer states.
- $F \subseteq 2^{Q \cup Q^{\text{in}}}$ is a set of final transducer states.
- $\delta \subseteq 2^{Q \cup Q^{\text{in}}} \times 2^{Q \cup Q^{\text{in}}}$ is the transition relation.

## 7.3 TRANSDUCER EXECUTION

An execution of a transducer $T$ is a finite non-empty sequence of states, $s_1, s_2, \ldots, s_n$, where $\forall i, 1 \leq i < n : (s_i, s_{i+1}) \in \delta$.

The input of a transducer is a path: $\pi = p_1, \ldots, p_n$, where each $p_i$ is a set of input variables from some input domain $D$, where $Q^{\text{in}} \subseteq D$. Formally, $\pi \in (2^D)^+$. An execution $s_1, s_2, \ldots, s_n$ of a transducer $T$ is defined to be an execution for $\pi$ if for every input variable $v \in Q^{\text{in}}$ and for every $i, 1 \leq i \leq n$ the following holds: $v \in s_i \Leftrightarrow v \in p_i$.

A transducer is defined to *accept* its input starting from point $j$ if there exists an execution $s_1, \ldots, s_j, \ldots, s_n$ where $s_1 \in I$, $s_n \in F$, and the output $q^{\text{out}}$ variable is true at state $s_j$.

The example below is a transducer that accepts at a point of a path where $a$ holds exactly in every other state until the end of the path, the last state being one where $a$ does not hold. The beginning of the path can be anything, as long as the end of the path consists of at least one state where $a$ holds and one state where it does not.

$T = (Q, Q^{\text{in}}, q^{\text{out}}, I, F, \delta)$, where:

- $Q = \{q_0, q_1\}$,
- $Q^{\text{in}} = \{a\}$,
- $q^{\text{out}} = q_0$,
- $I = 2^{Q \cup Q^{\text{in}}}$,
- $F = \{s \mid s \subseteq (Q \cup Q^{\text{in}}), q_0 \notin s \wedge (q_1 \in s \Leftrightarrow a \notin s)\}$, i.e. $\{\{q_1\}, \{a\}\}$, and
- $\delta = \{(s, s') \mid s \subseteq (Q \cup Q^{\text{in}}), s' \subseteq (Q \cup Q^{\text{in}}), (q_0 \in s \Rightarrow a \in s \wedge q_1 \in s') \wedge (q_1 \in s \Rightarrow a \notin s \wedge q_0 \in s')\}$.

The state variables are $q_0$ and $q_1$, and $q_0$ is also the output variable. $q_0$ is used to signal a non-deterministic guess that the rest of the path consists of states where $a$ alternates between true and false. The transition relation enforces this: $q_0$ or $q_1$ need not ever be true, but if they are, $a$ must have the appropriate value, and the other variable must be true in the next state. The constraints do not prevent transitions to states where e.g. $q_0$ and $q_1$ are both

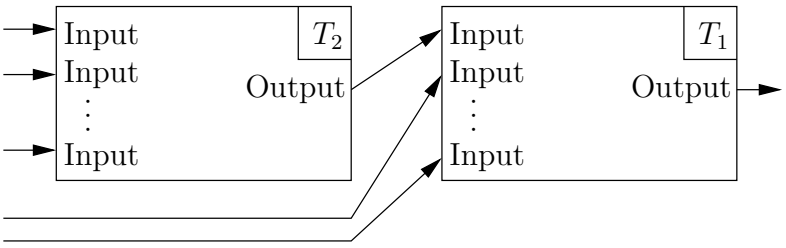



Figure 7.1: A composition of two transducers, $T_1$ and $T_2$. $T_1$ gets the output information from $T_2$ into an input variable.

true, which can be a bit counter-intuitive, but since no transitions from such states are present, they need not be explicitly ruled out from the reachable states of the transducer.

The final state constraint works together with the transition relation to enforce that the non-deterministic guess of setting $q_0$ to true is done properly: it states that $q_0$ can not be true, meaning that the alternation of $q_0$ and $q_1$ must always end with $q_1$, and $q_1$ holding in the last state is equivalent to $a$ not holding. This constraint is necessary, since the transition relation does not constrain $a$ in the succeeding state: $q_0$ implies that $q_1$ must hold next, but that does not mean that $a$ must not hold next. States where both $q_1$ and $a$ hold have no outgoing transitions, but the final state constraint must rule out such states as the final state.

## 7.4 TRANSDUCER COMPOSITION

For transducer composition, we define a renaming operation for sets. Let $S$ be a set, and let $a \in S$. Renaming $a$ to $b$ in the set $S$ is then denoted by $S[a/b]$, and defined as $(S \setminus \{a\}) \cup \{b\}$. For a pair of sets $(S, S')$, $(S, S')[a/b]$ is shorthand for $(S[a/b], S'[a/b])$. The latter notation is used for renaming with transition relations.

Let $T_1 = (Q_1, Q_1^{\text{in}}, q_1^{\text{out}}, I_1, F_1, \delta_1)$ and $T_2 = (Q_2, Q_2^{\text{in}}, q_2^{\text{out}}, I_2, F_2, \delta_2)$ be two transducers. A composition of the transducers $T_1$ and $T_2$, with respect to some $q^{\text{in}} \in Q_1^{\text{in}}$, connects the output of $T_2$ to one input variable of $T_1$. Figure 7.1 illustrates this. The resulting transducer has all the input variables of $T_1$ and $T_2$, except for the one input of $T_1$ that was used up by the composition. The output variable of the new transducer is the output variable of $T_1$. Transducer composition should not be confused with concatenating the accepted paths of the transducers. The properties of the resulting transducer depend solely on the components: $T_1$ is free to use the information it gets from $T_2$ in any way.

Transducer composition is denoted as $T_2 \rhd_{q^{\text{in}}} T_1$, and defined as $T_\rhd = (Q_\rhd, Q_\rhd^{\text{in}}, q_\rhd^{\text{out}}, I_\rhd, F_\rhd, \delta_\rhd)$, where:

- $Q_\rhd = Q_1 \cup Q_2$,
- $Q_\rhd^{\text{in}} = (Q_1^{\text{in}} \setminus \{q^{\text{in}}\}) \cup Q_2^{\text{in}}$,
- $q_\rhd^{\text{out}} = q_1^{\text{out}}$,

- $I_{\triangleright} = \left\{ s_1[q^{\text{in}}/q_2^{\text{out}}] \cup s_2 \;\middle|\; \begin{array}{l} s_1 \in I_1, s_2 \in I_2, \text{ and} \\ q^{\text{in}} \in s_1 \Leftrightarrow q_2^{\text{out}} \in s_2, \text{ and} \\ \bigwedge\limits_{v \in Q_1^{\text{in}} \cap Q_2^{\text{in}}} v \in s_1 \Leftrightarrow v \in s_2 \end{array} \right\}$,
- $F_{\triangleright} = \left\{ s_1[q^{\text{in}}/q_2^{\text{out}}] \cup s_2 \;\middle|\; \begin{array}{l} s_1 \in F_1, s_2 \in F_2, \text{ and} \\ q^{\text{in}} \in s_1 \Leftrightarrow q_2^{\text{out}} \in s_2, \text{ and} \\ \bigwedge\limits_{v \in Q_1^{\text{in}} \cap Q_2^{\text{in}}} v \in s_1 \Leftrightarrow v \in s_2 \end{array} \right\}$, and
- $\delta_{\triangleright} = \left\{ (s_1 \cup s_2, s_1' \cup s_2')[q^{\text{in}}/q_2^{\text{out}}] \;\middle|\; \begin{array}{l} (s_1, s_1') \in \delta_1, (s_2, s_2') \in \delta_2, \text{ and} \\ (q^{\text{in}} \in s_1 \Leftrightarrow q_2^{\text{out}} \in s_2) \wedge (q^{\text{in}} \in s_1' \Leftrightarrow q_2^{\text{out}} \in s_2'), \text{ and} \\ \bigwedge\limits_{v \in Q_1^{\text{in}} \cap Q_2^{\text{in}}} (v \in s_1 \Leftrightarrow v \in s_2) \wedge (v \in s_1' \Leftrightarrow v \in s_2') \end{array} \right\}$.

The constraints set by the initial states, the final states, and the transition relations are preserved in the composition. This is done by forming the new initial states from the union of the variables from each initial state of $T_1$ with the variables form each initial state of $T_2$, but only when they agree on the plugged variable and the input variables they share. The same is done for the final states and the transitions. $T_1$ and $T_2$ may not share any state variables and the plugged input variable cannot exist in $T_2$, i.e. we require that $Q_1 \cap Q_2 = \emptyset$ and $q^{\text{in}} \notin Q_2^{\text{in}}$.

To give an example of transducer composition, we will use the previous transducer example. An additional condition is added: the suffix of the path where $a$ alternates must contain at least two states where $a$ holds. To achieve this, we will construct a transducer that accepts its input if its input variable becomes true at least twice, and then compose it with the previous example.

The new transducer is defined as $T_2 = (Q_2, Q_2^{\text{in}}, q_2^{\text{out}}, I_2, F_2, \delta_2)$, where:

- $Q_2 = \{q\}$,
- $Q_2^{\text{in}} = \{q_2^{\text{in}}\}$,
- $q_2^{\text{out}} = q$,
- $I_2 = \{s \mid s \subseteq (Q \cup Q^{\text{in}}), q \in s \Leftrightarrow q_2^{\text{in}} \in s\}$, i.e. $\{\emptyset, \{q, q_2^{\text{in}}\}\}$,
- $F_2 = \{s \mid s \subseteq (Q \cup Q^{\text{in}}), q \notin s\}$, i.e. $\{\emptyset, \{q_2^{\text{in}}\}\}$, and
- $\delta_2 = \{(s, s') \mid s \subseteq (Q \cup Q^{\text{in}}), s' \subseteq (Q \cup Q^{\text{in}}), (q \in s' \Leftrightarrow q \in s) \vee q_2^{\text{in}} \in s'\}$.

The only state variable, $q$, is used to verify that the input is true twice. The transition relation permits changing the value of $q$ only when the input variable $q_2^{\text{in}}$ is true. Thus, if the input is true only once, it can be set to true, but not to false again. This, in turn, violates the constraint on the final state. Therefore, if $q$ becomes true in an execution, $q_2^{\text{in}}$ must be true at least twice. Also note that $q_2^{\text{in}}$ being true does not force changing the value of $q$, so $q_2^{\text{in}}$ can be true an odd number of times as well. In this case we also need an initial state constraint to be able to detect if the input variable is true in the first state, and set $q$ accordingly. A quirk of the transducer is that no execution exists if $q_2^{\text{in}}$ is true in the initial state, but not in any subsequent state. This

is not a real problem, however, because of the way accepting is defined with transducers.

The composition of the two examples is $T_\triangleright = (Q_\triangleright, Q^{\text{in}}_\triangleright, q^{\text{out}}_\triangleright, I_\triangleright, F_\triangleright, \delta_\triangleright) = T \triangleright_{q_2^{\text{in}}} T_2$, where:

- $Q_\triangleright = \{q, q_0, q_1\}$,
- $Q^{\text{in}}_\triangleright = \{a\}$,
- $q^{\text{out}}_\triangleright = q$,
- $I_\triangleright = \{s \mid s \subseteq (Q \cup Q^{\text{in}}), q \in s \Leftrightarrow q_0 \in s\}$,
- $F_\triangleright = \{s \mid s \subseteq (Q \cup Q^{\text{in}}), q_0 \notin s \wedge (q_1 \in s \Leftrightarrow a \notin s) \wedge q \notin s\}$,
- $\delta_\triangleright = \{(s, s') \mid s \subseteq (Q \cup Q^{\text{in}}), s' \subseteq (Q \cup Q^{\text{in}}), ((q \in s' \Leftrightarrow q \in s) \vee q_0 \in s') \wedge (q_0 \in s \Rightarrow a \in s \wedge q_1 \in s') \wedge (q_1 \in s \Rightarrow a \notin s \wedge q_0 \in s')\}$.

Also note that this composition relies on the fact that $T$ can set its output to true in every state where $a$ holds and the requirement of alternation is met: with long suffixes where $a$ alternates, the output can be true whenever $a$ is. This means that $T_2$ can expect at least two states where its input is true. $T_2$ is not so well behaved, however: its output can be set to true in the first state where its input becomes true, but it will stay true until its input becomes true in a subsequent state, after which it is false again. This could be fixed by composing $T_\triangleright$ with another transducer that sets its output to true only in the first state where its input is true, for example.

## 7.5 TRANSDUCERS FOR FORMULAE

The following rewriting rules can be used to remove SEREs that are not a part of a tail implication or a tail conjunction, thus removing the need to separately define transducers for them.

- $\pi \models_i^v r! \Leftrightarrow \pi \models_i^v r \diamond\!\!\rightarrow \top$.
- $\pi \models_i^{\text{strong}} r \Leftrightarrow \pi \models_i^{\text{strong}} r!$, and

$\mathbf{X}![k]\phi$ is rewritten to $\overbrace{\mathbf{X}\mathbf{X}\ldots\mathbf{X}}^{k}\phi$, which is equivalent to the former when using strong semantics.

For the transducer construction for the formulae $r \mapsto \phi$ and $r \diamond\!\!\rightarrow \phi$, we define some additional notation. For the SERE $r$, we use $AP(r)$ to denote the set of atomic propositions appearing in $r$. We also define a function that maps the states of a transducer to the atomic propositions of the SERE: $\ell : 2^{Q^{\text{in}} \cup Q} \rightarrow 2^{AP(r)}$. For any state $s$ of the transducer: $\ell(s) = s \cap AP(r)$. We use $\ell(s_0, \ldots, s_n)$ to denote $\ell(s_0), \ldots, \ell(s_n)$.

A transducer $T$ that is constructed for a formula $\phi$ must have the following property: for every path $\pi$ there must be an execution where the transducer accepts at every point $i$ if $\pi \models_i^{\text{strong}} \phi$. It is not sufficient that for each such $i$ there exists some execution where $T$ accepts at $i$, there must exist a single execution where $T$ accepts at every such $i$. This is because of how the

composition is defined, and because transducers for formulae use the output values of the transducers for its sub-formulae. If a transducer requires that a sub-formula is true at certain points, the output of the transducer for the sub-formula must be true at those points within a single execution. Naturally a transducer for a formula $\phi$ can not have executions for any $\pi$ where they accept at $i$ if $\pi \not\models_i^{\text{strong}} \phi$.

### 7.5.1 $r \mapsto \phi$

In the transducer construction we assume that the base language $\mathcal{L}(r)$ of $r$ is not empty. If $\mathcal{L}(r) = \emptyset$, the formula can be rewritten to $\top$, and no transducer is constructed.

The intuition behind the transducer is that multiple copies of an automaton for the SERE are simulated, which yields the matches for the SERE. When a simulated copy accepts, $\phi$ should hold.

Let $A_r = (Q_r, \Sigma, \delta_r, q_0, F_r)$ be a finite, non-deterministic automaton with $F_r$ reachable from every state and no $\varepsilon$-transitions, s.t. $\mathcal{L}(A_r) = \mathcal{L}(r)$ and $\Sigma = 2^{AP(r)}$. We can then construct a transducer $T = (Q, Q^{\text{in}}, q^{\text{out}}, I, F, \delta)$ for the tail implication, where:

- $Q = Q_r$,
- $Q^{\text{in}} = AP(r) \cup \{q^\phi\}$, where $q^\phi$ is the input variable to which the output of $T_\phi$ is connected,
- $q^{\text{out}} = q_0$, the initial state of $A_r$,
- $I = 2^{Q \cup Q^{\text{in}}}$,
- $F = \{s \mid s \subseteq (Q \cup Q^{\text{in}}), s \cap Q_r = \emptyset\}$, and
- $\delta = \left\{ (s, s') \;\middle|\; \begin{array}{l} \bigwedge\limits_{(v,\sigma,v') \in \delta_r} \Big( (v \in s \wedge \ell(s) = \sigma) \Rightarrow v' \in s' \Big) \text{ and} \\ F_r \cap s' \neq \emptyset \Rightarrow q^\phi \in s \end{array} \right\}$, where $s \subseteq (Q \cup Q^{\text{in}})$ and $s' \subseteq (Q \cup Q^{\text{in}})$.

The transducer for the entire sub-formula is obtained with the composition $T_\phi \rhd_{q^\phi} T$, where $T_\phi$ is a transducer for $\phi$. Intuitively, the first part of the transition relation handles simulating copies of the automaton for the SERE, and the second part states that when the simulated automaton accepts, $\phi$ must hold. The final state constraint states that no copies of the simulated automata can be left running, which takes care of the requirement that the suffix of the path cannot belong to the prefix language of $r$.

This transducer only satisfies the following semantics:

$$\pi \models_i^v r \mapsto \phi \Leftrightarrow \left\{ \begin{array}{l} \forall j > i : \text{ if } s_i, \ldots, s_j \in \mathcal{L}(r) \text{ then } \pi \models_j^v \phi, \text{ and} \\ \text{if } v = \text{strong then } s_i, \ldots \notin \mathcal{F}(r) \cup \{\varepsilon\}. \end{array} \right.$$

The missing part of the semantics from the semantics chapter, if $\varepsilon \in \mathcal{L}(r)$ then $\pi \models_i^v \phi$, can be added by rewriting the formula to $(r \mapsto \phi) \wedge \phi$ if $\varepsilon \in \mathcal{L}(r)$, which is trivial to check from $A_r$.

This transducer must be run one step further than the actual path that is under inspection, because of how the state variables of $A_r$ are handled. Intuitively, in the second state of the execution, $A_r$ can be interpreted to have been given one input. Therefore, to determine its state after the last input in the path, one more step of execution must be performed.

**Proposition 7.1.** *If $s_0, \ldots, s_i, \ldots, s_n$ is an execution of the transducer that accepts at $s_i$, then $\ell(s_i, \ldots, s_{n-1}) \notin \mathcal{F}(r)$ and $\forall j, i < j < n$ : if $s_i, \ldots, s_j \in \mathcal{L}(r)$ then $q^\phi \in s_j$.*

*Proof.* Follows from Lemmas 7.2, 7.3, and 7.4. Lemmas 7.2 and 7.3 together prove that all matches for $r$ are detected whenever $q_0$ is set to true, and Lemma 7.4 proves that the suffix of the path after $q_0$ is true cannot belong to the prefix language of $r$. □

**Lemma 7.2.** *If $A_r$ has a sequence of transitions for an input $\sigma_i, \sigma_{i+1}, \ldots, \sigma_{j-1}$ that takes $A_r$ from some state $v_i$ to some state $v_j$, and $v_i \in s_i$, then for every execution $s_i, \ldots, s_j$, where $\forall l, i \leq l < j : \ell(s_l) = \sigma_l$, $v_j \in s_j$.*

*Proof.* Proof by induction over $j$:

- *Base case*: $i = j$, so $v_j \in s_j$ is equivalent to $v_i \in s_i$.
- *Induction assumption*: The above statement holds when $j \leq k$.
- *Induction step*: For every $(v, \sigma, v') \in \delta_r$ and $(s, s') \in \delta$, $(v \in s \wedge \ell(s) = \sigma) \Rightarrow v' \in s'$. In other words, if $v$ is true in some transducer state $s$ and $\ell(s) = \sigma$, then $v'$ is true in every follower state $s'$ of $s$. This means that if $A_r$ has a transition that takes it from $v$ to $v'$ with the input $\ell(s)$, and $v$ holds in $s$, then $v'$ must hold in every follower state of $s$. The induction assumption states that $v_k \in s_k$. If $A_r$ has a transition $(v_k, \sigma_k, v_{k+1})$, then $v_{k+1} \in s_{k+1}$. Therefore the above statement holds for $j \leq k + 1$ as well, and by induction for any $j$.

□

**Lemma 7.3.** *If $s_0, \ldots, s_n$ is an execution of $T$, $q_0 \in s_0$, and $\ell(s_0, \ldots, s_{n-1}) \in \mathcal{L}(r)$, then $F_r \cap s_n \neq \emptyset$.*

*Proof.* Because $\ell(s_0, \ldots, s_{n-1}) \in \mathcal{L}(r)$, there exists a sequence of transitions $(q_0, \ell(s_0), q_1)$, $(q_1, \ell(s_1), q_2)$, …, $(q_{n-1}, \ell(s_{n-1}), q_n)$, each in $\delta_r$, for some $q_1, q_2, \ldots \in Q_r$ and $q_n \in F_r$. This, combined with Lemma 7.2 implies $q_n \in s_n$. □

**Lemma 7.4.** *If $s_0, \ldots, s_n$ is an execution of $T$ that accepts at $s_i$, then $\ell(s_i, \ldots, s_{n-1}) \notin \mathcal{F}(r)$.*

*Proof.* Suppose that $\ell(s_i, \ldots, s_{n-1}) \in \mathcal{F}(r)$, i.e. $\ell(s_i, \ldots, s_{n-1})$ is a proper prefix of some word in $\mathcal{L}(r)$. Since $q_0 \in s_i$, and because of Lemma 7.2, $q_j \in s_n$ for some $q_j \in Q_r$. This contradicts with the terminating condition of $T$, so $s_i, \ldots, s_{n-1} \notin \mathcal{F}(r)$. □

**Proposition 7.5.** *Let $\pi = p_0, p_1, \ldots, p_{n-1}$ be a path. There exists an accepting execution $s_0, s_1, \ldots, s_n$ of $T$ for $\pi$ such that for every pair of indices $i, j$ the following holds: if $p_i, \ldots, p_j \in \mathcal{L}(r)$ and $\pi \models_j^{strong} \phi$, then $q_0 \in s_i$.*

*Proof.* Follows from Lemmas 7.6 and 7.7. Lemma 7.6 proves that an accepting execution exists for a path where a single accepting state is needed, and Lemma 7.7 proves that the union of two accepting executions is an accepting execution. □

**Lemma 7.6.** *If $\pi = p_0, p_1, \ldots, p_{n-1}$ is a path and $\pi \models_k^{strong} r \mapsto \phi$ for some $k$, then there exists an accepting execution $s_0, s_1, \ldots, s_n$ of $T$ for $\pi$ such that $q_0 \in s_k$.*

*Proof.* The execution $s_0, s_1, \ldots, s_n$ can be constructed in the following way:

- $\ell(s_i) = s_i$ when $0 \leq i < k$, i.e. only atomic propositions hold in the states before $s_k$, all the state variables are false.
- $q_0 \in s_k$.
- $s_k \setminus \ell(s_k) = q_0$, i.e. $q_0$ is the only state variable that holds in $s_k$.
- $\bigwedge_{(v,\sigma,v') \in \delta_r} \Big( (v \in s_i \wedge \ell(s_i) = \sigma) \Leftrightarrow v' \in s_{i+1} \Big)$ when $k \leq i < n$, i.e. the implication in the transition relation is replaced with equivalence after $s_k$.

For the above execution, if $q \in s_i$ for some $q \in Q_r$, then there exists an execution of $A_r$ that takes it from $q_0$ to $q$ with the input sequence $\ell(s_k, \ldots, s_{i-1})$. This is proven by induction over $i$:

- *Base case*: $i = k$, and the execution of $A_r$ is the empty sequence.
- *Induction assumption*: The above statement holds when $i \leq j$.
- *Induction step*: Let $m$ be any $k < m \leq n$. A state variable $v \in Q_r$ holds in the state $s_m$ of the above execution if and only if there is a transition of $A_r$ that takes it from $v$ to $v'$ with the input $\ell(s_{m-1})$, and $v \in s_{m-1}$. Thus, for every state variable in $s_{j+1}$, there is a transition of $A_r$ with a corresponding state variable in $s_j$ with the input $\ell(s_j)$. The induction assumption states that for every state variable in $s_j$ a corresponding execution exists, and therefore an execution exists for every state variable in $s_{j+1}$, and by induction for every state variable in any state of the execution.

As a direct consequence, if $F_r \cap s_m \neq \emptyset$ for some $m$, then $\ell(s_k, \ldots, s_{m-1}) \in \mathcal{L}(r)$. Since $\pi \models_k^{\text{strong}} r \mapsto \phi$, $q^\phi$ must hold in $s_{m-1}$. Moreover, since $\ell(s_k, \ldots, s_n) \notin \mathcal{F}(r)$, and since every state of $A_r$ has an execution that leads to an accepting state, $s_n$ cannot contain any variables from $Q_r$, so the terminating condition holds. □

**Lemma 7.7.** *If $s_0, s_1, \ldots, s_n$ and $s'_0, s'_1, \ldots, s'_n$ are accepting executions of $T$ s.t. $\forall i, 0 \leq i \leq n : \ell(s_i) = \ell(s'_i)$, then $s_0 \cup s'_0, s_1 \cup s'_1, \ldots, s_n \cup s'_n$ is an accepting execution of $T$.*

*Proof.* The terminating condition obviously holds, since $s_n \cap Q_r = \emptyset$ and $s'_n \cap Q_r = \emptyset$. The transition relation can be shown to hold with:

$$
\begin{aligned}
& \bigwedge_{(v,\sigma,v')\in\delta_r} \Big( (v \in s_k \wedge \ell(s_k) = \sigma) \Rightarrow v' \in s_{k+1} \Big) \wedge \\
& \bigwedge_{(v,\sigma,v')\in\delta_r} \Big( (v \in s'_k \wedge \ell(s'_k) = \sigma) \Rightarrow v' \in s'_{k+1} \Big) \\
\Rightarrow & \\
& \bigwedge_{(v,\sigma,v')\in\delta_r} \Big( ((v \in s_k \wedge \ell(s_k) = \sigma) \Rightarrow v' \in s_{k+1}) \wedge \\
& \qquad ((v \in s'_k \wedge \ell(s'_k) = \sigma) \Rightarrow v' \in s'_{k+1}) \Big) \\
\Rightarrow & \\
& \bigwedge_{(v,\sigma,v')\in\delta_r} \Big( (v \in s_k \wedge \ell(s_k) = \sigma) \vee (v \in s'_k \wedge \ell(s'_k) = \sigma) \Rightarrow \\
& \qquad (v' \in s_{k+1}) \vee (v' \in s'_{k+1}) \Big) \\
\Rightarrow & \\
& \bigwedge_{(v,\sigma,v')\in\delta_r} \Big( (v \in s_k \vee v \in s'_k) \wedge (\ell(s_k) = \sigma) \Rightarrow \\
& \qquad (v' \in s_{k+1}) \vee (v' \in s'_{k+1}) \Big) \\
\Rightarrow & \\
& \bigwedge_{(v,\sigma,v')\in\delta_r} \Big( (v \in (s_k \cup s'_k) \wedge \ell(s_k) = \sigma) \Rightarrow \\
& \qquad v' \in (s_{k+1} \cup s'_{k+1}) \Big)
\end{aligned}
$$

and with:

$$
\begin{aligned}
& \Big( F_r \cap s_{k+1} \neq \emptyset \Rightarrow q^\phi \in s_k \Big) \wedge \Big( F_r \cap s'_{k+1} \neq \emptyset \Rightarrow q^\phi \in s'_k \Big) \\
\Rightarrow & \\
& \Big( F_r \cap s_{k+1} \neq \emptyset \vee F_r \cap s'_{k+1} \neq \emptyset \Big) \Rightarrow \Big( q^\phi \in s_k \vee q^\phi \in s'_k \Big) \\
\Rightarrow & \\
& \Big( F_r \cap (s_{k+1} \cup s'_{k+1}) \neq \emptyset \Big) \Rightarrow \Big( q^\phi \in (s_k \cup s'_k) \Big)
\end{aligned}
$$

□

### 7.5.2 $r \diamond\!\!\rightarrow \phi$

In the transducer construction we assume that the base language $\mathcal{L}(r)$ of $r$ is not empty. If $\mathcal{L}(r) = \emptyset$, the formula can be rewritten to $\bot$, and no transducer is constructed.

The intuition behind the transducer is a non-deterministically simulated automaton for the SERE, together with the enforcing that $\phi$ must hold when the simulated automaton accepts. The simulation is different from the tail implication, because only one match needs to be captured, instead of all possible matches.

Let $A_r = (Q_r, \Sigma, \delta_r, q_0, F_r)$ be a finite, non-deterministic automaton with $F_r$ reachable from every state and no $\varepsilon$-transitions, s.t. $\mathcal{L}(A_r) = \mathcal{L}(r)$ and $\Sigma = 2^{AP(r)}$. We can then construct a transducer $T = (Q, Q^{\text{in}}, q^{\text{out}}, I, F, \delta)$ for the tail conjunction, where:

- $Q = Q_r$,
- $Q^{\text{in}} = AP(r) \cup \{q^\phi\}$, where $q^\phi$ is the input variable to which the output of $T_\phi$ is connected,
- $q^{\text{out}} = q_0$, the initial state of $A_r$,
- $I = 2^{Q \cup Q^{\text{in}}}$,
- $F = \{s \mid s \subseteq (Q \cup Q^{\text{in}}), s \cap Q_r = \emptyset\}$, and
- $\delta = \left\{ (s, s') \;\middle|\; \bigwedge_{v \in Q_r} \left( v \in s \Rightarrow \bigvee_{(v,\sigma,v') \in \delta_r} \ell(s) = \sigma \wedge (v' \in s' \vee (v' \in F_r \wedge q^\phi \in s)) \right) \right\}$, where $s \subseteq (Q \cup Q^{\text{in}})$ and $s' \subseteq (Q \cup Q^{\text{in}})$.

The transducer for the entire sub-formula is obtained with the composition $T_\phi \rhd_{q^\phi} T$, where $T_\phi$ is a transducer for $\phi$. Intuitively, the transition relation takes care of the simulation of the automaton, except for the innermost parenthesis, which allow for the termination of the simulation if a match is found and $\phi$ holds. The state variables can be seen as a promise to find a match starting from that state of the automaton, and the final state constraint enforces that no such promise is left unfulfilled when the execution stops.

The above transducer only satisfies the following semantics: $\pi \models_i^{\text{strong}} r \diamond\!\!\rightarrow \phi \Leftrightarrow \exists j > i : s_i, \ldots, s_j \in \mathcal{L}(r)$ and $\pi \models_j^{\text{strong}} \phi$. The case where $\varepsilon \in \mathcal{L}(r)$ can be handled as with the transducer for tail implication, by rewriting the formula to $r \diamond\!\!\rightarrow \phi \vee \phi$ if $\varepsilon \in \mathcal{L}(r)$. The final part of the semantics is not relevant here, since strong semantics is always in use.

**Lemma 7.8.** *If $s_i, \ldots, s_j$ is an execution of the transducer, $q_i \in s_i$ for some $q_i \in Q_r$, and $\forall l, i \leq l < j : q^\phi \notin s_l$, then there is some $q_j \in s_j$ s.t. $q_j \in Q_r$ and the input sequence $\ell(s_i, \ldots, s_{j-1})$ takes $A_r$ from $q_i$ to $q_j$.*

*Proof.* Proof by induction over $j$:

- *Base case*: $i = j$ and $s_i = s_j$, so both claims hold trivially.
- *Induction assumption*: The claims hold when $j \leq k$.
- *Induction step*: There is an execution $s_i, \ldots, s_k$, $q_i \in s_i$, and $q_k \in s_k$. Because of the initial assumption, we also assume that $q^\phi \notin s_k$. Now we extend the execution with a single step to $s_{k+1}$. Note that if the transition relation does not permit such an extension, then the initial assumption that the execution exists is broken. Since $q^\phi \notin s_k$, the transition relation is equivalent to:

  $$q_k \in s_k \Rightarrow \bigvee_{(q_k,\sigma,q') \in \delta_r} \ell(s_k) = \sigma \wedge q' \in s_{k+1}$$

  This means that there exists some $q' \in Q_r$ s.t. $q' \in s_{k+1}$ and $(q_k, \sigma, q') \in \delta_r$. Now both claims of Lemma 7.8 hold for $s_{k+1}$, and by induction for any execution.

□

**Proposition 7.9.** *If $s_0, \ldots, s_i, \ldots, s_n$ is an execution of the transducer that accepts at $s_i$, then there exists some $j$ s.t. $i \leq j < n$, $q^\phi \in s_j$, and $\ell(s_i, \ldots, s_j) \in \mathcal{L}(r)$.*

*Proof.* Because of Lemma 7.8 and the terminating condition of the transducer, there must be some $s_j$ in the execution s.t. $q^\phi \in s_j$. Additionally, there is some $q_j \in s_j$ such that $\ell(s_i, \ldots, s_{j-1})$ takes $A_r$ from $q_0$ to $q_j$. On the other hand, if there is no transition $(q_j, \ell(s_j), q') \in \delta_r$ for some $q' \in F_r$, then the transition relation at that point is equivalent to:

$$q_j \in s_j \Rightarrow \bigvee_{(q_j, \sigma, q') \in \delta_r} \ell(s_j) = \sigma \wedge q' \in s_{j+1}$$

That would mean that there is some $q'$ in $s_{j+1}$, and because of Lemma 7.8 the terminating condition would not be satisfied. That means that there must exist some transition from $q_j$ to an accepting state of $A_r$ with the input $\ell(s_j)$, which in turn means that $\ell(s_i, \ldots, s_j) \in \mathcal{L}(r)$. □

**Proposition 7.10.** *Let $\pi = p_0, p_1, \ldots, p_{n-1}$ be a path. There exists an accepting execution $s_0, s_1, \ldots, s_n$ of $T$ for $\pi$ such that $\forall i, 0 \leq i < n : (\exists j, i \leq j < n : \ell(s_i, \ldots, s_j) \in \mathcal{L}(r)$ and $q^\phi \in s_j) \Rightarrow q_0 \in s_i$.*

*Proof.* Follows from Lemmas 7.11 and 7.12. □

**Lemma 7.11.** *If $\pi = p_0, p_1, \ldots, p_{n-1}$ is a path such that $\pi \models_k^{\text{strong}} r \diamond\!\!\rightarrow \phi$ for some $k$, then there exists an accepting execution $s_0, s_1, \ldots, s_n$ of $T$ for $\pi$ such that $q_0 \in s_k$.*

*Proof.* Because $\pi \models_k^{\text{strong}} r \diamond\!\!\rightarrow \phi$, there exists some $j, k \leq j < n$ s.t. $\pi \models_j^{\text{strong}} \phi$ and $p_k, \ldots, p_j \in \mathcal{L}(r)$. The execution can then be constructed as follows:

- $\ell(s_i) = s_i$ when $0 \leq i < k$, i.e. only atomic propositions hold in the states before $s_k$, all the state variables are false.
- $q_0 \in s_k$.
- $s_k \setminus \ell(s_k) = q_0$, i.e. the only state variable in $s_k$ is $q_0$.
- Let $(q_0, p_k, q_{k+1}), (q_{k+1}, p_{k+1}, q_{k+2}), \ldots, (q_j, p_j, q_{j+1})$ be a sequence of transitions that takes $A_r$ from $q_0$ to some $q_{j+1} \in F_r$ with the input $p_k, \ldots, p_j$. Such a sequence is guaranteed to exist, since $p_k, \ldots, p_j \in \mathcal{L}(r)$. For each $s_i$, $k \leq i \leq j$, $s_i \setminus \ell(s_i) = q_i$. In other words, the only state variable in each state from $s_k$ to $s_j$ is the state variable from the execution of $A_r$.
- For each $s_i$, $j < i \leq n$, $\ell(s_i) = s_i$, i.e. no state variables are true after $s_j$.

The terminating condition obviously holds for the above execution. The transition relation trivially holds for each pair of states $(s_i, s_{i+1})$ when $0 \leq i < k$, since no state variables are true, and therefore every implication in the transition relation is true.

Each pair of states $(s_i, s_{i+1})$, $k \leq i < j$ also satisfies the transition relation, since the state variables are taken from an execution of $A_r$, and therefore:

$$q_i \in s_i \Rightarrow (q_i, \sigma, q_{i+1}) \in \delta_r \wedge \ell(s) = \sigma \wedge q_{i+1} \in s_{i+1}$$

The rest of the implications are again trivially true, as no other state variables are true.

The pair of states $(s_j, s_{j+1})$ satisfies the transition relation, because there is a transition $(q_j, \ell(s_j), q_{j+1}) \in \delta_r$, and therefore:

$$q_j \in s_j \Rightarrow (q_j, \sigma, q_{j+1}) \in \delta_r \wedge \ell(s) = \sigma \wedge q_{j+1} \in F_r \wedge q^\phi \in s_j$$

Note that $q_{j+1} \notin s_{j+1}$.

For each pair $(s_i, s_{i+1})$, $j < i < n$, again no state variables are true, so the implications in the transition relation hold trivially. □

**Lemma 7.12.** *If $s_0, s_1, \ldots, s_n$ and $s'_0, s'_1, \ldots, s'_n$ are accepting executions of $T$ s.t. $\forall i, 0 \leq i \leq n : \ell(s_i) = \ell(s'_i)$, then $s_0 \cup s'_0, s_1 \cup s'_1, \ldots, s_n \cup s'_n$ is an accepting execution of $T$.*

*Proof.* The terminating condition obviously holds, since $s_n \cap Q_r = \emptyset$ and $s'_n \cap Q_r = \emptyset$. The transition relation can be shown to hold with:

$$\begin{aligned}
&\bigwedge_{v \in Q_r} \left( v \in s_k \Rightarrow \bigvee_{(v,\sigma,v') \in \delta_r} \ell(s_k) = \sigma \wedge (v' \in s_{k+1} \vee (v' \in F_r \wedge q^\phi \in s_k)) \right) \wedge \\
&\bigwedge_{v \in Q_r} \left( v \in s'_k \Rightarrow \bigvee_{(v,\sigma,v') \in \delta_r} \ell(s'_k) = \sigma \wedge (v' \in s'_{k+1} \vee (v' \in F_r \wedge q^\phi \in s'_k)) \right) \\
\Rightarrow \\
&\bigwedge_{v \in Q_r} \left( \left( v \in s_k \Rightarrow \bigvee_{(v,\sigma,v') \in \delta_r} \ell(s_k) = \sigma \wedge (v' \in s_{k+1} \vee (v' \in F_r \wedge q^\phi \in s_k)) \right) \wedge \right. \\
&\qquad \left. \left( v \in s'_k \Rightarrow \bigvee_{(v,\sigma,v') \in \delta_r} \ell(s'_k) = \sigma \wedge (v' \in s'_{k+1} \vee (v' \in F_r \wedge q^\phi \in s'_k)) \right) \right) \\
\Rightarrow \\
&\bigwedge_{v \in Q_r} \left( (v \in s_k \vee v \in s'_k) \Rightarrow \right. \\
&\qquad \left( \bigvee_{(v,\sigma,v') \in \delta_r} \ell(s_k) = \sigma \wedge (v' \in s_{k+1} \vee (v' \in F_r \wedge q^\phi \in s_k)) \right) \vee \\
&\qquad \left. \left( \bigvee_{(v,\sigma,v') \in \delta_r} \ell(s'_k) = \sigma \wedge (v' \in s'_{k+1} \vee (v' \in F_r \wedge q^\phi \in s'_k)) \right) \right) \\
\Rightarrow \\
&\bigwedge_{v \in Q_r} \left( (v \in s_k \vee v \in s'_k) \Rightarrow \right. \\
&\qquad \left( \bigvee_{(v,\sigma,v') \in \delta_r} \ell(s_k) = \sigma \wedge (v' \in s_{k+1} \vee (v' \in F_r \wedge q^\phi \in s_k)) \vee \right. \\
&\qquad \left. \left. \ell(s'_k) = \sigma \wedge (v' \in s'_{k+1} \vee (v' \in F_r \wedge q^\phi \in s'_k)) \right) \right)
\end{aligned}$$

$$
\begin{aligned}
&\Rightarrow \\
&\quad \bigwedge_{v \in Q_r} \Big( (v \in s_k \vee v \in s'_k) \Rightarrow \\
&\qquad \Big( \bigvee_{(v,\sigma,v') \in \delta_r} \ell(s_k) = \sigma \wedge ((v' \in s_{k+1} \vee (v' \in F_r \wedge q^\phi \in s_k)) \vee \\
&\qquad (v' \in s'_{k+1} \vee (v' \in F_r \wedge q^\phi \in s'_k))) \Big) \Big) \\
&\Rightarrow \\
&\quad \bigwedge_{v \in Q_r} \Big( (v \in s_k \vee v \in s'_k) \Rightarrow \\
&\qquad \Big( \bigvee_{(v,\sigma,v') \in \delta_r} \ell(s_k) = \sigma \wedge ((v' \in s_{k+1} \vee v' \in s'_{k+1}) \vee (v' \in F_r \wedge q^\phi \in s_k)) \Big) \Big) \\
&\Rightarrow \\
&\quad \bigwedge_{v \in Q_r} \Big( (v \in s_k \cup s'_k) \Rightarrow \\
&\qquad \Big( \bigvee_{(v,\sigma,v') \in \delta_r} \ell(s_k) = \sigma \wedge ((v' \in s_{k+1} \cup s'_{k+1}) \vee (v' \in F_r \wedge q^\phi \in s_k)) \Big) \Big)
\end{aligned}
$$

□

### 7.5.3 $\phi_1 \mathbf{U} \phi_2$

The transducer for the until operator is $T = (Q, Q^{\text{in}}, q^{\text{out}}, I, F, \delta)$, where:

- $Q = \{q^{\mathbf{U}}\}$,
- $Q^{\text{in}} = \{q^{\text{left}}, q^{\text{right}}\}$,
- $q^{\text{out}} = q^{\mathbf{U}}$,
- $I = 2^{Q \cup Q^{\text{in}}}$,
- $F = \{s \mid s \subseteq (Q \cup Q^{\text{in}}), q^{\mathbf{U}} \in s \Leftrightarrow q^{\text{right}} \in s\}$, and
- $\delta = \left\{ (s, s') \,\middle|\, q^{\mathbf{U}} \in s \Leftrightarrow \left( q^{\text{right}} \in s \vee (q^{\text{left}} \in s \wedge q^{\mathbf{U}} \in s') \right) \right\}$, where $s \subseteq (Q \cup Q^{\text{in}})$ and $s' \subseteq (Q \cup Q^{\text{in}})$.

The transducer for the entire sub-formula is obtained with the compositions $T_2 \rhd_{q^{\text{right}}} (T_1 \rhd_{q^{\text{left}}} T)$, where $T_1$ is a transducer for $\phi_1$ and $T_2$ is a transducer for $\phi_2$.

Because strong semantics is in use, the semantics of the until-operator is reduced to: $\pi \models_i^{\text{strong}} \phi_1 \mathbf{U} \phi_2 \Leftrightarrow \exists j \geq i : \pi \models_j^{\text{strong}} \phi_2$ and $\forall k, i \leq k < j : \pi \models_k^{\text{strong}} \phi_1$.

**Proposition 7.13.** *For any path $\pi = p_0, p_1, \ldots, p_n$, and for every execution $s_0, s_1, \ldots, s_n$ of $T$ for $\pi$, $q^{\mathbf{U}} \in s_i$ if and only if $\pi \models_i^{\text{strong}} \phi_1 \mathbf{U} \phi_2$.*

*Proof.* The proposition can be proven by backward induction from the final state of the execution.

- *Base case*: In the final state the terminating condition is equivalent to the semantics: since there is no next state in the execution, $\pi \models_n^{\text{strong}} \phi_1 \mathbf{U} \phi_2$ if and only if $q^{\text{right}} \in s_n$.

- *Induction assumption*: The proposition holds for states $s_{n-k}$ with some limit for $k$.

- *Induction step*: The transition relation states that $q^{\mathbf{U}} \in s_{n-(k+1)} \Leftrightarrow (q^{\text{right}} \in s_{n-(k+1)} \vee (q^{\text{left}} \in s_{n-(k+1)} \wedge q^{\mathbf{U}} \in s_{n-k}))$. Clearly $q^{\text{right}} \in s_{n-(k+1)}$ implies $\pi \models_{n-(k+1)}^{\text{strong}} \phi_1 \mathbf{U} \phi_2$. Moreover, since $q^{\mathbf{U}} \in s_{n-k}$ if and only if $\pi \models_{n-k}^{\text{strong}} \phi_1 \mathbf{U} \phi_2$, $q^{\text{left}} \in s_{n-(k+1)} \wedge q^{\mathbf{U}} \in s_{n-k}$ implies both $\exists j > i : q^{\text{right}} \in s_j$ and $\forall l : n-(k+1) \leq l < j : q^{\text{left}} \in s_l$. Therefore $q^{\text{left}} \in s_{n-(k+1)} \wedge q^{\mathbf{U}} \in s_{n-k}$ implies $\pi \models_{n-(k+1)}^{\text{strong}} \phi_1 \mathbf{U} \phi_2$.

  On the other hand, $\pi \models_{n-(k+1)}^{\text{strong}} \phi_1 \mathbf{U} \phi_2$ implies that either $\pi \models_{n-(k+1)}^{\text{strong}} \phi_2$ or $\exists j > i : q^{\text{right}} \in s_j$ and $\forall l : n-(l+1) \leq l < j : q^{\text{left}} \in s_l$. The latter implies that $\pi \models_{n-k}^{\text{strong}} \phi_1 \mathbf{U} \phi_2$, and therefore $q^{\mathbf{U}} \in s_{n-k}$. As a consequence, $\pi \models_{n-(k+1)}^{\text{strong}} \phi_1 \mathbf{U} \phi_2$ implies $q^{\text{right}} \in s_{n-(k+1)} \vee (q^{\text{left}} \in s_{n-(k+1)} \wedge q^{\mathbf{U}} \in s_{n-k})$.

  Now we have established that $q^{\mathbf{U}} \in s_{n-(k+1)} \Leftrightarrow q^{\text{right}} \in s_{n-(k+1)} \vee (q^{\text{left}} \in s_{n-(k+1)} \wedge q^{\mathbf{U}} \in s_{n-k})$ is equivalent to $\pi \models_{n-(k+1)}^{\text{strong}} \phi_1 \mathbf{U} \phi_2$, so the proposition holds for $k+1$, and by induction for any $k$.

□

### 7.5.4 $\phi_1 \mathbf{R} \phi_2$

The transducer for the releases operator is $T = (Q, Q^{\text{in}}, q^{\text{out}}, I, F, \delta)$, where:

- $Q = \{q^{\mathbf{R}}\}$,
- $Q^{\text{in}} = \{q^{\text{left}}, q^{\text{right}}\}$,
- $q^{\text{out}} = q^{\mathbf{R}}$,
- $I = 2^{Q \cup Q^{\text{in}}}$,
- $F = \{s \mid s \subseteq (Q \cup Q^{\text{in}}), q^{\mathbf{R}} \in s \Leftrightarrow (q^{\text{left}} \in s \wedge q^{\text{right}} \in s)\}$, and
- $\delta = \left\{(s, s') \,\middle|\, q^{\mathbf{R}} \in s \Leftrightarrow \left(q^{\text{right}} \in s \wedge (q^{\text{left}} \in s \vee q^{\mathbf{R}} \in s')\right)\right\}$, where $s \subseteq (Q \cup Q^{\text{in}})$ and $s' \subseteq (Q \cup Q^{\text{in}})$.

The transducer for the entire sub-formula is obtained with the compositions $T_2 \triangleright_{q^{\text{right}}} (T_1 \triangleright_{q^{\text{left}}} T)$, where $T_1$ is a transducer for $\phi_1$ and $T_2$ is a transducer for $\phi_2$.

Because strong semantics is in use and the paths are finite, the semantics of the releases-operator is reduced to: $\pi \models_i^{\text{strong}} \phi_1 \mathbf{R} \phi_2 \Leftrightarrow \exists k, i \leq k \leq |\pi| : \pi \models_k^{\text{strong}} \phi_1$ and $\forall j, i \leq j \leq k : \pi \models_j^{\text{strong}} \phi_2$.

**Proposition 7.14.** *For any path $\pi = p_0, p_1, \ldots, p_n$, and for every execution $s_0, s_1, \ldots, s_n$ of $T$ for $\pi$, $q^{\mathbf{R}} \in s_i$ if and only if $\pi \models_i^{\text{strong}} \phi_1 \mathbf{R} \phi_2$.*

*Proof.* The proposition can be proven by backward induction from the final state of the execution.

- *Base case*: In the final state the terminating condition is equivalent to the semantics: since there is no next state in the execution, $\pi \models_n^{\text{strong}} \phi_1 \mathbf{R} \phi_2$ if and only if $q^{\text{left}} \in s_n \wedge q^{\text{right}} \in s_n$.
- *Induction assumption*: The proposition holds for states $s_{n-k}$ with some limit for $k$.
- *Induction step*: The transition relation states that $q^{\mathbf{R}} \in s_{n-(k+1)} \Leftrightarrow (q^{\text{right}} \in s_{n-(k+1)} \wedge (q^{\text{left}} \in s_{n-(k+1)} \vee q^{\mathbf{R}} \in s_{n-k}))$. Dividing the right side of the equivalence, we see that $q^{\text{right}} \in s_{n-(k+1)} \wedge q^{\text{left}} \in s_{n-(k+1)}$ implies $\pi \models_{n-(k+1)}^{\text{strong}} \phi_1 \mathbf{R} \phi_2$. Additionally, since $q^{\mathbf{R}} \in s_{n-k}$ if and only if $\pi \models_{n-k}^{\text{strong}} \phi_1 \mathbf{R} \phi_2$, the other part, $q^{\text{right}} \in s_{n-(k+1)} \wedge q^{\mathbf{R}} \in s_{n-k}$ implies both $\exists l, n-k \leq l \leq n : q^{\text{left}} \in s_l$ and $\forall j, n-(k+1) \leq j \leq n : q^{\text{right}} \in s_j$. Therefore $q^{\text{right}} \in s_{n-(k+1)} \wedge q^{\mathbf{R}} \in s_{n-k}$ implies $\pi \models_{n-(k+1)}^{\text{strong}} \phi_1 \mathbf{R} \phi_2$.

  On the other hand, $\pi \models_{n-(k+1)}^{\text{strong}} \phi_1 \mathbf{R} \phi_2$ implies that either $\pi \models_{n-(k+1)}^{\text{strong}} \phi_1$ and $\pi \models_{n-(k+1)}^{\text{strong}} \phi_2$, or that $\exists l, n-k \leq l \leq n : q^{\text{left}} \in s_l$ and $\forall j : n-(l+1) \leq j \leq l : q^{\text{right}} \in s_l$. The latter implies that $\pi \models_{n-k}^{\text{strong}} \phi_1 \mathbf{R} \phi_2$, and therefore $q^{\mathbf{R}} \in s_{n-k}$. As a consequence, $\pi \models_{n-(k+1)}^{\text{strong}} \phi_1 \mathbf{R} \phi_2$ implies $q^{\text{right}} \in s_{n-(k+1)} \wedge (q^{\text{left}} \in s_{n-(k+1)} \vee q^{\mathbf{R}} \in s_{n-k})$.

  Now we have established that $q^{\mathbf{R}} \in s_{n-(k+1)} \Leftrightarrow q^{\text{right}} \in s_{n-(k+1)} \wedge (q^{\text{left}} \in s_{n-(k+1)} \vee q^{\mathbf{U}} \in s_{n-k})$ is equivalent to $\pi \models_{n-(k+1)}^{\text{strong}} \phi_1 \mathbf{R} \phi_2$, so the proposition holds for $k+1$, and by induction for any $k$.

□

### 7.5.5 $\mathbf{X}\ \phi$

The transducer for the next operator is $T = (Q, Q^{\text{in}}, q^{\text{out}}, I, F, \delta)$, where:

- $Q = \{q^{\mathbf{X}}\}$,
- $Q^{\text{in}} = \{q^{\phi}\}$,
- $q^{out} = q^{\mathbf{X}}$,
- $I = 2^{Q \cup Q^{\text{in}}}$,
- $F = \{\emptyset, \{q^{\phi}\}\}$, and
- $\delta = \{(s, s') \mid q^{\mathbf{X}} \in s \Leftrightarrow q^{\text{in}} \in s'\}$, where $s \subseteq (Q \cup Q^{\text{in}})$ and $s \subseteq (Q \cup Q^{\text{in}})$.

The transducer for the entire sub-formula is obtained with the composition $T_\phi \rhd_{q^\phi} T$, where $T_\phi$ is a transducer for $\phi$.

**Proposition 7.15.** *For any path $\pi = p_0, p_1, \ldots, p_n$, and for every execution $s_0, s_1, \ldots, s_n$ of $T$ for $\pi$, $q^{\mathbf{X}} \in s_i$ if and only if $\pi \models_i^{\text{strong}} \mathbf{X}![1]\ \phi$.*

*Proof.* In the final state $s_n$, $q^{\mathbf{X}} \notin s_n$. In every other state $s_i$, the transition relation states that $q^{\mathbf{X}} s_i \Leftrightarrow q^{\phi} \in s_{i+1}$. □

### 7.5.6 Transducers for $\wedge$ and $\neg$

Transducers for the conjunction and the negation can be somewhat trivially constructed by having input variables for the sub-formulae and adding the constraints to the initial and final state sets and the transition relation. The output variable of the conjunction transducer is constrained to be true exactly in the states where both input variables are true, and the output variable for the negation transducer is true exactly in the states where its input variable is not.

# 8 OBSERVER IMPLEMENTATION

Model checking PSL with the transducers from the previous chapter can be done by constructing a NuSMV module from the transducer of the desired PSL formula. Transducers are fairly straightforward to translate into NuSMV modules: the state and input variables can be used in the NuSMV module without any changes, and the transition relation is fairly straightforward to implement using the NuSMV's `TRANS`-statement. Also the initial states can be enforced with the `INIT`-statement. The final state condition needs some attention, though. NuSMV does not have have a straightforward way of enforcing constraints on final states, but the same effect can be achieved by the following construction. A special purpose variable `fs` is added to the module. The `fs` variable is restricted so that it can get the value *true* if and only if the final state conditions hold. This is done by adding to the transition relation the statement $s \notin F \Rightarrow \neg\texttt{fs}$, where $s$ is any state of the system and $F$ is the set of final states.

The main idea of doing model checking with these transducers is to reduce the problem of whether the property holds to a problem of whether a state in which `fs` holds is reachable. The transducer module can be seen as an observer automaton: it is run together with the actual model, and a conclusion is made based on the reachability of some states in the observer. In this case, a set of states are labelled as bad states: if they are reachable, the property does not hold. Moreover, the path to such a state is a counterexample of the property being checked for.

The reduction to reachability checking is done by making an observer module for the negation of the property that is being checked for, and adding an initial constraint that forces the output variable for the observer to true. Then, an invariant specification `INVARSPEC !fs` is added and checked for. This achieves the desired result, since the semantics of a PSL formula is such that every path of the model must satisfy it. That means that the specification is broken if and only if there is some path in the model where the negation of the specification holds in the first state. Furthermore, since the initial state is such that the output variable for the observer module is true, the negation of the specification holds if and only if it is possible for the observer to reach a valid end state.

Figure 8.1 shows the components of the process. The model and the PSL formula are written by the user of the system. The PSL formula then goes to the PSL parser in NuSMV, which is the one that IBM distributes freely on their PSL/Sugar site [2]. The parsed PSL formula is traversed by the formula translator that generates the observer module with the rules described in the previous chapter. The SEREs in the formula are translated to finite state machines by a separate component.

The actual implementation that was done for this work is a proof-of-concept, whose main purpose is to verify the feasibility of such an implementation and to allow experimentation with the algorithm. It is not very tightly integrated to NuSMV. The formula translator in the Figure 8.1 is run after the initial generation of the parse tree by NuSMV, and the parse tree is altered as a result. It only supports verification of a single PSL formula at a time, and only

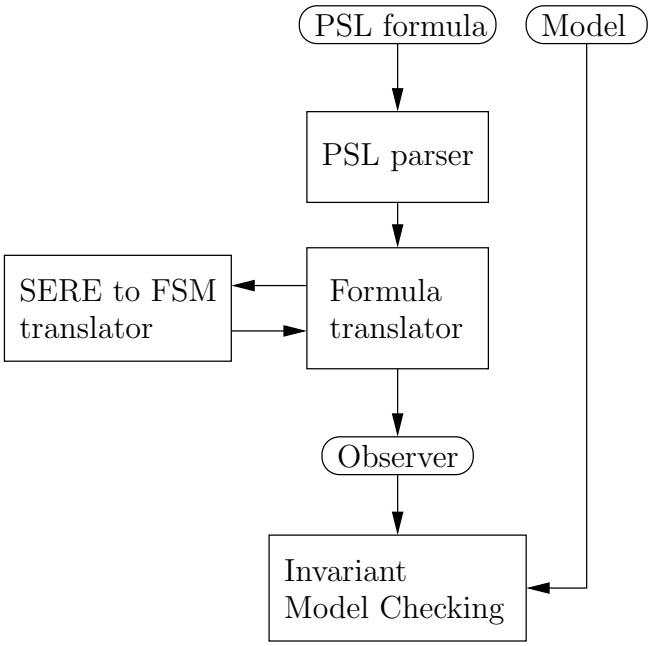



Figure 8.1: The components of model checking with the observer. The ovals in the figure are input to a component. The formula translator and the SERE to FSM translator are the components implemented in this work.

boolean variables are supported as atomic propositions. Still, it is sufficient to perform meaningful experiments on the algorithm, and generates minimal overhead in the verification workflow.

## 8.1 CONVERTING SERES TO FINITE STATE AUTOMATA

The finite state automata are generated recursively from the SEREs by first generating the automata for the sub-expressions of the SERE and then combining them into an automaton for the entire SERE. For the union, concatenation, and repetition of SEREs this can be efficiently done by just adding $\varepsilon$-transitions between the automata for the sub-expressions. Concatenation with overlap and intersection of SEREs are more complicated, and the structure of the automata need to be changed.

The alphabet of the automata is $2^{AP}$, where $AP$ is the set of atomic propositions in the SERE. To simplify the construction, a single transition here is used to represent a group of transitions. The labels on transitions in this chapter are conjunctions of literals $l \in L$, where $L = \{p \mid p \in AP\} \cup \{\neg p \mid p \in AP\}$. A transition with the label $l_1 \wedge l_2 \wedge \ldots \wedge l_n$, where each $l_i, \ldots \in L$, represents the group of transitions where the positive literals are present, and the negative literals are not. To express this more formally, a transition $(s, (l_1 \wedge l_2 \wedge \ldots \wedge l_n), s')$ whose label is a conjunction of literals in $L$ can be converted to a set of transitions: $\{(s, \sigma, s') \mid \forall i, 1 \leq i \leq n : l \in AP \Rightarrow l \in \sigma\} \cap \{(s, \sigma, s') \mid \forall i, 1 \leq i \leq n : l = \neg p, p \in AP \Rightarrow p \notin \sigma\}$.

The union of two sub-automata $A_1$ and $A_2$ can be combined into a single automaton $A$ by adding $\varepsilon$-transitions from the initial state of $A$ to the initial states of $A_1$ and $A_2$, and keeping the accepting states of both as accepting.

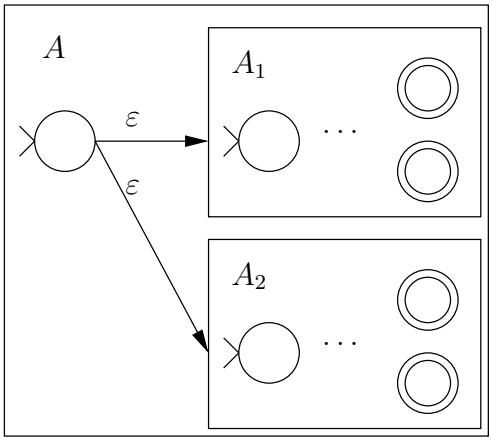



Figure 8.2: The union of two automata is constructed by adding an $\varepsilon$-transition from the new initial state to the initial state of each sub-automaton.

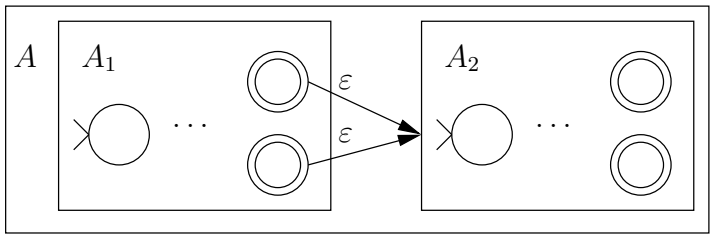



Figure 8.3: The concatenation of two automata is constructed by adding an $\varepsilon$-transition from each accepting state of the first automaton to the initial state of the second.

This combination is illustrated in Figure 8.2.

The concatenation of two sub-automata $A_1$ and $A_2$ can be combined into a single automaton $A$ by adding an $\varepsilon$-transition from each accepting state of $A_1$ to the initial state of $A_2$. The new initial state is the initial state of $A_1$ and the new accepting states are those of $A_2$. The combination is illustrated in Figure 8.3.

The Kleene-closure automaton $A$ of a sub-automaton $A'$ can be formed by adding a new, accepting initial state, and by adding an $\varepsilon$-transition from the new initial state to the initial state of $A'$ and $\varepsilon$-transitions from the accepting states of $A'$ to the initial state of $A'$. Figure 8.4 illustrates this.

The concatenation with an overlap is done by first removing the $\varepsilon$-transitions from the component automatons $A_1$ and $A_2$. Then all the transitions to the accepting states of $A_1$ and all the transitions from the initial state of $A_2$ are processed. These are combined pair-wise so that from every transition to an accepting state in $A_1$ gets paired with every transition from the initial state of $A_2$. The combined transitions get labels that are conjunctions of the labels from the transitions that are combined. Figure 8.5 illustrates this.

The intersection of two automata $A_1$ and $A_2$ is done by first removing the $\varepsilon$-transitions from both, and then pairing each state of $A_1$ with each state of $A_2$. The transitions from each new state are formed by taking each transition from the state from $A_1$ and pairing it with each transition from the state from $A_2$. The labels on the resulting transitions are conjunctions of the labels on the transitions that are combined. The new initial state is the pair formed

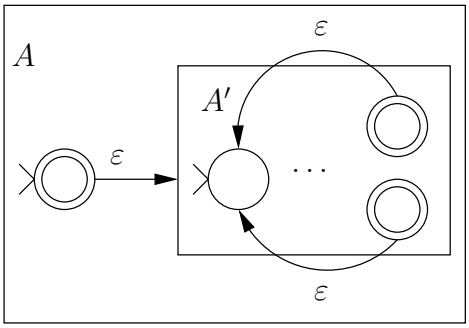



Figure 8.4: The Kleene-closure of an automaton is formed by adding $\varepsilon$-transitions from a new initial state to the old initial state, as well as $\varepsilon$-transitions from the accepting states to the old initial state.

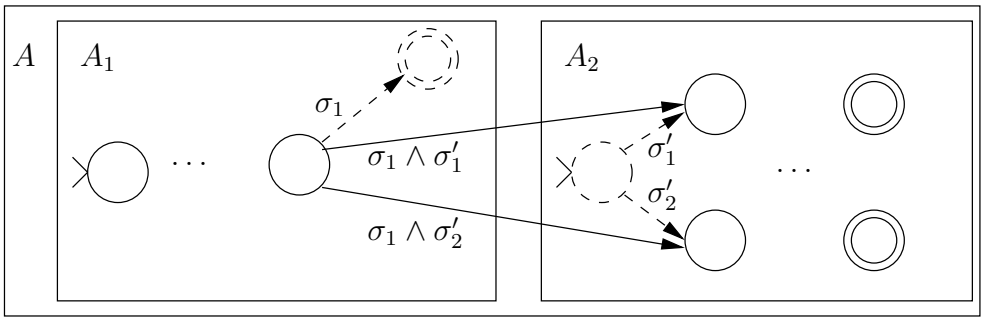



Figure 8.5: The concatenation with an overlap is done by combining every transition that leads to an accepting state in $A_1$ with every transition that starts from the initial state in $A_2$.

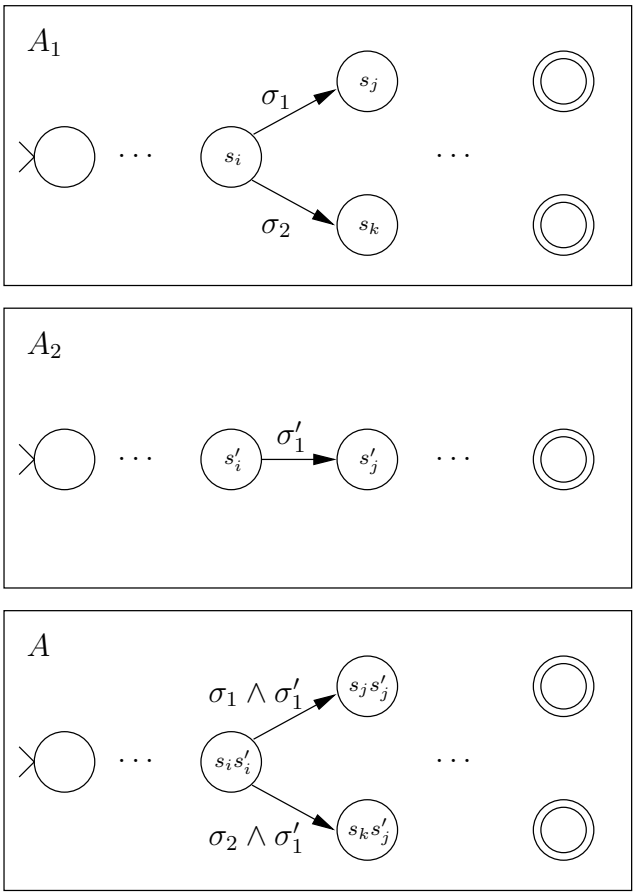



Figure 8.6: The intersection of automata is done by pairing each state from one automaton with each state from the other. The transitions are formed by taking conjunctions of the transitions from both states of the pair.

by the initial states of $A_1$ and $A_2$. The new accepting states are such states where each of the pair was an accepting state in its original automaton. This is illustrated by Figure 8.6.

### 8.1.1 Removing $\varepsilon$-transitions

The $\varepsilon$-transitions can be removed by following $\varepsilon$-transitions from each state until a visible transition is reached. To avoid problems with $\varepsilon$-loops, the set of states visited this way is stored, and transitions that lead to visited states are ignored. A transition is then added with the label of the visible transition, from the state the search started, to the state where the visible transition goes to. This is done for each state and for each visible transition that can be reached with $\varepsilon$-transitions. Then the $\varepsilon$-transitions are removed, along with states that have become unreachable this way.

## 8.2 PRODUCING THE OBSERVER MODULE

The observer module for a PSL formula is constructed by first negating the formula and converting it to the positive normal form. Then, a transducer

is recursively generated for any sub-formulae and any SEREs converted to automata as described in Section 8.1. From these a transducer is constructed as described in Chapter 7. The transducer is then converted to a NuSMV module by enforcing the transition relation with `TRANS`-statements, forcing the output of the top-level formula transducer to true with an `INIT`-statement, and adding an invariant specification that states that a valid final state of the transducer cannot be reached.

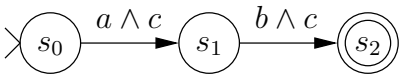
For example the PSL formula $\mathbf{G}\ a \cdot b \cap c[*] \mapsto \mathbf{G}\ d$[1] translates into an observer module in the following way:

1. The negated positive normal form of the formula is $\mathbf{F}\ a \cdot b \cap c[*] \diamond\!\!\rightarrow \mathbf{F}\ \neg d$.

2. The observer for $\mathbf{F}\ \neg d$ is:

```
MODULE observer(d)
VAR
  future_1 : boolean;
  fs : boolean;
INVAR
  (!d -> future_1)
INVAR
  (future_1 -> (!d | !fs))
TRANS
  (future_1 <-> (!d | next(future_1)))
```

3. The automaton for the SERE $a \cdot b \cap c[*]$ is:




4. The observer for $a \cdot b \cap c[*] \diamond\!\!\rightarrow \mathbf{F}\ \neg d$ is then:

```
MODULE observer(a, b, c, d)
VAR
  future_1 : boolean;
  s0 : boolean;
  s1 : boolean;
  s2 : boolean;
  fs : boolean;
INVAR
  (!d -> future_1)
INVAR
  (future_1 -> (!d | !fs))
INVAR
  ((s2 | (s0 | s1)) -> !fs)
TRANS
  (future_1 <-> (!d | next(future_1)))
TRANS
```

[1] $r[*]$ is shorthand for $[*0] \cup r[+]$.

```
  ((s0 -> ((a & c) & next(s1))) &
   (s1 -> ((b & c) & future_1)))
```

5. Finally, adding the sub-formula for the future-operator, the `INIT`-condition and the invariant specification, the result is:

```
MODULE observer(a, b, c, d)
VAR
  future_1 : boolean;
  s0 : boolean;
  s1 : boolean;
  s2 : boolean;
  future_2 : boolean;
  fs : boolean;
INVAR
  (!d -> future_1)
INVAR
  (future_1 -> (!d | !fs))
INVAR
  ((s2 | (s0 | s1)) -> !fs)
INVAR
  (s0 -> future_2)
INVAR
  (future_2 -> (s0 | !fs))
TRANS
  (future_1 <-> (!d | next(future_1)))
TRANS
  ((s0 -> ((a & c) & next(s1))) &
   (s1 -> ((b & c) & future_1)))
TRANS
  (future_2 <-> (s0 | next(future_2)))
INIT
  future_2
INVARSPEC !fs
```

# 9 EXPERIMENTS

The prototype implementation was subjected to a series of experiments to measure its performance and capabilities. The experiments were conducted on a Debian Linux machine with an Intel Core 2 CPU running at 1.86 GHz and a main memory of 2 GiB. The memory limit for the process was set to 512 MiB and running time limit was set to 300 seconds (5 minutes).

The test models for the experiments were randomly generated Kripke structures represented as NuSMV modules. The random generation was done by the same test script that was used in [22]. The number of atomic propositions was six and the number of expected edges from a state was three in every experiment.

The test formulae for the experiments were randomly generated PSL properties that were generated with the same algorithm that is used by the LTL-to-Büchi translator testbench, described in [29]. A slight modification was made to the algorithm, however, to obtain legal PSL formulae. Unlike in LTL, PSL has restrictions on what operators can appear at what points of the formula, so when selecting a new random operator, only the valid choices are considered.

The default BDD-based invariant checking of NuSMV was used in the experiments. A set of test runs was made for formula sizes 10, 20, 30, 50, and 100. Here the size of the formula means the number of nodes in its parse tree. For each formula size a set of test runs was made with model sizes 10, 100, 1000, 2000, 3000, 5000, and 10000. Here the model size means the number of states in the Kripke structure. Each set of test runs consisted of 20 randomly generated pairs of a model and a formula.

## 9.1 TEST RUN RESULTS

Figures 9.1 to 9.3 present the minimum, median, and maximum run times in the experiment set. Figure 9.4 presents the minimum, median, and maximum run times of test sets with model size 5000. Figures 9.5 to 9.6 present the number of experiments where either the time limit or the memory limit was exceeded. No resource limits were exceeded in runs with formula size 20. The runs where a resource limit was reached were not included in figures 9.1 to 9.4.

The figures 9.1 to 9.3 show the expected increase of running time with the increase of model size. The increase of maximum running times in some test batches seems to be due to a randomly generated formula that contains lots of SERE intersections. Since the SERE intersection involves an exponential blow-up in the size of the automaton for it, the observer construction time and memory requirement grows exponentially with respect to the formula, and much higher running times are expected for such formulae.

Interestingly enough, the minimum and median running times in Figure 9.4 are flat across the different formula sizes, suggesting that the model size greatly dominates the running time with these parameters. The increase in the maximum running time again shows that with many intersection op-

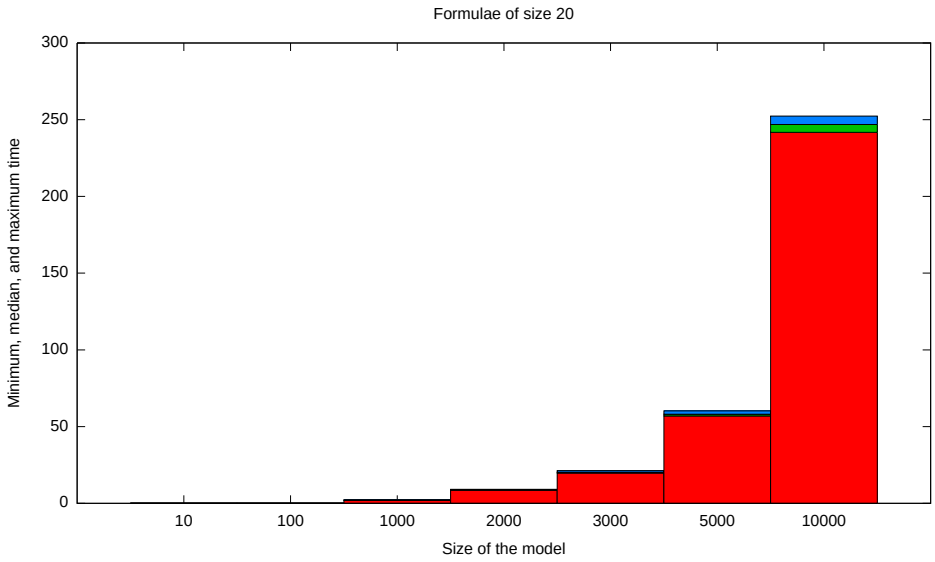



Figure 9.1: The minimum, median, and maximum times of the test batches with formula size 20.

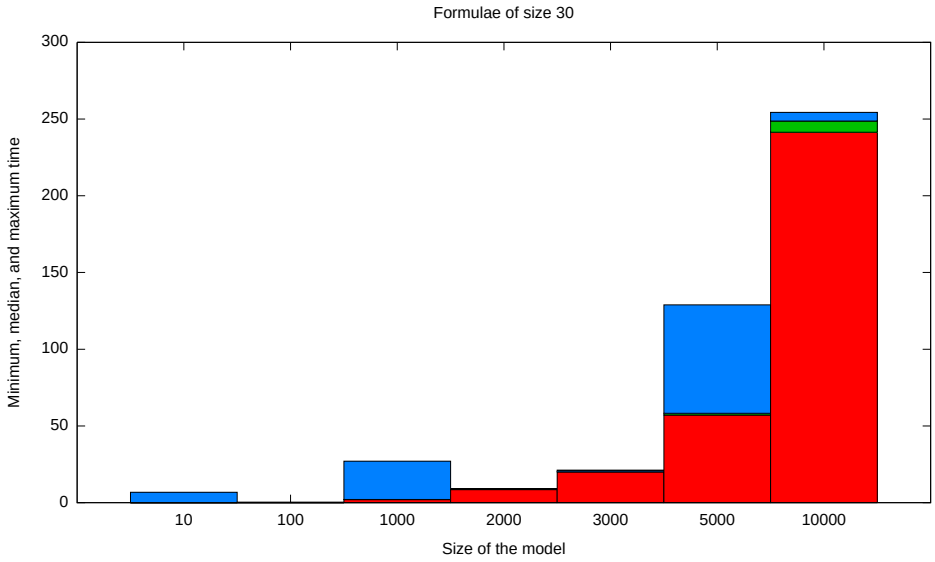



Figure 9.2: The minimum, median, and maximum times of the test batches with formula size 30.

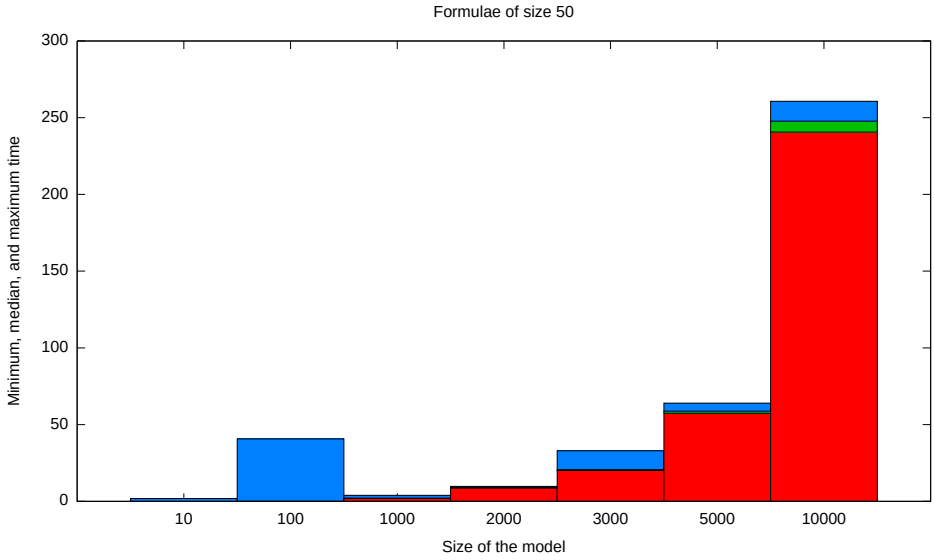



Figure 9.3: The minimum, median, and maximum times of the test batches with formula size 50.

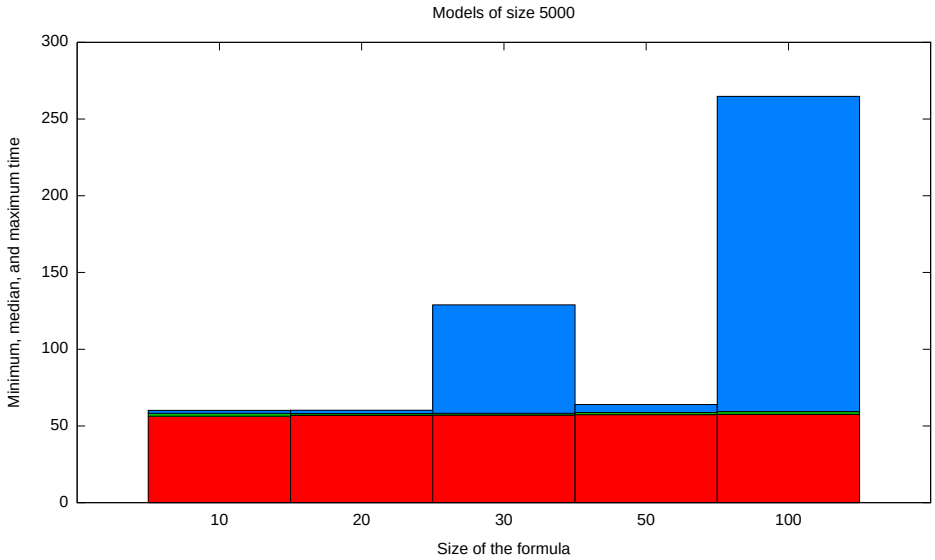



Figure 9.4: The minimum, median, and maximum times of the test batches with model size 5000.

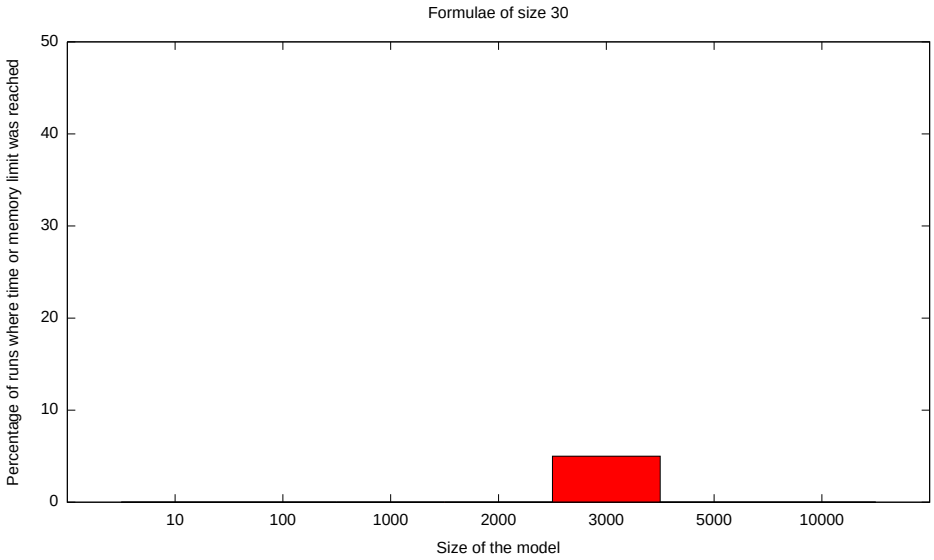



Figure 9.5: The number of runs in the test batches where either the time limit or memory limit was reached with the formula size 30.

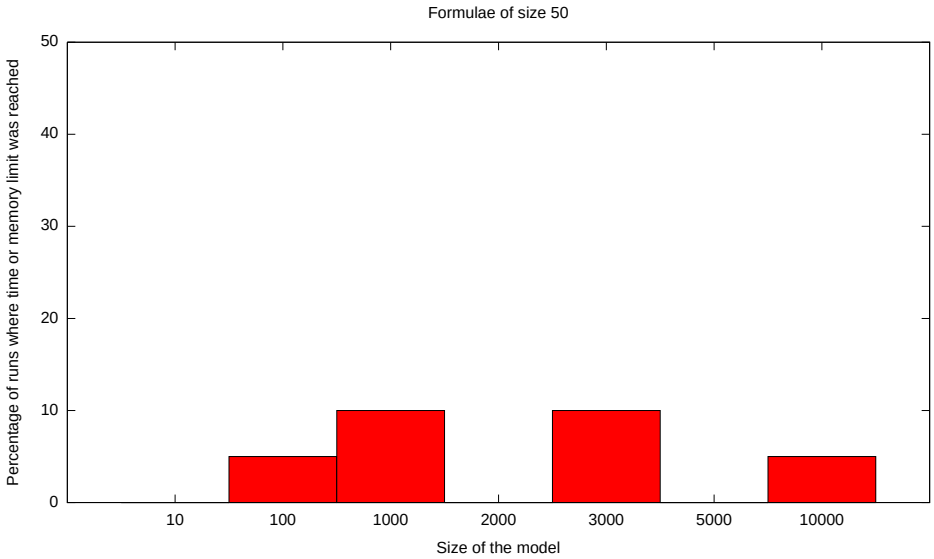



Figure 9.6: The number of runs in the test batches where either the time limit or memory limit was reached with the formula size 50.

erators, the formula becomes the dominating factor in running time.

Figures 9.5 to 9.6 show that the number of runs where the memory or time limit was reached does not seem to depend on the model size. This further indicates that the heavy use of the SERE intersection operator is the main cause for high resource usage in these experiments.

# 10 CONCLUSION

In this work we have covered evaluating PSL formulae on finite paths. Chapter 4 introduced the basics of the PSL language, and Chapter 5 formalised the semantics of the language. Additional operators were defined to make rewriting PSL formulae to the positive normal form possible. The strong, neutral, and weak semantics are all covered in the chapter. Chapter 6 discusses the use of the semantics in Chapter 5 for finite paths, and introduces the rewriting rules needed to convert formulae to the positive normal form.

In Chapter 7 transducers, a formalism similar to symbolic finite state automata was introduced, along with definitions of acceptance of a path and composition of transducers. Then, a method for compositionally constructing transducers from PSL formulae was described, along with proofs of its correctness.

Chapter 8 describes the prototype implementation of the translation described in Chapter 7 into the open-source model checker NuSMV. Implementation issues were discussed, along with the conversion of SEREs to finite state automata. Chapter 9 describes the test setup that was used to run experiments on the implementation, and presents the results and their interpretation.

The test runs show that this method of model checking PSL is indeed feasible. With the parameters used in the experiments, the model size is the dominating factor in the verification times. SERE intersection, which causes an exponential state explosion problem, can cause considerable variations to the total running times, however, since converting a SERE with many intersection operators to a finite state machine can take a long time, and quickly consume the available memory resources.

The main contributions of this work are the revision of the semantics in the proposal for the revised PSL standard, the translation of PSL properties to symbolic finite state automata, the correctness proof of the translation with respect to the presented semantics, and the prototype implementation for NuSMV. To the author's knowledge, this is the first freely available implementation for PSL model checking.

## 10.1 FUTURE WORK

Future work for the PSL observer could include a complete, integrated implementation of the algorithm to NuSMV, and a comparison of the implementation with other methods of model checking PSL. Unfortunately the currently released version (2.4.3) of NuSMV does not support full model checking of PSL, so a direct comparison with alternative algorithms cannot be made at this time. Moreover, benchmarking against other model checking tools is not directly comparable, since the performance of each tool varies according to the modelling formalism used. Additionally, each tool has its own strengths and weaknesses in how the state space of the model is handled.

Another thing to note is that the precise syntax and semantics of PSL is still under development, and the version presented here is based on the most

recent proposal [1]. Thus, if the precise mechanics of PSL changes, the tools need to be updated accordingly. The PSL parser in NuSMV, for example, is not up to date with the language specification, and some PSL properties are not accepted by the parser. Therefore, future work should also include updating the components that directly deal with PSL.

Other future work could also include adding support for temporal operators that refer to the past. This would be a relatively simple addition to the observers described in this work. Another, slightly more complicated, addition to the observers would be the ability to also accept infinite paths as input, enabling the complete model checking of all PSL properties.

## Acknowledgements

This report is a reprint of my Master's Thesis. The funding of the Technology Industries of Finland Centennial Foundation, the Model-based safety evaluation of automation systems (MODSAFE) project, and the Academy of Finland (project 112016) are gratefully acknowledged. I would especially like to thank my supervisor Prof. Keijo Heljanko and D.Sc. (Tech.) Tommi Junttila for their input and instructions for this thesis.

Finally I would like to thank my parents and Laura for their support during my studies and during the writing of this thesis.

# BIBLIOGRAPHY


[1] Formal syntax and semantics of IEEE Std 1850 Property specification language. Retrieved on 13th October, 2008, from `http://www.eda.org/ieee-1850/ieee-1850-issues/hm/att-0690/Final_Annex_B_08.pdf`.

[2] IBM Haifa Research Lab, PSL/Sugar. `http://www.haifa.ibm.com/projects/verification/sugar/`.

[3] IEEE Standard for Property Specification Language (PSL). *IEEE Std 1850-2005*, pages 1–143, 2005. Available online at `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1524461`.

[4] Ilan Beer, Shoham Ben-David, Cindy Eisner, Dana Fisman, Anna Gringauze, and Yoav Rodeh. The Temporal Logic Sugar. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, volume 2102 of *Lecture Notes in Computer Science*, pages 363–367. Springer, 2001.

[5] Shoham Ben-David, Roderick Bloem, Dana Fisman, Andreas Griesmayer, Ingo Pill, and Sitvanit Ruah. Automata Constructon Algorithms Optimized for PSL. Technical Report Deliverable 3.2/4, ProdSyd project, 2005. Available from: `http://www.prosyd.org/`.

[6] Armin Biere, Keijo Heljanko, Tommi A. Junttila, Timo Latvala, and Viktor Schuppan. Linear Encodings of Bounded LTL Model Checking. *Logical Methods in Computer Science*, 2(5), 2006.

[7] Randal E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[8] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002.

[9] Alessandro Cimatti, Marco Roveri, and Stefano Tonetta. Symbolic Compilation of PSL. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1737–1750, 2008.

[10] Koen Claessen and Johan Mårtensson. An Operational Semantics for Weak PSL. In Alan J. Hu and Andrew K. Martin, editors, *Formal Methods in Computer-Aided Design, 5th International Conference, FMCAD 2004, Austin, Texas, USA, November 15-17, 2004, Proceedings*,

volume 3312 of *Lecture Notes in Computer Science*, pages 337–351. Springer, 2004.

[11] Edmund M. Clarke, Orna Grumberg, and Kiyoharu Hamaguchi. Another Look at LTL Model Checking. *Formal Methods in System Design*, 10(1):47–71, 1997.

[12] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 2000.

[13] Cindy Eisner and Dana Fisman. *A Practical Introduction to PSL (Series on Integrated Circuits and Systems)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[14] Cindy Eisner, Dana Fisman, John Havlicek, and Johan Märtensson. The $\top$,$\bot$ approach to truncates semantics. Technical Report 2006.01, Accellera, 2006. Available at `http://www.accellera.org/activities/techrep/Accellera_Technical_Report_2006-01.pdf`.

[15] Klaus Havelund and Grigore Rosu. Efficient monitoring of safety properties. *International Journal on Software Tools for Technology Transfer, STTT*, 6(2):158–173, 2004.

[16] Kenneth McMillan. The SMV language. *Cadence Berkeley Labs*, Jan 1999.

[17] Orna Kupferman and Robby Lampert. On the Construction of Fine Automata for Safety Properties. In Susanne Graf and Wenhui Zhang, editors, *Automated Technology for Verification and Analysis, 4th International Symposium, ATVA 2006, Beijing, China, October 23-26, 2006*, volume 4218 of *Lecture Notes in Computer Science*, pages 110–124. Springer, 2006.

[18] Orna Kupferman and Moshe Y. Vardi. Model Checking of Safety Properties. In Nicolas Halbwachs and Doron Peled, editors, *Computer Aided Verification, 11th International Conference, CAV '99, Trento, Italy, July 6-10, 1999, Proceedings*, volume 1633 of *Lecture Notes in Computer Science*, pages 172–183. Springer, 1999.

[19] Martin Lange. Linear Time Logics Around PSL: Complexity, Expressiveness, and a Little Bit of Succinctness. In Luís Caires and Vasco Thudichum Vasconcelos, editors, *CONCUR 2007 - Concurrency Theory, 18th International Conference, CONCUR 2007, Lisbon, Portugal, September 3-8, 2007, Proceedings*, volume 4703 of *Lecture Notes in Computer Science*, pages 90–104. Springer, 2007.

[20] Timo Latvala. On Model Checking Safety Properties. Research Report A76, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, December 2002.

[21] Timo Latvala. Efficient Model Checking of Safety Properties. In Thomas Ball and Sriram K. Rajamani, editors, *Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9-10, 2003, Proceedings*, volume 2648 of *Lecture Notes in Computer Science*, pages 74–88. Springer, 2003.

[22] Timo Latvala, Armin Biere, Keijo Heljanko, and Tommi A. Junttila. Simple Is Better: Efficient Bounded Model Checking for Past LTL. In Radhia Cousot, editor, *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings*, volume 3385 of *Lecture Notes in Computer Science*, pages 380–395. Springer, 2005.

[23] Marie-Fleur Revel. Translating PSL into the Linear Time $\mu$-Calculus. Master's thesis, Ludwig-Maximilians-UniversitÃd't, MÃijnchen, 2006.

[24] Kenneth McMillan. *Symbolic Model Checking: An approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, 1992.

[25] Max Michel. Computation of Temporal Operators. *Logique et Analyse*, 28(110–111):137–152, 1985.

[26] Amir Pnueli and Aleksandr Zaks. PSL Model Checking and Run-Time Verification Via Testers. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods, Hamilton, Canada, August 21-27, 2006, Proceedings*, volume 4085 of *Lecture Notes in Computer Science*, pages 573–586. Springer, 2006.

[27] Sitvanit Ruah, Dana Fisman, and Shoham Ben-david. Automata construction for on-the-fly model checking PSL safety simple subset. Technical report, PSL Safety Simple Subset, IBM, Tech. Rep. H-0234, 2005.

[28] Klaus Schneider. Improving Automata Generation for Linear Temporal Logic by Considering the Automaton Hierarchy. In Robert Nieuwenhuis and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, LPAR 2001, Havana, Cuba, December 3-7, 2001, Proceedings*, volume 2250 of *Lecture Notes in Computer Science*, pages 39–54. Springer, 2001.

[29] Heikki Tauriainen and Keijo Heljanko. Testing LTL formula translation into Büchi automata. *International Journal on Software Tools for Technology Transfer, STTT*, 4(1):57–70, 2002.

[30] Antti Valmari. The State Explosion Problem. In Wolfgang Reisig and Grzegorz Rozenberg, editors, *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets, held in Dagstuhl, September 1996*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer, 1996.

TKK REPORTS IN INFORMATION AND COMPUTER SCIENCE

| | |
|---|---|
| TKK-ICS-R7 | Kai Puolamäki, Samuel Kaski<br>Bayesian Solutions to the Label Switching Problem. June 2008. |
| TKK-ICS-R8 | Abhishek Tripathi, Arto Klami, Samuel Kaski<br>Using Dependencies to Pair Samples for Multi-View Learning. October 2008. |
| TKK-ICS-R9 | Elia Liitiäinen, Francesco Corona, Amaury Lendasse<br>A Boundary Corrected Expansion of the Moments of Nearest Neighbor Distributions. October 2008. |
| TKK-ICS-R10 | He Zhang, Markus Koskela, Jorma Laaksonen<br>Report on forms of enriched relevance feedback. November 2008. |
| TKK-ICS-R11 | Ville Viitaniemi, Jorma Laaksonen<br>Evaluation of pointer click relevance feedback in PicSOM. November 2008. |
| TKK-ICS-R12 | Markus Koskela, Jorma Laaksonen<br>Specification of information interfaces in PinView. November 2008. |
| TKK-ICS-R13 | Jorma Laaksonen<br>Definition of enriched relevance feedback in PicSOM. November 2008. |
| TKK-ICS-R14 | Jori Dubrovin<br>Checking Bounded Reachability in Asynchronous Systems by Symbolic Event Tracing. April 2009. |
| TKK-ICS-R15 | Eerika Savia, Kai Puolamäki, Samuel Kaski<br>On Two-Way Grouping by One-Way Topic Models. May 2009. |
| TKK-ICS-R16 | Antti E. J. Hyvärinen<br>Approaches to Grid-Based SAT Solving. June 2009. |

ISBN 978-952-248-041-5 (Print)
ISBN 978-952-248-042-2 (Online)
ISSN 1797-5034 (Print)
ISSN 1797-5042 (Online)