Static Analysis of Symbolic Transition Systems with Goose

Sebastiaan la Fleur

Supervisors:
prof.dr. J.C. van de Pol, University of Twente
prof.dr.ir. A. Rensink, University of Twente
J. Luijsterburg MSc., ING
K. van der Vlist MSc., ING

UNIVERSITY OF TWENTE.

Formal Methods and Computer Science
University of Twente
The Netherlands
March 21, 2018

A thesis presented for the degree of:
Master of Science

Contents

Acknowledgements

This thesis is the finale of 7.5 years of study and I have worked on it with with pride and pleasure from July 2017 to March 2018. As with many larger projects, this work has had input from a number of colleagues and friends and with this section I would like to acknowledge and thank each of them.

First, I would like to thank Jaco van de Pol. Jaco is the first supervisor for my thesis and has been intensively involved from the start. Jaco has given me the freedom and trust to find my own path through this project and has guided me whenever I needed help. We have had numerous discussions to improve how best we could explain the different facets of the work and on what existing literature I could or could not use while also maintaining a distance from the project so I always felt independent and capable.

Second, I would like to thank Arend Rensink. Arend is the second supervisor for my thesis. He found the time in a full schedule to critically read my thesis and provide feedback that improved the readability of the work and the notation used considerably. Arend's passion for the subject matched my own and this was the source for fun and lively discussions that inspired enthusiasm for the work.

Third, I would like to thank dr.ir. Sebastiaan J.C. Joosten. Sebastiaan was a colleague at the university. He was a lively sparring partner with whom I designed some methods which are not in this thesis and our discussions were a significant inspiration source for Goose.

Fourth, I would like to thank Joris Luijsterburg, Kevin van der Vlist, Jorryt-Jan Dijkstra, Joost Bosman & Viet Nguyen. Jorryt was my supervisor at ING for the discovery part of the project while Joris and Kevin are my supervisors at ING for the rest of the project. All three helped me find my way through the existing tooling for the Rebel DSL. Joris and Kevin helped me figure out all requirements for the ING use-case and to create the survey of ING specfications. Joost & Viet are the managers at ING which manage the Rebel project and I would like to thank them for facilitating the project.

Finally I would like to thank all my friends and family for their unwavering moral support.

Aan het einde komt alles goed. Is het nog niet goed, dan is het nog niet aan het einde.

Chapter 1

Introduction

The world as we know today is built on the back of generations of mathematicians, logicians and computer scientists. Computers and the programs which run on them have become a necessity in all parts of the modern world. It was not too long ago when in 1961 rooms full of mathematicians were laid off at NASA and replaced by an electronic computer. Now, we have more computing power in our pockets in the form of cellphones and we use it to communicate wirelessly with friends, family and co-workers. Even governments assume each citizen owns some sort of computer with internet access: The Dutch tax authority is in the process of a multi-year project to switch from filing taxes using paper to filing our taxes on-line. The infrastructure to enable all this includes kilometres of optical or copper wiring, processors with components smaller than a human hair is thin and screens with enough pixels so humans are unable to see the different pixels with the naked eye. All this infrastructure is instrumented by programs or software, written by programmers or software engineers. The instructions they created are not as simple as one may think. Some is based on mathematically proven logic. The logic we use today to create algorithms and to prove that they work correctly has been studied and worked on by logicians and mathematicians over multiple centuries. The combination of the disciplines of logic and math with mechanical and electronic computers have given rise to the discipline of computer science.

In order to automate all the different processes and tasks with software of our daily life, programmers had to create a lot of software. Marc Andreessen stated in 2011 that "Software is eating the world" (Wall Street Journal). All the software worldwide is difficult to maintain: new features, deprecation of programming languages in favour of others and new execution platforms create the need to constantly change and update software. The software is also complex: we use software to organize and track all satellites, asteroids and space debris near Earth, simulate the drag on a virtual air-plane design and to track, prevent and solve traffic jams. Due to the scale and complexity many issues or bugs in existing software exist and replacing or updating the existing systems is again a difficult task. The discipline of software engineering within computer science is focused on improving the way software is created to manage the maintenance and complexity concerns.

Due to the sheer amount and complexity of software, new ways to manage code have to be created to be able to switch execution platforms efficiently and to prevent bugs. Large companies are driven to search for alternative ways of managing their codebases [20] [23] [24]. ING are a prime example who are researching if model-driven engineering is part of their solution. Their plan is to create a Domain-Specific Language (DSL) called Rebel [24] to specify their business logic and to generate the entire code base from these DSL specifications. A DSL is a programming language specialized for a specific domain such as Rebel is specialized to specify banking products. ING is researching if a DSL may be used to consistently and unambiguously define specifications, show the specifications graphically to help with interpretation and in the process also save on time when developing the specifications. Also, new programming bugs should not be introduced when using a verified code generator. This method improves the reliability of the code.

Business logic/semantic issues still remain and one method of finding semantic bugs is to (dis)prove properties such as invariants (a property which must be always true). Our task is to find useful properties to verify and techniques to prove these properties while the specifications are being written. It is therefore important that the technique used and the properties verified are able to run on an average workstation within reasonable time. Finding and/or proving properties such as safety (a certain state is never reached) and liveness (useful progress is always made) properties increases the reliability of generated code even further.

The DSL ING has developed and is currently researching is called Rebel. Rebel is a DSL to specify banking products. Each banking product may be specified using specifications which are based on the formalism of Symbolic Transition Systems (STS). These variations of transition systems describe the states and executions of a program with possibly an infinite state space. An example of a STS may be seen in figure 1.1. There exist two nodes with the labels *Open* and *Frozen*. The node *Open* has a start transition. The start transition has a single constraint where each assignment which satisfies the constraint is a starting state. "After" the start transition, some state holds and is in the *Open* node. Between these nodes are the transitions *Withdrawal*, *Deposit*, *Freeze account* and *Unfreeze account*. Each transition has a guard constraint (which is above the line) and a relation constraint (which is below the line). A transition may be taken when the current state satisfies the guard. When a transition is taken, we move from the original state to some new state in the new node so the original and new state together satisfy the relation constraint. We say that each origin state and destination state which satisfy the guard and relation constraints are related. With the relation constraint we are referencing two states: the origin and destination state. Any variable in the relation constraint which is postfixed with the $'$ symbol is referencing a variable for the destination state while any variable without the $'$ symbol is referencing a variable for the origin state.

Within the STS there may exist (in)finitely many paths from a starting transition and starting node to other nodes by taking transitions. When some state is reached for some node, we say that the state is reachable in that node. A node is therefore an abstraction for a number of reachable states. With our approach, we shall use properties called reachable state constraints where each reachable state must satisfy the reachable state constraint.

With Rebel, ING is interpreting the transitions of a STS as actions as the target for code generation. In our example of figure 1.1 the *Freeze account*

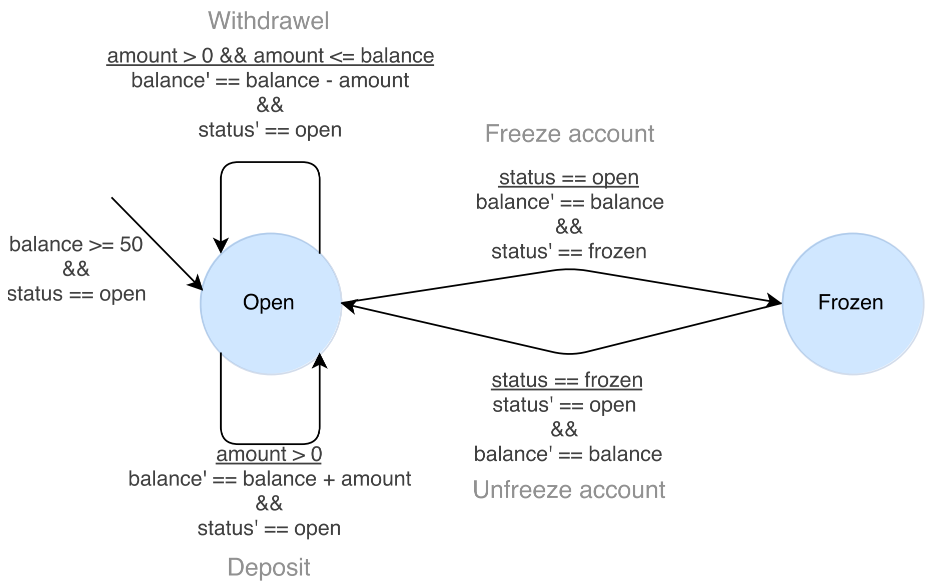


Figure 1.1: STS for a bank account with withdrawal, deposit, freeze and unfreeze account transitions.
Guards are above the line, relations below the line for transitions

action changes the status of the account to `frozen`. Each action is generated as an API call and the guard and relation constraints then become pre- and post-conditions which must be satisfied before and after each action.

The goal of this research is two-fold: **1)** To define useful properties to verify for a STS and 2) To detail our approach of verifying these useful properties for a STS. The motivation to research these two goals is to increase the reliability of the Rebel specifications or any other language which is based on the symbolic transition system formalism. We shall show how we may prove that: 1) a property holds for all states which are reachable (safety property) 2) there exist no transitions which may never be taken (dead transitions) 3) there exist no states so no transitions may be taken (sinkholes) 4) there exist no start transitions which may never be used (unsatisfiable start transitions) and 5) there exist no reachable states that a transition may be taken (the guard is satisfied) but can never finish (there exists no model for the relation) (unsatisfiable relation). The proving technique we shall show uses properties which symbolically describe all reachable states for a node. We shall also show a method how to deduce these reachable state constraints (RSCs) from existing RSCs or from starting transitions. We shall also show that this method is not complete for STSs which contain cycles in which case a user may supply a RSC for a node in the specification. The two-step algorithm to first deduce all RSCs for some STS (if possible) and then try to prove the five verification properties is nicknamed Goose. We are unable to elaborate on the motivation for this name.

Prior to designing Goose we researched existing methods for verifying useful properties for STSs. We found three tools: MCMT, Z3 and nuXmv. MCMT is a tool designed to verify safety properties for STS-like systems where arrays are also allowed as a type for variables. Unfortunately, a general translation from an arbitrary STS to the input language of MCMT is not possible. Z3 and nuXmv are tools designed to verify safety properties for arbitrary infinite-state systems. STSs are included in this classification of systems. Z3 and nuXmv each implement a variation of the IC3 algorithm which we will detail in chapter 3.2.1. We shall see that Z3 and nuXmv treat the STS to verify as a blackbox and they try to recreate the graph structure with nodes and transitions using symbolic properties similar to reachable state constraints. Z3 and nuXmv call these properties frames. Because Z3 and nuXmv have to recreate the structure, they are unable to leverage the knowledge of cycles in the system. When they try to verify certain safety properties, they are forced to unwind the cycle in sequent steps until they reach a counterexample or a fix-point. Unwinding the cycle may increase the resources needed to (dis)prove a safety property considerably as we will see in chapters 3.2.7 and 5.5.

Based on the goals and the research we shall answer a number of research questions with this document. We shall elaborate on the research questions in chapter 1.1. This introduction is concluded with an overview of the structure of this document in chapter 1.2.

1.1 Research Questions

The goal of this document is two-fold: **1)** To define useful properties to verify for a STS and 2) To detail our approach of verifying these useful properties for a STS. We have split these goals into a number of research questions. The list of research questions below also contain a reference to a chapter. The referenced chapter answers that specific research question.

1. What useful properties may be verified for a STS? (chapter 4.3)
2. What verification techniques & tools exist which may help to verify properties about a STS? (chapter 3)
3. What approach may be taken to decrease the resources needed and to increase the solvability to verify our chosen useful properties for a STS over existing techniques? (chapter 4)
4. Is this approach sound, complete, does it always terminate and what are the practical limitations? (chapter 4)
5. Using a suite of Rebel specifications commonly used by ING, how do the selected tools perform in terms of solvability and execution time to verify on an average workstation and is the execution time within the practical limit of a few seconds? (chapter 5)

In chapter 3 we shall introduce the tools MCMT, nuXmv and Z3. As we will explain, these are the existing tools that are able to only verify safety properties for some STS. The practical application of these tools is limited when the STS in question contains cycles. This was the motivation to create our own approach which can verify several properties including safety properties and is practically usable when the STS contains those cycles where the existing work is found unusable. Therefore our list of research questions also contains questions relating to our own approach.

Contributions

The initial goal of the research was to define useful verification properties and to find and use an existing technique to verify the useful properties for arbitrary STSs. We quickly found that the existing techniques have a long execution time, as we will describe, when considering STSs containing cycles and certain safety properties. This was the motivation to start designing our own approach which uses user-given or generated properties which summarize all reachable states per node, allowing to summarize all cycle iterations into a single property. A prototype was developed and the ambition arose to evaluate the new approach with existing approaches in terms of execution time and if it is able to solve the query correctly (solvability). We also discovered that the new approach is able to prove more verification properties than just safety properties. In a later stadium we discovered that the tool Z3, which we use as a SMT solver in our approach, is also able to verify safety properties for arbitrary STSs and may even prove the other verification properties which we have defined. Due to time constraints, this is pushed to future work. All in all the deliverables of this research include this document with an overview of existing techniques, the theoretical framework for the new approach and an evaluation of the new approach with the existing approaches in terms of execution time and solvability and a prototype implementing the new approach to be used for evaluation and to show the feasibility of the new approach.

Methodology

In order to answer the research questions of chapter 1.1, we have done the following tasks:

1. Literature research for existing techniques to verify one or more properties for arbitrary STSs.
2. Based on existing literature and discussions with ING define a number of verification properties deemed useful to ING.
3. Evaluate existing techniques on what useful verification properties they are able to verify and if they are applicable for the ING use-case.
4. Created the theoretical framework for a new approach which solved the issues existing techniques had for the ING use-case.
5. Built a prototype implementing the new approach
6. Evaluated the prototype on what useful verification properties it is able to verify and if it is applicable for the ING use-case.

These tasks were performed with the main goal of finding a technique which is able to verify as many of the useful verification properties for the survey of ING specifications as described in appendix A in reasonable time. We have defined reasonable time as a few seconds on an average workstation. The workstation used in the experiments of chapter 5 is deemed average by ING and the researchers.

1.2 Document Structure

Our document is structured as follows: We shall first introduce the concepts of first-order logic, constraints, SMT solvers, STS and strongly connected components in chapter 2. We shall show how existing techniques relate to the verification of STSs in chapter 3. Then we shall give our approach in chapter 4 and how it compares to existing techniques in chapter 5. The document concludes with the areas of future work in chapter 6 and some concluding remarks in chapter 7.

Chapter 2

Pre-requisite Knowledge

In this chapter we shall define and cite the knowledge needed to understand our approach in chapter 4. It begins with a description of first-order logic in chapter 2.1. Then, we shall use first-order logic to define constraints in chapter 2.2. The understanding of constraints and first-order logic is necessary for our overview of SMT solvers in chapter 2.3 and our definition of symbolic transition system (STS) in chapter 2.4. We will end the chapter with an overview of strongly connected components in chapter 2.5.

2.1 First-order Logic

First-order logic is a logic allowing to describe statements about a set of entities e.g. all chairs are made of wood. It is a more expressive logic compared to proposition logic allowing to not only reason about true or false for specific facts, but also reason if facts are true for all or some entities within a domain D. For our research we shall use the syntax as described in Ben-Ari [8]. A first-order logic formula may be constructed using a number of syntactical elements. Examples include the ones given in table 2.1.

Together, these syntactical elements allow to describe statements such as:

$$\forall pot.\exists lid.isPot(pot) \rightarrow isLid(lid) \wedge lidForPot(lid, pot)$$
Every pot has a lid

$$\exists car.hasColour(car, col)$$
There exists a car with colour col

It is important to note that first-order logic formulas are without predefined interpretation. This means that while the labels of relations, functions, variables and constants may hint to some semantic meaning, strictly, this first has to be defined. For instance, the first example statement might seem to only work for pots and lids. Any first-order logic statement might be interpreted for any domain D which may consist of any entities. The property *isPot* of predicate *isPot*(*pot*) might also be interpreted to have the meaning *isHuman*, *isLid* as *isEdible*, *lidForPot* as *canEat* and we might take the domain of all living creatures. Now, the statement is interpreted as *All humans can eat an edible, living*

Table 2.1: Examples of syntactical elements for first-order logic formulas

Syntactical Element	**Example**	**Definition**
Function	$f(a_1, a_2..., a_n)$	A function f over parameters a_1 to a_n. Each parameter a_i may be a constant, variable or another function and evaluates to a member of D. The function maps every n-tuple of a_1 to a_n to some member in domain D.
Relation	$p(a_1, a_2..., a_n)$	A relation p over parameters a_1 to a_n. Each parameter a_i may be a constant, variable or another function and evaluates to a member of D. The function maps every n-tuple of a_1 to a_n to some member in domain D.
Quantifier	$\forall x.p(x)$ and $\exists x.p(x)$	Define a statement $p(x)$ for all members or some members of D.
Variable	x	A free variable to be assigned some member of D or quantified over D by the existential or universal closure.
Constant	a	A constant which must be assigned some member of D.
Boolean operator	$\wedge, \vee, \rightarrow, \neg...$	Binary and unary operators from propositional logic.

creature. Therefore, when describing first-order logic formulas, it is important to also note the domain and the meaning of the relations and functions.

The first example formula contains only bounded variables; variables *pot* and *lid* are both bounded by a quantifier. The second example formula also contains the constant or free variable *col* as it is not bounded by a quantifier. This symbol has to be assigned a member of D in order to evaluate the formula to true or false. If we say that *col* is a constant, we must assign it a member of D with the interpretation. If we say that *col* is a free variable, we must assign it a member of D with a variable assignment. The interpretation and variable assignment for a formula are different. An interpreted formula may need a variable assignment before the formula may be evaluated to true if it contains free variables. When a formula is interpreted, the constants are assigned some member of D and the remaining symbols are considered the free variables.

Evaluating a first-order logic formula

As we have shown, we have to describe the context of a formula or, in other words, interpret the formula. This interpretation consists of a domain D, a definition for all relations and functions and an assignment for all constants.

Definition 1. *An interpretation I for a first-order logic formula F consists of a domain D, a definition of all relations and functions and an assignment for all constants.*

In order to evaluate the truth value of some formula, we also need to assign all free variables some member from D. $\sigma[x_0 \leftarrow d_0, ..., x_n \leftarrow d_n]$ will be used as the syntax for variable assignment σ where values $d_0..d_n$ are assigned to the respective variables $x_0..x_n$.

Definition 2. *With an interpretation I and an assignment to free variables $\sigma[x_i \leftarrow d_i]$ a first-order logic formula may be evaluated to true or false.*

Finally, when an interpretation I and some assignment to free variables σ evaluates some formula A to true, we consider that a model of A and we write $I, \sigma \models A$. When considering multiple formulae $U = \{A_1, ...A_n\}$, a model is an interpretation and assignment so that for each of the formulas $I, \sigma \models A_i$.

Definition 3. *Considering a set of formulae $U = \{A_0, ..., A_n\}$, $I, \sigma \models U$ iff $\forall A_i \in U. I, \sigma \models A_i$*

Satisfiability & Validity

When considering some (set of) formula A, the formula may have none or some models. In some cases, the formula is always true. We call A satisfiable if there exists a model for A, falsifiable if there exists a model for the negation of A, valid if there are no models for the negation of A and unsatisfiable if there are no models for A. Therefore, if a formula is not falsifiable it has to be valid and if a formula is not satisfiable it is unsatisfiable.

Logical Consequence

Considering some set of first-order logic formulae U, there might exist other formulae A that describe at least the same models as U. For instance, $A = \forall a.P(a) \land R(a)$ is a logical consequence of $U = \{\forall a.P(a), \forall a.R(a)\}$ as for all models of U, the same models evaluate to true for A.

Definition 4. *Given a set of formulae U and some formula A, A is a logical consequence of U (written as $U \models A$) iff every model of U is a model A.*

Theories

With first-order logic formulas we are able to specify properties for a set of entities. For instance, we may specify the commutativity of addition as the formula $\forall a.\forall b.\mathit{equal}(+(a, b), +(b, a))$ with the function $+$ as the addition operator and *equal* as the equality relation.

We may use a set of formulae to partition a specific set of models. If then for every formula, which also has at least these models, it is also in the set then the set is called a theory. In other words, the set is closed under logical consequence for all possible formulae.

Definition 5. *A set of formulae U is a theory iff for any formula A if $U \models A$ then $A \in U$.*

For a more detailed description of first-order logic, we would like to refer to Ben-Ari [8].

2.2 Constraints

A constraint is a first-order logic formula which is always interpreted within the mathematical and logical domain to true or false. We shall consider the domain with the members $\mathbb{B} \cup \mathbb{Z} \cup \mathbb{R} \cup \mathbb{S} \cup$ *user-defined* or in words: the boolean, integer number, real number, string and user-defined domains. The user-defined domain is a special domain which includes types with values defined by a user as an enumeration. We consider a subset of mathematical, relational, propositional and string operators. Later in this section we detail which operators we specifically consider.

Operators are functions and are defined for their respective subset of the domains. For instance, addition $+$ is defined as a binary operator for two numbers to a single number and disjunction $\vee$ is defined for two boolean values to a boolean value. Constants and (free) variables may have a type. A type in this context means that the variable or constant has a value which belongs to one of the domains mentioned in the previous paragraph.

Constraints allow us to write properties in interpreted, widely known domains. We may write the constraint $a == 5$ given that a is declared as an integer variable. The constraint has the model $\sigma[5 \leftarrow a]$. This is an example of a constraint with a single model. The constraint $a > 100$ may have infinitely many models if a is an unbounded integer, while the constraint $a < 0 \wedge a > 0$ is unsatisfiable as there is no number which is both smaller and larger than 0.

Definition 6. *A constraint is a first-order logic formula where the interpretation has domain D as $\mathbb{B} \cup \mathbb{Z} \cup \mathbb{R} \cup \mathbb{S} \cup$ user-defined, relations and functions are defined as defined within the respective domains($+$ is addition, $-$ is minus, $==$ is equality,...) and constants are always assigned. A constraint is always accompanied with the declaration of variables and their types $Decl_v$ and the declaration of enumeration types $Decl_{enum}$*

We also define the $var(C)$ function.

Definition 7. *Given a set of constraints C, the function $var(C)$ returns the set of all variables used in any $c \in C$.*

Language

With this chapter we will define the constraint language we will consider throughout this document; including the definition of a STS. While different or broader definitions may exist, we limit ourselves to the constructs and operators of this chapter. The constraints we shall consider may be defined using the syntax elements in tables 2.2, 2.4, 2.5 and 2.6. We consider values, variables, operators and types. All variables must be free as we do not consider quantifiers in our language. The types we consider are integer, real, boolean, string and enumeration. Values have the type depending from which domain the constant symbol is used. Variables must be declared with a type.

Table 2.2: Values/constants syntax elements for defining constraints

Value	**Semantics**	**Type**	**Syntax example**
Integer number	Constant number without decimals	Integer	4
Real number	Constant number with decimals	Real	4.5
Boolean value	Constant true or false value	Boolean	*true*
String value	Constant sequence of characters	String	*"abc192"*
Enumeration	Constant from collection of user-defined finite, constant values	User-defined	Given $enum\,Answer = Yes\|No$, example value: Yes

Table 2.2 shows which values are possible and how variables are defined. The type of the variable is given when a variable is defined and after the definition the variable may be used within any constraints. Enumerations are a special kind of type. They are a user-defined type with a set of values. These values are user-given, alphanumerical values with a user-given, alphanumerical type. In our example table, the enumeration type *Answer* is defined with two values: *Yes* and *No*. The definitions of enumeration types must be given separately for the constraints where they are used.

Table 2.3: Variable syntax elements for defining constraints

Value	**Semantics**	**Type**	**Syntax example**
Variable	Identifier with a type denoting a possible but unknown value	Integer, Real, Boolean, String or user-given	Given *var balance* : *Real*, example variable: *balance*

Table 2.3 shows how variables may be declared and used. With the example, we define the variable *balance* with the *Real* type outside of any constraint so we may use the *balance* variable with any constraint.

Table 2.4: Mathematical operators for defining constraints

Operator	**Semantics**	**Unary/Binary**	**Syntax example**
$+$	Add operator	Binary operator	$4 + y$
$-$	Subtraction operator	Binary operator	$4 - y$
$*$	Multiply operator	Binary operator	$6 * y$
$/$	Division operator	Binary operator	$6/y$
ˆ	Exponent operator	Binary operator	$6\hat{}y$
%	Modulo operator	Binary operator	6%2
$-$	Negation operator	Unary operator	$-y$

The mathematical operators which we consider are defined in table 2.4. All binary mathematical operators, except modulo, are defined for all mathematical types. The mathematical types are the integer and real type. Negation is defined for all mathematical types and modulo is defined for the integer type.

Table 2.5: Propositional operators for defining constraints

Operator	**Semantics**	**Unary/Binary**	**Syntax example**
&&	And operator	Binary operator	*true&&y*
||	Or operator	Binary operator	*true||y*
!	Not operator	Unary operator	!*y*
$\rightarrow$	Implication operator	Binary operator	$true \rightarrow false$

The propositional operators which we consider are defined in table 2.5. All propositional operators expect their arguments to be of boolean type and the resulting expression type is boolean. Within this document we shall commonly write the conjunction operator && also as $\wedge$ and the disjunction operator || also as $\vee$.

Table 2.6: Relational operators for defining constraints

Operator	**Semantics**	**Unary/Binary**	**Syntax example**
$\geq$	Greater than or equals operator	Binary operator	$4 >= y$
$>$	Greater than operator	Binary operator	$4 > y$
$\leq$	Lesser than or equals operator	Binary operator	$6 <= y$
$<$	Lesser than operator	Binary operator	$6 < y$
$==$	Equals operator	Binary operator	$4 == y$
$! =$	Not equals operator	Binary operator	$5! = y$

The relational operators which we consider are defined in table 2.6. The arguments of the operators must have the same types and the resulting type is boolean. Also, the 'greater than', 'greater than or equals', 'lesser' and 'lesser than or equals' operators are considered the mathematical relational operators. Their arguments may only be one of the mathematical types. Within this document we shall commonly write the $\geq$ operator as $>=$, and $\leq$ operator as $<=$.

Table 2.7: String operators for defining constraints

Operator	**Semantics**	**Type**	**Syntax example**
++	Concatenation operator	Binary operator	$"ab" + +y$

The string operators which we consider are defined in table 2.7. The arguments must be of type string and the resulting type is also of type string.

Operator precedence

When a constraint such as $4 + 5 * 8.5 == 10$ is to be evaluated, there may be multiple orders in which the operators are evaluated. It might be $((4 + 5) * 8.5) == 10$ or $(4 + (5 * 8.5)) == 10$. By setting the operator precedence, we remove this ambiguity. Our operator precedence is based on the operator precedence from the C programming language [2]. Operator precedence we shall consider from highest priority to least:

1. $-(negation)$

2. ˆ
3. $*, /, \%$
4. $+, -(subtraction), ++$
5. !
6. $\geq, >, \leq, <$
7. $==, !=$
8. $\wedge$
9. $\vee$
10. $\rightarrow$

2.3 SMT Solver

Satisfiability Module Theories solvers, or SMT solvers, are a class of solvers dedicated to finding models for first-order logic formulas and also constraints as defined in chapter 2.2. The semantics of the pre-defined domains are given as a theory. Each theory is the basis for a decision procedure to find models within the interpretation domain of the theory such as real numericals and booleans. The theory defines the semantics we commonly use with these domains such as the addition operator + which is defined as the sum of two numbers. In other words, SMT solvers solve first-order logic formulas interpreted as existentially-closed constraints. The goal of SMT solvers is to find an assignment which evaluates the formula to true or prove that no such assignment exists.

When we consider constraints for which we use a SMT solver to find a model, the interpretation is set to the pre-defined domains. Therefore the SMT solver tries to only find an assignment. We shall use the terms model and assignment in this context interchangeably.

SMT solvers are a broadening of Satisfiability solvers (SAT solvers). SAT solvers find models for arbitrary propositional formulas and therefore only use the semantics for the boolean domain.

2.3.1 Z3

An example of a SMT solver is Z3 [14]. Z3 implements both a Z3-only DSL and the SMT-LIB standard [7] input languages. The SMT-LIB is a standard for an input language for SMT solvers; the motivation to design such a standard was to create standardized benchmarks. The SMT-LIB input language is different from our constraint language. For example, SMT-LIB uses the Polish notation and equality in SMT-LIB is checked with the = symbol .

Listing 2.1: Example of Z3 query in SMT-LIB version 2.

```
(declare-var x (Int))
(declare-var y (Int))

(assert (= (+ x y) 5))
(assert (> x 2))

(check-sat)
(get-model)
```

A formula to solve for Z3 is given in listing 2.1. It specifies two variables x and y for which it tries to find a model so that all assertions are true. A possible model Z3 might give for this query is the assignment $\sigma[3 \leftarrow x, 2 \leftarrow y]$. The last two commands `check-sat` and `get-model` respectively check the satisfiability of the formula and return a model for the formula.

2.3.2 Solvability

SMT Solvers are programs implementing an attempt at an incomplete, non-terminating decision procedure for the satisfiability of first-order logic formulae extended with constraints. These programs will not terminate or return an answer for some formulae. In the last case they will return *unknown.* When the program is able to return an answer, it will state that the formulae have a model (with a free variable assignment if applicable) or that some formulae has no model. SMT solvers return the *not satisfiable* answer for the latter as it is able to construct a proof that no model exists.

In general it is impossible to give a decision procedure for the satisfiability of first-order logic formulae. Unbounded non-linear integer arithmetic is a class of formulae for which finding a model in general is undecidable [13]. Also proving the abscence of models for some formulae using certain quantifier patterns is undecidable [8].

2.3.3 Proving validity with a SMT Solver

We have noted that a SMT solver is able to give three answers:

1. Satisfiable formula with a certain model
2. Unsatisfiable formula
3. Unknown

We can leverage a SMT solver to prove the validity of certain formulas. As we have stated in chapter 2.1, validity is the absence of models for the negation of the formula; in other words, the unsatisfiability of the negation of the formula [8].

Definition 8. *Given some first-order logic formula P, P is valid if $\neg P$ is unsatisfiable.*

As an example, we shall prove that all numbers which are not zero, are either smaller or greater than zero. Or, as a universally closed constraint: $A = [\forall i \in \mathbb{R}. i!{=}0 \rightarrow i < 0 \vee i > 0]$. To show the validity of this formula, we have to show the absence of falsifying models. A falsifying model is any model so that the formula is false; in other words, that the negation of the formula is true. So we ask Z3 to find a model for the formula: $\neg A$. If Z3 deems the formula unsatisfiable, we know that our formula is valid and if Z3 deems it satisfiable then the model is a counterexample to show for which model the formula is false. In this case A is valid as Z3 states that the negation of the formula is unsatisfiable.

2.4 Symbolic Transition System (STS)

Transition systems (also known as state machines or automata) are one of the fundamental concepts of formal methods for computer science and as such many variants exist. Examples of where transition systems are used are ioco analysis (testing) [26], parsing and grammers (compiler construction) [25] and process algebra [22]. Our research will look specifically at symbolic transition systems (STSs).

Description of STS

Unlike other transition systems, a STS considers a number of variables which we shall call the state variables. These state variables together make up all possible states. A state in this situation is a value assignment to all state variables.

Like all transition systems, STSs consists of nodes, directed transitions between nodes and start transitions for individual nodes. Each node corresponds to a set of possible/reachable states and each transition defines a relation between states of the origin node to states of the destination node as a relation. Each transition is guarded by a constraint which we call the guard. Only for those states in the origin node for which the guard is true, the transition expresses a relation to states in the destination node. In other words, the guard restricts the reachable states of the origin node to that set of reachable states for which the transition exists.

The transition describes the relation between states of the origin and destination node through both the guard and the transition relation. The transition relation is a constraint which describes the relation between states of the origin node and the destination node. The guard and transition relation together are a relation between reachable states in the origin and destination node. Any reachable state in the origin and destination are related if together they are a model of both the guard and the transition relation.

Start transitions consist of a single constraint using the state variables to describe a set of states which is reachable for some node.

We restrict the allowed syntax for constraints to the syntax introduced in section 2.2.

Example STS

As an example, we shall elaborate with figure 2.1. The state variables are $\{i\}$ where variable i is of type integer. There is a starting transition entering the node *i is even* with constraint $i == 0$. While in the state *i is even*, there are 2 possible transitions: increment i with 2 and stay in *i is even* and increment i with 1 and go to *i is uneven or even.* The $'$ symbol refers to the variable i after the transition, while the variable i without the $'$ operator refers to the variable before the transition. The definition of the $'$ operator is given later in this section. Both transitions have guards which are valid. The transition from *i is uneven or even* to *i is even* shows how guards are denoted. For this transition i has to be even ($i\%2 == 0$) and as the relation the assignment to i in the state stays the same.

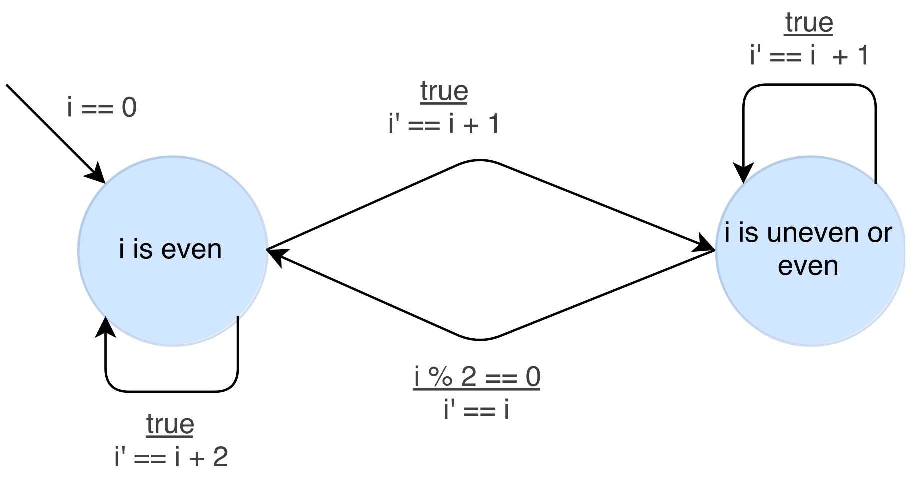


Figure 2.1: STS to keep track if number i is even or not.
Guards are above the line, relations below the line for transitions

Definition of STS

While we define the $'$ symbol later in this section, we do already need the function var_{next} before we are able to give our definition of a STS. Briefly, var_{next} gives all variables followed by the $'$ symbol.

Definition 9. *Given a set of constraints $c \in C$, the function $var_{next}(C)$ returns the set of all variables used in C which are followed by a $'$ symbol.*

With everything so far we give our definition for a STS.

Definition 10. *A symbolic transition system or STS is defined by $(N, (V, T_V), T_{start}, T, N_{final})$:*

- *A set of nodes N.*
- *A set of variables V also known as the state variables with a function T_V which maps each variable $v \in V$ to a type.*
- *A set of starting transitions T_{start} where each starting transition $t_{start} \in T_{start}$ is defined as $(d_{t_{start}}, C_{t_{start}})$ with a destination node $d \in N$ and constraint $C_{t_{start}}$ for which $var(C_{t_{start}}) \subseteq V$.*
- *A set of directed transitions T where a transition $t \in T$ is defined as $(o_t, d_t, C_t^{guard}, C_t^{relation}, (V_t, T_{V_t}))$ where:*
  - *$o_t \in N$ is the originating node*
  - *$d_t \in N$ is the destination node*
  - *C_t^{guard} is the guard constraint for which $var(C_t^{guard}) \subseteq V \cup V_t$*
  - *$C_t^{relation}$ is the relation constraint for which $var_{next}(C_t^{relation}) = V \land var(C_t^{relation}) \subseteq V \cup V_t$*
  - *V_t is the set of transition variables for which $V_t \cap V = \emptyset$ with a function T_{V_t} which maps each variable $v_t \in V_t$ to a type.*
- *A set of accepting nodes N_{final} so $N_{final} \subseteq N$.*

′ symbol

When the transition relation constraint for a transition is defined, we would like to refer to both the state variables of the states of the origin node and the destination node. The constraints of the guard and transition relation may reference the state variables of the state of the origin node while the transition relation may also reference variables of the state in the destination node.

To overcome this scoping issue, we introduce the $'$ symbol. Any variables followed by $'$, reference the state variables of the state in the destination node of the transition. Any variables not proceeded by $'$, reference values of the states variables of the state in the origin node of the destination.

The constraints of the start transitions may only reference the state variables of the state of the destination node, so start transitions do not have this scoping issue. Therefore, we do not introduce the $'$ symbol for constraints of the start transitions.

Path

Paths may be taken through the STS by first choosing a start transition t_{start} and then choosing a number of transitions. After the start transition, any transition t may be chosen so that $o_t = d_{t_{start}}$. In words, any transition which originates in the destination of the start transition may be taken. Analogously, after any transition t_1, any transition t_2 may be chosen so that $o_{t_1} = d_{t_2}$.

Definition 11. *A path Π through some STS is formed by the finite sequence $t_{start}, t_1, t_2..., t_n$ where $n \geq 0$, $t_{start} \in T_{start}$ and $t_i \in T$. If $n \geq 1$ $d_{t_{start}} = o_{t_1}$ should hold for the pair (t_{start}, t_1) with destination node $d_{t_{start}}$ and origin node o_{t_1}. For any pair (t_i, t_{i+1}) $d_{t_i} = o_{t_{i+1}}$ should hold. We also define:*

- *d_s is the starting node of Π.*
- *d_{t_n} is the last node of Π.*
- *t_{start} is the starting transition of Π.*
- *t_n is the last transition of Π.*
- *A path Π to node $n \in N$ is a path where the last node is n.*
- *A node n is contained in Π if for any t_i either $o_{t_i} = n$ or $d_{t_i} = n$.*

Within this document we shall express paths from one node n_{start} to another node $n_{destination}$. These paths do not contain a start transition but start with a transition originating from n_{start}. The last transition in the path should have $n_{destination}$ as destination node.

Definition 12. *A path $\Pi^{n_{start}}_{n_{destination}}$ through some STS from n_{start} to $n_{destination}$ is formed by the finite sequence $t_1, t_2..., t_n$ where $n \geq 1$, $t_i \in T$, $o_{t_1} = n_{start}$, $d_{t_n} = n_{destination}$ and for any pair (t_i, t_{i+1}) $d_{t_i} = o_{t_{i+1}}$ should hold. We also define:*

- *n_{start} is the starting node of $\Pi^{n_{start}}_{n_{destination}}$*
- *$n_{destination}$ is the last node of $\Pi^{n_{start}}_{n_{destination}}$*
- *t_1 is the first transition of $\Pi^{n_{start}}_{n_{destination}}$*
- *t_n is the last transition of $\Pi^{n_{start}}_{n_{destination}}$*
- *A node n is contained in $\Pi^{n_{start}}_{n_{destination}}$ if for any t_i either $o_{t_i} = n$ or $d_{t_i} = n$*

The empty sequence of transitions from a node n is also considered a path where $n = n_{start}$, $n = n_{destination}$ and there are no first and last transitions.

State

We have noted that a state is an assignment for the state variables for some node in the STS. Therefore, a state is an assignment for the state variables and contains a node $n \in N$. We shall use the notation ς for a state while we keep the notation σ for some assignment for a first-order logic formula.

Definition 13. *A state is an assignment ς_n for all variables $v \in V$ for some node $n \in N$.*

States may become (part of) a model for start transitions or transitions. A state may be a model for the constraint of a start transition. States in origin and destination node of some transition may together be a model of the guard and transition relation of some transition between the origin and destination node. A state is then reachable in some node n if there is a path to n and we can find states in each node so they are models for the transitions in between. We note again that the state ς_o for the origin node assigns values to the variables not followed by the $'$ symbol while the state ς_d for the destination node assigns values to the variables followed by the $'$ symbol. This is formalised in definition 14.

Definition 14. *Given some STS with start transitions T_{start}, we say a state ς_d is reachable in node d when considering some path Π to node d if either:*

- *there is a start transition $t_{start} \in T_{start}$ where $\varsigma_d \models C_{t_{start}}$*
- *there exists a ς_o so $\varsigma_o, \varsigma_d \models \exists v_t \in V_{t_{last}}.[C^{guard}_{t_{last}} \wedge C^{relation}_{t_{last}}]$ for the last transition t_{last} of Π where $o = o_{t_{last}}$ and $d = d_{t_{last}}$ and ς_o is reachable in o.*

Transition variables

With our definition of a STS we note that the guard and transition relation may contain variables which are in the set of transition variables. These variables are used to show relations between state variables using variables which do not need to be remembered across nodes. As an example, the variable *amount* in figure 1.1 is a transition variable as the state variables only contain the variables *balance* and *status*.

2.5 Strongly Connected Components

Within graph theory strongly connected components [17] is a fundamental concept. A strongly connected component (SCC) within a graph is a set of nodes where each pair of nodes in the SCC are mutually reachable.

Definition 15. *Given some STS with nodes N and transitions T, a strongly connected component $SCC \subseteq N$ is defined as for all $n \in SCC$ there exists a path $\Pi^n_{n'}$ from n to all nodes $n' \in SCC$.*

The theoretical value of SCCs is to find the partitions where all nodes which are mutually reachable are in the same partition. Figure 2.2 shows the SCCs for some graph to show this characteristic visually. Each of the light blue areas are a partition of nodes. The light blue arrows show which partition is reachable from another.

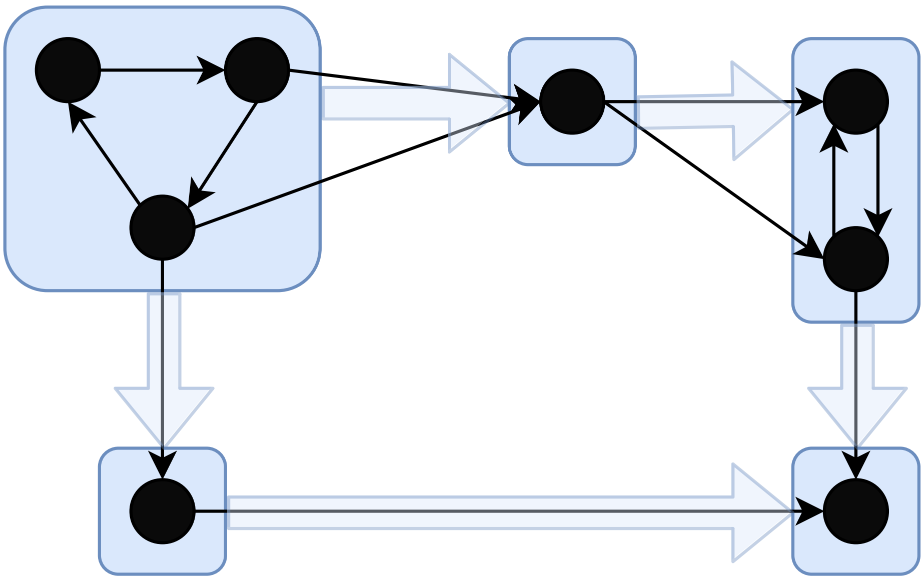


Figure 2.2: Overview of strongly connected components for some graph. Shows the transitions between SCCs.

To find all SCCs for some graph structure, Gabow [17] has created his path-based strong component algorithm. While we shall use this algorithm with our approach, we shall not elaborate the procedure within this thesis.

When contracting all nodes of a strongly connected component to a single node, the result is again a graph. We shall not define a graph formally. It consists of nodes and transitions between the nodes. There are no guards, transition relations or starting transitions. The resulting graph is acyclic and is called the condensation (or quotient graph in some works) of the strongly connected components [17]. An example of a condensation is the blue graph of figure 2.2.

Definition 16. *Given some STS with nodes N and transitions T and strongly connected components SCC, the condensation Q_{STS} is a graph with a set of nodes N_Q and a set of transitions T_Q. Also:*

- *for each $scc \in SCC$ there is a node $n_{scc} \in N_Q$ and there is a function δ which maps any node $n \in N$ to a condensation node n_{scc} so $n \in scc$*
- *for any transition $t \in T$ between nodes o_t and d_t where $\delta(o_t) \neq \delta(d_t)$ there is a transition $t^{\delta(o_t)}_{\delta(d_t)} \in T_Q$ with origin node $\delta(o_t)$ and destination node $\delta(d_t)$*

Corollary 1. *The condensation Q_{STS} with nodes N_Q is acyclic: there is no path from a node $n_{scc} \in N_Q$ to node n_{scc} except for the empty path.*

2.6 Conclusion

With this chapter we have introduced the basic concepts to prepare the reader for the following chapters. In chapter 2.1 we have introduced the basic concepts of first-order logic. We have introduced a specific type of first-order logic formulas as constraints in chapter 2.2 where the domain is set to a number of known domains such as integer numbers and strings. We have also introduced SMT solvers in chapters 2.3 which are able to find models for first-order logic formulas and constraints. We have used constraints to define symbolic transition systems (STSs) in chapter 2.4. Finally we have also introduced strongly connected components (SCCs) in chapter 2.5 where we have also defined the condensation of a STS and cited an algorithm to find the SCCs for a STS.

Chapter 3

Related Techniques & Tools

In this chapter we will detail related work on the verification of STSs. In chapter 3.1 we will explain mathematical induction and how it is used by Donaldson et. al. [15] to substitute cycles for a single property in control flow graphs; effectively summarizing the result of any number of cycle iterations into a single step. In chapter 3.2 we will show the tool nuXmv [11] which is able to verify safety properties for the class of infinite state systems using an algorithm named IC3 [11]. In chapter 3.3 we will show the tool Z3 which is a SMT-solver but also a model checker for infinite state systems implementing a synonymous IC3 strategy as nuXmv named the PDR strategy [19]. In chapter 3.4 we shall show the model checker MCMT [4] which is able to verify safety properties for the class of array-based infinite state systems. We shall also show that it is not possible to specify arbitrary STSs in the input language of MCMT. Finally, in chapter 3.5 we shall show an algorithm which summarizes all reachable states as one property which is used to verify if some state is reachable.

We shall evaluate for nuXmv, Z3 and MCMT if it is possible to specify arbitrary STSs in their input language. We check if the domains for our definition of constraints (chapter 2.2) may be used and if the STS structure may be specified in the input language. We shall see that this isn't possible for MCMT in general while it will be possible for nuXmv and Z3. For nuXmv and Z3 we shall show how it is possible with an example of specifying the bank account example (figure 2.1) in the input language of the tools. For MCMT we shall also give the example of the bank account example specified in MCMT input language but also detail why it isn't possible in general.

3.1 Mathematical induction

Mathematical induction is a proof technique with which you can prove properties inductively by showing a property holds for some base case and showing that the property is preserved for succeeding cases. Formally the prove obligation is as follows:

Say we take a range L from l to ∞ with l is some integer number and we want to prove property $P(o)$ holds for any $o \in L$.

- Base case: Prove $P(l)$

- Inductive step: For all $i \in L$, show $P(i) \rightarrow P(i+1)$ is valid.

Mathematical induction is sound as we know the base case $P(l)$ holds and each subsequent use of the statement based on l e.g. $P(l+1)$, $P(l+2)$... also holds. More concrete example: Say the base case is 0. We know it holds for 0 and for $i+1$. Therefore it must hold for 1 as $0+1=1$. Therefore, we also know it works for 2, 3....

3.1.1 Removing Cycles with Inductive properties

Donaldson et. al. [15] show how induction may be used to prove that some property holds for any number of cycle iterations for some cycle in control flow graphs. If a property is proven to be preserved before and after a cycle iteration, Donaldson et. al. show how the control flow graph may be altered to remove the cycle while preserving the semantics. To explain we use figure 3.1 where we use a STS instead of a control flow graph. This STS contains a single cycle which is the self loop on node *i >= 0*

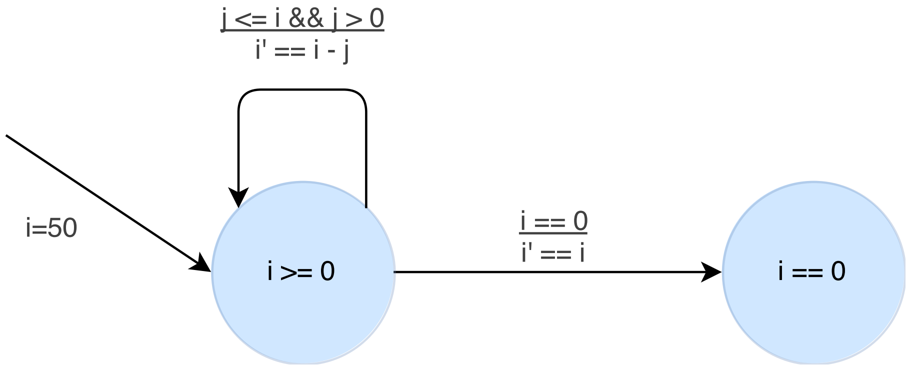


Figure 3.1: STS where i eventually becomes 0. Inductively one can prove $i >= 0$ within and after the cycle.
Guards are above the line, relations below the line for transitions

For the cycle to be removed, we need some property P which will hold before and after any cycle iterations for node *i >= 0*. Due to our use of cycle iterations, range L is $L : 0 \mathinner{..} \infty$ or $L = \mathbb{N}$. The proof obligation using induction over cycle iterations then changes to:

Given node n exists within a cycle and some integer $o \in \mathbb{N}$, we want to prove property $P(o)$ holds in n for o cycle iterations starting and ending in n. Also take an integer k where $k \geq 0$. $T(o)$ are the constraints introduced by the oth cycle iteration.

- Base cases: Show $T(0) \rightarrow P(0)$ is valid
- Inductive step: For all $o \in \mathbb{N}$, show $[P(o) \wedge T(o+1)] \rightarrow P(o+1)$ is valid.

$T(0)$ coincides with the initial state. The initial state are the constraints valid before any cycle iterations.

Now, in figure 3.1 we can take $i \geq 0$ for property P. To show P holds in node *i >= 0*, we use induction over the cycle iterations from and to node *i >= 0*. First we have to show P holds for the base case $P(0)$ which means P holds before any cycle iterations. The proof obligation is then:

Base case: Show $i == 50 \rightarrow i \geq 0$ is valid

For our example it is trivial to see the base case holds as $i == 50 \rightarrow i \geq 0$ or $50 \geq 0$ which is always valid.

Now we also have to show P holds for any number of cycle iterations. Therefore we have to show $P(o + 1)$ holds given $P(o)$ holds. To show this, we need some scoping as variable i is referenced before and after the transition. We shall use a cycle iteration number to identify the variable. i_o means variable i in the oth cycle iteration. Now we have to show:

Inductive case: For all $o \in \mathbb{N}$ show
$[i_o \geq 0 \wedge j_o \leq i_o \wedge j_o > 0 \wedge i_{o+1} == i_o - j_o] \rightarrow i_{o+1} \geq 0$ is valid

Again this statement is valid. The proof is as follows: $i_o \geq 0$, $j_o > 0$, $j_o \leq i_o$ and $i_{o+1} == i_o - j_o$ must hold or else there is no proof obligation due to the implication. If $i_{o+1} \geq 0$ was not true, we know $i_o - j_o < 0$ must hold as $i_{o+1} == i_o - j_o$. This would mean $i_o < j_o$. However, we know $j_o \leq i_o$. Therefore $i_{o+1} \geq 0$ must hold and the statement is valid.

Using induction over the number of cycle iterations we have now shown that P is always true in node *i >= 0*.

Donaldson et. al. show in their work that the cycle may be substituted by a structure which represents the proof obligations which show P is valid in node *i >= 0*. As we do not use the replacement structure as introduced by Donaldson et. al, we shall not show this structure.

3.1.2 Completeness

Not all possible loop invariants may be proven through induction. For example, take for the initial state $T(0) : i > 0$, the transition $T(o) : i' == i * i_{abs} \wedge abs(i.i_{abs})$ where $abs(i, i_{abs})$ is the absolute relation that i_{abs} is the absolute number of i and for $P : i > -10$. Our proof obligations become:

- Base case: Show $i > 0 \rightarrow i > -10$ is valid
- Inductive step: For all $o \in \mathbb{N}$, show $[i_o > -10 \wedge i_{o+1} == i_o * i_{abs,o} \wedge abs(i_o, i_{abs,o})] \rightarrow i_{o+1} > -10$ is valid.

The base case is valid as $0 > -10$. The inductive step, however, is not valid. A counterexample is $i_o == -9$, $i_{abs,o} == 9$ and $i_{o+1} == -81$ as the inductive step then instantiates to:

- $[-9 > -10 \wedge -81 == -9 * 9 \wedge abs(-9, 9)] \rightarrow -81 > -10$
- $[\mathit{true} \wedge \mathit{true} \wedge \mathit{true}] \rightarrow \mathit{false}$
- $\mathit{true} \rightarrow \mathit{false}$
- false

An inductive property for this example is $i > 0$ (we leave the proof to the reader). $i > 0$ implies $i > -10$ and therefore the property $i > -10$ holds but is not inductive.

3.1.3 k-Induction

k-Induction is an extension to mathematical induction. Instead of solving for a single base case, with k-induction you prove k base cases. For the inductive step you may then assume k cases hold and show that the property still holds after $k+1$ steps. We shall reference this extension but not further explain it. k-induction may prove more properties than traditional mathematical induction. An example is shown by Wahl [27].

3.1.4 Relation to our Work

We could use mathematical induction for our approach. It allows us to (over-approximately) summarize cycles as a single property for unbounded variables and verify if that invariant is indeed correct. There are, however, a number of considerations:

- Mathematical induction needs a property P for each cycle in the STS.
- Only inductive properties may be proven with this technique. There exist properties which cannot be proven with mathematical induction.

We shall show in chapter 4 how we will manage these considerations with our approach.

3.2 nuXmv

nuXmv [11] is a symbolic model checker based on the popular NuSMV [10]. The name of the tool and the input language used is similar as with NuSMV but the techniques used by nuXmv are significantly different from any of the techniques employed by NuSMV. NuSMV can only check specifications with a finite state space while nuXmv is able to also check an infinite state space with unbounded variables. The reason nuXmv is able to check an infinite state space is due to the procedures employed and the use of a SMT solver over a SAT solver.

3.2.1 IC3

nuXmv contains a suite of techniques to check safety properties or LTL specifications. The latest is a technique based on IC3 [11]. The goal of IC3 is to show that a property, called the goal property, is invariant or valid within the system. We also refer to this goal property as the safety property. If the goal property is not valid within the system, IC3 finds a counterexample. IC3 is an algorithm which uses symbolic over-approximations of each set of states reachable by a number of steps in the specification represented by a first-order logic formula. IC3 uses one formula to express the state reachable after all paths of a certain length n and this formula is called the frame for n steps. The idea is to find all frames and to show the frames imply the goal property. If two frames are identical they are merged and considered a fixpoint which do not need further analysis. The relation between two subsequent frames is implication as a frame implies the next frame given all possible transitions.

IC3 also has two refinement steps. With the first refinement step, the algorithm refines all frames with information about the (negation of the) goal property while still preserving the soundness of the frame. If this strengthening succeeds, the negation of the goal property is included in the frames making counterexamples of the goal property impossible.

The second refinement step is done when a counterexample is found. There is a frame which results in the counterexample after one transition. The frames leading up to this frame are strengthened, if possible, with information about the (negation of the) counterexample to make the counterexample unreachable. The strengthening is again checked if the new frame is still a sound approximation of the set of the states.

IC3 terminates when a correct counterexample of the goal property is found or when there is no new frame possible and all of the frames imply the goal property.

3.2.2 (Symbolic) Bounded Model Checking

nuXmv also has a bounded model checking [9] strategy using various algorithms including an interpolation-based algorithm [21]. The idea of bounded model checking is to find all reachable states within a bounded maximum path length. Any properties to be verified are verified using this partial state space. A property is said to hold if the property is valid for all states in the partial state space. While it is not exhaustive, in practice it is used to find most bugs [9].

The symbolic bounded model checking strategy [9] is a technique where sets of possible states are represented using (approximated) first-order logic formulas and transitions are modelled between these symbolic representations. IC3 is such a symbolic model checking strategy. In some cases, such as IC3, the procedure is exhaustive.

Unfortunately we have not been able to find publications of the exact bounded model checking strategies employed by nuXmv except for IC3. We approached the authors of nuXmv for any publications but unfortunately they haven't given us any references.

3.2.3 Induction over Paths

nuXmv also has a model checking method which uses mathematical induction over the length of the paths. The goal of this technique is to prove or disprove that a goal property is invariant or valid within the system. It checks for each transition if it reaches a state where the goal property may be false assuming the goal property is valid before the transition.

3.2.4 nuXmv Input language

An example of the input language for nuXmv may be seen in listing 3.1. Any specification must start with the definition of the module name. Next all variables must be defined. In our example, we included the `VAR`iables for the current state, actions, current deposit/withdrawal amount and the account balance.

The starting transitions are specified with the keyword `INIT`. Leaving transitions for a single node are specified with a single transition formula with the `TRANS` keyword. Finally, safety properties to be verified may be defined with the `INVARSPEC` keyword.

Listing 3.1: Example of nuXmv specification for a simple bankaccount which can open/close/freeze and deposit/withdraw money.

```
MODULE main

VAR node : { open,frozen };
VAR balance : real;
VAR status : { OPEN,CLOSED,FROZEN };
VAR amount_withdrawal_t : real;
VAR amount_deposit_t : real;

INIT ((node = open) & ((balance = 50) & (status = OPEN)));

-- Transitions withdrawal, deposit and freeze
TRANS (node = open -> ((
   -- Withdraw transition
     (amount_withdrawal_t > 0) & (amount_withdrawal_t <= balance)
      & (next(balance) = (balance - amount_withdrawal_t)
      & (next(status) = OPEN)) & (next(node) = open))
   -- Deposit transition
   | (amount_deposit_t > 0) & ((next(balance) = (amount_deposit_t + balance)
      & (next(status) = OPEN)) & (next(node) = open))
   -- Freeze transition
   | (status = OPEN) & ((next(balance) = balance) & (next(status) = FROZEN)
      & (next(node) = frozen))));

-- Transitions unfreeze
TRANS (node = frozen -> ((
   -- Freeze transition
     (status = FROZEN) & ((next(balance) = balance)
      & (next(status) = OPEN)) & (next(node) = open))));

INVARSPEC (balance < 10000);
```

3.2.5 Specifying STSs in nuXmv

It is possible to specify arbitrary STSs in nuXmv. With this subsection we shall show one method of how to use specify an arbitrary STS in the input language of nuXmv. Listing D.1 shows the complete template with pseudo code on how to specify a STS in nuXmv. The following paragraphs explain the complete template part by part.

On lines 3 to 6 (listing 3.2) we define a variable `node` which is an enumeration and the values are the labels of each node in order to specify the nodes of the STS.

```
3  VAR node : { [foreach node]
4                 node.label [node isNotLast] , [end isNotLast]
5               [end foreach]
6              };
```

Listing 3.2: Pseudo code on how to specify the nodes of an STS in nuXmv input language.

On lines 8 to 10 (listing 3.3) we define all state variables as `VAR`.

```
8  [foreach state variable as var]
9    VAR var.label : var.type;
10 [end foreach]
```

Listing 3.3: Pseudo code on how to specify the state variables of an STS in nuXmv input language.

On lines 12 to 14 (listing 3.4) we also define all transition variables as `VAR` and the label is the combination of the transition variable label and the transition label. nuXmv only has global variables similar to STS state variables. Therefore, transition variables have to be declared as a `VAR`.

```
12 [foreach transition variable as var]
13   VAR var.label_var.transition.label_t : var.type;
14 [end foreach]
```

Listing 3.4: Pseudo code on how to specify the transition variables of an STS in nuXmv input language.

On lines 16 to 20 (listing 3.5) we define all start transitions as a single `INIT`. When multiple `INIT` properties are defined, they are combined as a conjuction. For a STS with multiple start transitions, one start transition may be chosen as the starting point. Therefore, we combine all start transitions as a disjunction for a single `INIT`. We also include the destination node of the start transition as a conjunction to each start transition.

```
16 INIT [foreach start transition as trans]
17        ( (node = trans.destination)
18          & trans.relation
19        ) [trans isNotLast] | [end isNotLast]
20      [end foreach];
```

Listing 3.5: Pseudo code on how to specify the start transitions of an STS in nuXmv input language.

On lines 22 to 28 (listing 3.6) we define a `TRANS` for all transitions leaving some node. We imply the current state is from that origin node and one of the transitions leaving that node must be true. Therefore, we combine the guards and the relations of each leaving transition as a disjunction.

```
22 [foreach node as n]
23   TRANS (node = n.label -> (
24     [foreach n.leaving_transitions as trans]
25       (trans.guard) & (trans.relation) |
26     [end foreach]
27   ));
28 [end foreach]
```

Listing 3.6: Pseudo code on how to specify the transitions of an STS in nuXmv input language.

Finally, on lines 31 to 33 (listing 3.7) we define the safety property as a `INVARSPEC` which is a property to be checked globally for all reachable states.

```
31 [foreach safety property as prop]
32 INVARSPEC (prop);
33 [end foreach]
```

Listing 3.7: Pseudo code on how to specify the safety property for an STS in nuXmv input language.

The complete listing of the previous paragraphs is shown in listing D.1 of appendix D.

nuXmv uses the `next(<var name>)` function to reference the value of the next state. For the relation of the transitions, we map each variable referencing the state in the destination node to use this `next` function with the variable label as the argument.

3.2.6 Instrumenting nuXmv

Instrumenting nuXmv is not straightforward as the tool allows for a great deal of techniques and checks. We have instrumented nuXmv with the commands in listings 3.8, 3.9 and 3.10. First we start the nuXmv program. Then we call `go_msat` to parse the specification. Finally the respective call to start some algorithm is done.

Listing 3.8: Instrumenting nuXmv to check safety property of specification using IC3

```
nuxmv -int Account.smv
go_msat
check_invar_ic3 -i
```

Listing 3.9: Instrumenting nuXmv to check safety property of specification using mathematical induction

```
nuxmv -int Account.smv
go_msat
msat_check_invar_bmc -a classic
```

Listing 3.10: Instrumenting nuXmv to check safety property of specification using interpolant bounded model checking

```
nuxmv -int Account.smv
go_msat
msat_check_invar_bmc -a interpolants
```

3.2.7 Verifying Cycles with IC3

nuXmv may use IC3 to discover the state space and find counterexamples or it proves that safety properties are (in)valid. In our example listing 3.1 we have included a safety property that the balance has to remain below `100000` while we are only able to deposit a maximum of `50` per deposit. Trying to check this safety property is difficult for nuXmv due to the nature of the safety property and the deposit cycle. Every iteration of the deposit cycle leads to a new state and this cycle can only be unwinded until a counterexample is found or a property is found which holds before and after any cycle iteration. IC3 tries to discover new states incrementally and the counterexample can only be found by unwinding the cycle until a balance greater than `100000` is found. As we limited the deposit to a maximum of `50` per deposit, it will take a minimum of *2000* cycle iterations. These types of safety properties forcing the cycles to unwind are important for ING and should be solved in reasonable time.

3.2.8 Verifying Cycles with Mathematical Induction

nuXmv may use mathematical induction to verify a safety property for some cycle. With mathematical induction, it may be necessary that the safety property is strong enough to show any relations between variables. As an example, we take the nuXmv specification of listing 3.11. The specification describes a single node system with a single self loop where variable i is incremented by variable $0 \leq j \leq 50$. The safety property `i >= 0` is true, but with mathematical induction nuXmv is not able to verify this property. When mathematical induction tries to verify this safety property, it assumes `i >= 0` before the transition. Now variable j is unbounded and the counterexample `i = 0 & j = -1 & next(i) = -1 & next(j) = 1` is given. The safety property describes nothing about variable j and as such, mathematical induction has to assume j is unbounded which leads to the spurious counterexample.

This is also an example that it is hard to reason if a counterexample given by mathematical induction is an actual counterexample or just the failing of mathematical induction to prove some property. The given counterexample by nuXmv is a spurious counterexample as know the state with $j \leftarrow -1$ is impossible to reach as $j > 0$ is globally valid. We can therefore not trust any counterexamples given by mathematical induction as a correct counterexample as we first have to verify if the counterexample is reachable.

Listing 3.11: Example of nuXmv specification where the safety property is not strong enough to prove or disprove it.

```
MODULE main

VAR i : integer;
VAR j : integer;

INIT i = 0;
INIT j = 50;

TRANS
  case
    TRUE : next(i) = i + j & next(j) > 0 & next(j) < 50;
  esac;


INVARSPEC i >= 0;
```

3.2.9 Feasibility of verifying STS with nuXmv

nuXmv is able to check safety properties of a STS. However, there are a couple of issues:

- Support and development for nuXmv seems to be minimal. We sought contact with the authors and creators of nuXmv and they noted a new version of nuXmv is planned but they have no timeline on when they will create or release it.
- nuXmv uses the definition of an infinite state system with a single transition function losing the structure of the STS in the process. This makes it hard to reason about cycles in specifications. This leads to performance issues when discovering this structure such as the difficulty in proving the invariant of listing 3.1 with IC3. We confirmed this with an experiment which we detail in chapter 5.5.
- nuXmv currently does not support non-linear arithmetic constraints [11].
- The results from nuXmv found with the bounded model checking strategy may not be exhaustive.
- nuXmv may use mathematical induction but we need extra verification to check if a counterexample is spurious. Also, mathematical induction depends on the properties it needs to verify to understand the relations between variables. The proof technique may fail with spurious counterexamples when the properties are not strong enough or when the proof technique is not strong enough to prove the property.
- nuXmv does not allow variables of the string domain

3.3 Z3

Z3 is a multi-purpose formal methods tool developed by Microsoft [14]. We have described in chapter 2.3.1 already how Z3 may be used as a SMT-solver. In this chapter we shall describe the Property-Directed Reachability (PDR) engine of Z3 created by Hoder and Bjorner [19]. PDR may be used to check if some state is reachable within the class of infinite state systems. PDR is a synonym for the algorithm IC3 [19] which is also implemented in nuXmv (chapter 3.2.1) although the input language used by Z3 is different. With this chapter we shall show the Z3 PDR input language, how to instrument Z3 and any considerations when using Z3 with the PDR engine to verify properties of arbitrary STSs. A more detailed look at IC3/PDR is in chapter 3.2.1. While nuXmv with IC3 has not implemented non-linear arithmetic, Hoder and Bjorner have included non-linear arithmetic operators [19].

3.3.1 Z3 PDR Input Language

The input language for the PDR engine of Z3 is extended SMT-LIB. As with the Z3 SMT input language it uses prefix notation although the function and keywords are different. An example of a simple bank account STS is shown as listing 3.12 where we check if some safety property is violated. The main concept behind the input language is relational: we define relations and show which members are contained in the relations using `rule`s to define assertions which constrain which members are in the relations. Relations in this context are the relations from relational logic.

We defined the enumeration status as a non-algebraic datatype on line 1. We have also defined the possible nodes as a non-algebraic datatype on line 3.

We defined the relations consisting of all reachable states and all reachable states violating our safety property as respectively the relation `states` and `faulty_states` on lines 6 and 8. In the specification we will define rules to show which values (of the state variables) are in the relations.

We defined the state and transition variables between lines 10 to 17 with the `declare-var` keyword. For each state variable we define a variable representing the current value and a variable representing the next value. Each transition variable is given a label starting with the variable name, the transition label and the postfix `_t` to know it is a transition variable.

With our example we are verifying if there is a state reachable which falsifies the safety property defined lines 75 to 78. Constraints such as the safety property may be defined as functions with some property using the keyword `define-func`. The function returns a boolean depending on if the property may be evaluated to true or false.

All transitions are also defined as functions with the keyword `define-func`. We joined a check if the current node is the origin node, the guard, the transition relation and the next node as the destination node as a conjunction. These properties show the relation between the states of each node.

The start transition is single function with the disjunctive combination of all start transition relations. We have defined the start transition for our example on lines 19 to 23 as the `initial` function.

Finally we also have to show which members are contained by the `states` and `faulty_states` relations. This is defined with rules 80 to 105. We add a rule that shows all states from the start transitions should be in the `states` relation on line 81. The rule on lines 85 to 92 shows the rule that for any state in `states` and any next state which satisfies some transition, the next state should also be in `states`. These two rules together are similar to the recursive definition 14 of reachable state in chapter 2.4. As we are checking for a safety property, we also have to check for each reachable states in `states` if it falsifies the safety property. This is done with the rule on lines 97 to 105 and any states which violate the safety property should be in `faulty_states`.

With this definition we can query Z3 for all members of `faulty_states`. This is done on line 109. If the relation is empty, the specification does not violate the safety property. Otherwise, the relation contains counterexamples for the safety property.

Listing 3.12: Example of Z3 PDR specification for a simple bank account which can open/close/freeze and deposit/withdraw money.

```
 1  (declare-datatypes () ((status OPEN CLOSED FROZEN)))
 2
 3  (declare-datatypes () ((Node open frozen)))
 4
 5  ; The collection of reachable, faulty states
 6  (declare-rel faulty_states (Node Real status))
 7  ; The collection of reachable states
 8  (declare-rel states (Node Real status))
 9
10  (declare-var next_node (Node))
11  (declare-var node (Node))
12  (declare-var balance (Real))
13  (declare-var next_balance (Real))
14  (declare-var status (status))
15  (declare-var next_status (status))
16  (declare-var amount_withdrawal_t (Real))
17  (declare-var amount_deposit_t (Real))
18
19  (define-fun initial () Bool
20    (or
21      (and (= node open) (and (= balance 50) (= status OPEN)))
22    )
23  )
24
25  ; unfreeze transition
26  (define-fun unfreeze () Bool
27    (and
28      (= node frozen)
29      (= next_node open)
30      (= status FROZEN)
31      (and (= next_balance balance) (= next_status OPEN))
32    )
33  )
34
35  ; freeze transition
36  (define-fun freeze () Bool
37    (and
38      (= node open)
```

```
    (= next_node frozen)
    (= status OPEN)
    (and (= next_balance balance) (= next_status FROZEN))
  )
)

; deposit transition
(define-fun deposit () Bool
  (and
    (= node open)
    (= next_node open)
    (> amount_deposit_t 0)
    (and (= next_balance (+ amount_deposit_t balance)) (= next_status OPEN))
  )
)

; withdrawal transition
(define-fun withdrawal () Bool
  (and
    (= node open)
    (= next_node open)
    (and (> amount_withdrawal_t 0) (<= amount_withdrawal_t balance))
    (and (= next_balance (- balance amount_withdrawal_t)) (= next_status OPEN))
  )
)

; Any of the following transitions
(define-fun transition () Bool
  (or
    withdrawal
    deposit
    freeze
    unfreeze
  )
)

; safety property
(define-fun invariant () Bool
  (< balance 10000)
)

; Any state satisfying the initial function is a reachable state
(rule (=> initial (states node balance status)))

; For any origin state in reachable states, any destination state
; that satisfies any transition is in reachable states
(rule
  (=>
    (and
      (states node balance status)
      transition
    )
    (states next_node next_balance next_status)
  )
)

; Any reachable state that counters the safety property
; is a faulty state
(rule
  (=>
    (and
      (states node balance status)
      (not invariant)
    )
    (faulty_states node balance status)
  )
)

; Ask for all faulty states
; Should be empty if safety property is preserved
(query faulty_states)
```

3.3.2 Specifying STSs in Z3 PDR

It is possible to specify arbitrary STSs in Z3 with the PDR strategy. With this subsection we shall show one method of how to use specify an arbitrary STS in the input language of Z3 PDR. Listing D.2 shows the complete template with pseudo code on how to specify a STS in Z3. The following paragraphs explain the complete template part by part.

On lines 1 to 5 (listing 3.13) we define an enumeration `Node` where the values are the labels of the nodes in the STS.

```
1  (declare-datatypes () ((Node
2      [foreach node as node]
3        node.label
4      [end foreach]
5  )))
```

Listing 3.13: Pseudo code on how to specify the nodes of an STS in Z3 PDR input language.

On lines 7 to 9 (listing 3.14) we define the possible enumerations and the values of the enumeration as a separate type.

```
7  [foreach enumeration as enum]
8    (declare-datatypes () ((enum.label enum.values)))
9  [end foreach]
```

Listing 3.14: Pseudo code on how to specify the enumerations of an STS in Z3 PDR input language.

On lines 11 to 22 (listing 3.15) we declare the `states` and `faulty_states` relations and the types of the state variables as their arguments. These relations now contain members of the type tuple and each of the members of the tuple coincide with the type of a state variable. We use these relations to respectively represent the reachable state and the reachable state which violates the safety property.

```
11 ; The collection of reachable, faulty states
12 (declare-rel faulty_states (Node
13 [foreach state variable as var]
14   var.type
15 [end foreach]
16 ))
17 ; The collection of reachable states
18 (declare-rel states (Node
19 [foreach state variable as var]
20   var.type
21 [end foreach]
22 ))
```

Listing 3.15: Pseudo code on how to specify the reachable and faulty states of an STS in Z3 PDR input language.

On lines 24 and 25 (listing 3.16) we define the variables `node` and `next_node` which represent the origin and destination node of the transition being taken.

```
24 (declare-var node (Node))
25 (declare-var next_node (Node))
```

Listing 3.16: Pseudo code on how to specify the variables containing the current and next node of an STS in Z3 PDR input language.

On lines 27 to 30 (listing 3.17) we define for each state variable two variables; a variable representing the current value of the state and a variable representing the next value of the state.

```
27 [foreach state variable as var]
28   (declare-var var.label (var.type))
29   (declare-var next_var.label (var.type))
30 [end foreach]
```

Listing 3.17: Pseudo code on how to specify the state variables of an STS in Z3 PDR input language.

On lines 33 and 35 (listing 3.18) we define each transition variable also as a global variable. The name of this variable is the combination of the transition variable label and the transition label.

```
33 [foreach transition variable as var]
34   (declare-var var.label_var.transition.label_t (var.type))
35 [end foreach
```

Listing 3.18: Pseudo code on how to specify the transition variables of an STS in Z3 PDR input language.

On lines 38 to 44 (listing 3.19) we define the `initial` function which is a disjunction of the relations of all start transitions. We also include the destination node as a conjunction with each of the relations.

```
38 (define-fun initial () Bool
39   (or
40     [foreach start transition as trans]
41       (and (= node trans.destination) (trans.relation))
42     [end foreach]
43   )
44 )
```

Listing 3.19: Pseudo code on how to specify the start transitions of an STS in Z3 PDR input language.

On lines 46 to 55 (listing 3.20) we define each transition. We combine the destination node, guard, relation and destination node as a conjunction.

```
46 [foreach transition as trans]
47   (define-fun trans.label () Bool
48     (and
49       (= node trans.origin)
50       (= next_node trans.destination)
51       (trans.guard)
52       (trans.relation)
53     )
54   )
55 [end foreach]
```

Listing 3.20: Pseudo code on how to specify the transitions of an STS in Z3 PDR input language.

On lines 58 to 64 (listing 3.21) we define a function `transition` which is the disjunction of all transition functions.

```
58 (define-fun transition () Bool
59   (or
60     [foreach transition as trans]
61       trans.label
62     [end foreach]
63   )
64 )
```

Listing 3.21: Pseudo code on how to specify a function containing all transitions of an STS in Z3 PDR input language.

On lines 66 to 72 (listing 3.22) we define the `safety_property` function as the conjunction of all safety properties.

```
66 (define-fun safety_property () Bool
67   (and
68     [foreach safety property as prop]
69       prop
70     [end foreach]
71   )
72 )
```

Listing 3.22: Pseudo code on how to specify a the safety property for an STS in Z3 PDR input language.

On lines 75 to 79 (listing 3.23) we define a rule stating that each state which evaluates the `initial` function as true, should be in the `states` relation.

```
75 (rule (=> initial (states node
76 [foreach state variable as var]
77   var.label
78 [end foreach]
79 )))
```

Listing 3.23: Pseudo code on how to specify what states are reachable states due to a start transition in Z3 PDR input language.

On lines 83 to 99 (listing 3.24) we define a rule stating that for each state in `states` and some next state which evaluates at least one transition function as true that the next state should also be in `states`.

```
83 (rule
84   (=>
85     (and
86       (states node
87         [foreach state variable as var]
88           var.label
89         [end foreach]
90       )
91       transition
92     )
93     (states next_node
94       [foreach state variable as var]
95         next_var.label
96       [end foreach]
97     )
98   )
99 )
```

Listing 3.24: Pseudo code on how to specify what states are reachable states due to a path to that state in Z3 PDR input language.

On lines 103 to 119 (listing 3.25) we define that each state in `states` which falsifies the `safety_property` must also be in `faulty_states`.

```
103 (rule
104   (=>
105     (and
106       (states node
107         [foreach state variable as var]
108           var.label
109         [end foreach]
110       )
111       (not safety_property)
112     )
113     (faulty_states node
114         [foreach state variable as var]
115           var.label
116         [end foreach]
117     )
118   )
119 )
```

Listing 3.25: Pseudo code on how to specify what reachable states are faulty in Z3 PDR input language.

Finally on line 123 (listing 3.26) we query the members of the `faulty_states` relation. This relation contains counterexamples for the safety property which are reachable.

```
123 (query faulty_states)
```

Listing 3.26: Pseudo code on how to query the faulty states in Z3 PDR input language.

The complete listing of the previous paragraphs is shown in listing D.2 of appendix D.

3.3.3 Instrumenting Z3

Instrumenting Z3 as a SMT-solver or using the PDR strategy is straightforward. After defining the SMT or PDR specification, one may just call Z3 with the specification as in listing 3.27. Z3 figures out which strategy to use based on the specification used.

Listing 3.27: Instrumenting Z3 to check a safety property for the bank account specification using the PDR strategy

```
z3 BankAccount.smt2
```

3.3.4 Feasibility of Verifying STS with Z3

Z3 with PDR uses the same algorithm as nuXmv and therefore shares the same considerations. The input language, however, is fundamentally different and Z3 has support for non-linear arithmetic. With our example in listing 3.12 we have shown how to check for safety properties. Considerations:

- Z3 uses the definition of an infinite state system with a number of rules losing the structure of the STS in the process. This leads to performance issues when discovering this structure such as the difficulty in proving the safety property of 3.12 with PDR. We confirmed this with an experiment which we detail in chapter 5.5.
- Any members of the `faulty_states` relation only show the direct values which violate the safety property. We are unable to show a path from some starting transition to the violating reachable state.

3.4 Model Checker Modulo Theories (MCMT)

Model Checker Modulo Theories (MCMT) [4] [18] is a symbolic model checker [9] for array-based infinite state systems. The core of the checker uses array logic theories to solve first-order logic formulas involving arrays. The state systems which are checked are a variation of our definition of STS. The main difference is that the state vector is represented as a vector of (in)finite arrays. An example of our simple account specification written in the MCMT input language can be seen in listing 3.28.

3.4.1 MCMT language

The MCMT input language is low-level. For instance, it does not implement subtraction directly. The scalar -1 together with the addition operator must be used to for subtraction. Any variables have to be declared using
`:local name type` or `:global name type` which respectively means an array *name* where each element is of type *type* and an array where each element has the same value. The `:global` array is used when you need a variable with a single value such as a real or integer. In our example listing, we only used `:global` arrays as we only needed variables with single values and no arrays.

The initial state is given as a single property as the block starting with the `:initial` keyword. Some safety property may be specified as a property starting with the `:unsafe` keyword. Transitions are blocks starting with the `:transition` keyword. For each transition there is a guard property. If the guard evaluates to true, then the `:val` statements are evalated. These statements may be read as $next(v_i) = some\ value$; in other words, they are direct value assignments for the variables of the next state. The `:val` statements reference the variables in the order in which they are defined. The first `:val` updates the value of the first variable defined and so on.

Listing 3.28: Example of MCMT specification for a bank account which can open/close/freeze and deposit/withdraw money.

```
:comment current state. 1=open 2=frozen
:global node int

:comment balance of the account
:global balance real

:comment amount of next deposit
```

```
:global amount_deposit real

:comment amount of next withdrawal
:global amount_withdrawal real

:initial
:var x
:cnj (= balance 50) (= node 1) (> amount_deposit 0) (> amount_withdrawal 0)

:unsafe
:var x
:cnj (> balance 10000)

:comment 1 deposit
:transition
:var j
:guard (= node 1) (> amount_deposit 0) (< amount_deposit 51)
:numcases 1
:case
:val 1
:val (+ balance amount_deposit)
:val 1
:val 1

:comment 2 withdraw
:transition
:var j
:guard (= node 1) (<= amount_withdrawal balance) (< amount_withdrawal 51)
:numcases 1
:case
:val 1
:val (+ balance (* -1 amount_withdrawal))
:val 1
:val 1

:comment 3 freeze
:transition
:var j
:guard (= node 1)
:numcases 1
:case
:val 2
:val balance
:val 1
:val 1

:comment 4 unfreeze
:transition
:var j
:guard (= node 2)
```

```
:numcases 1
:case
:val 1
:val balance
:val 1
:val 1
```

3.4.2 Backwards reachability

MCMT [4] uses backwards reachability analysis in a bounded model checking manner with inductive invariant generation as its core algorithm. Backwards reachability [9] is a technique used in bounded model checking. The procedure starts at the error state and tries to exhaustively find a path to a state which is reachable by the state system. If such a path is found it is an example that the error state may be reached. If such a path cannot be found, the system is deemed safe. In infinite state systems this exhaustive approach might not terminate as possible paths might be infinite. One way MCMT tries to solve this non-termination issue is to check for fix points. If the new state is the same as the previous state using a certain transition, there is no reason to search further as the next state for this transition is again the same.

Another way to solve the non-termination issue is to find a proof that the system is safe. MCMT employs the use of complex algorithms [4] called lazy abstraction, acceleration and term abstraction to calculate properties which symbolically represent (part of) the state space. These properties might be over-approximations. The algorithms were too complex to study in detail within the time limit of our project. Using the symbolic over-approximations, MCMT checks if they imply that the system is safe. If a counterexample is found, it is checked if this counterexample is a possible path within the state system. If it is not a correct counterexample, this information is used to refine the symbolic properties.

Finding a reachable path from the error state to the reachable state or finding a set of properties which describe the state system is bounded. MCMT stops execution when a path reaches a configurable maximum length or a property with a set maximum length.

3.4.3 Instrumenting MCMT

We have tried to use MCMT to verify the example in listing 3.28 with the specified unsafe property. The commands we have tried:

- mcmt Account.mcmt
- mcmt -AN Account.mcmt
- mcmt -CN Account.mcmt
- mcmt -Z Account.mcmt
- mcmt -CN -Z Account.mcmt

The *AN*, *CN*, *Z* respectively enable lazy abstraction and refinement, another variant of lazy abstract and refinement and acceleration. In all cases the execution was prematurely ended due to bounds reached. The system in our example should be deemed safe.

3.4.4 Yices & SAFARI

MCMT depends on the SMT solver Yices 1.0 [16]. Yices 1.0 does not work for non-linear integer or real arithmetic with multiple variables. We are unable to say if MCMT will function for non-linear arithmetic if Yices 1.0 is swapped for another SMT solver which does support non-linear arithmetic; this would require MCMT to be modified.

MCMT is reimplemented as the SAFARI [3] model checker although one of the authors (F. Alberti) has made clear all changes have also been integrated in MCMT which makes them practically equal. Alberti made this statement after a request for an executable for SAFARI as the website for SAFARI does not host an executable to be downloaded.

3.4.5 Feasibility of verifying STS

For this checker it is not possible to specify arbritary STSs. The input language has two limitations which we would need in order to specify a STS:

- It is not possible to reference the value of state variable before and after the transition. It is only possible to reference the value of the current variable.
- Specifying the value of a variable after the transition is with direct assignment; some computation that results in a value for the type of the variable such as $4 + 5$ which would return an integer.

Due to these limitations, it is impossible to directly map STS transitions to transitions in the MCMT input language. For instance, within the context of our example of listing 3.28 it is impossible to specify the relation *amount_deposit'* $>$ 0 as a direct direct assignment. We had to set the next value of *amount_deposit* as some value (in this case we set it to 1).

To conclude, we also note these other issues:

- Support and development for MCMT and SAFARI appears minimal with the last publication in 2014 and the last release on the 2nd of February 2017.
- Bounded model checking is not definitive or exhaustive in practice. MCMT is unable to prove our example safe.
- MCMT does not allow non-linear arithmetic.
- Variables in MCMT may not have the string type.

3.5 Reachability Analysis of STS

Bakker [6] shows a method of analysing all reachable states for a STS specification by joining the constraints on each transition as a conjunction for some path. This creates a single property (similar to our reachable state constraint from chapter 4) which symbolizes all reachable states for the destination node resultant from the path.

Then, for each property, Bakker's algorithm uses a proprietary constraint language called Dumont to manipulate and solve constraints to see if a model exists so a state in the destination node is reachable. To solve constraints the constraint program is translated to a Prolog program and executed with a Prolog interpreter. This is also called Constraint Logic Programming or CLP [6].

As this is a method to show if some state is reachable, Bakker uses it to find counterexamples for some safety property. If the method doesn't find a counterexample, the system is deemed to not violate the safety property. Bakker exhaustively explores the state space to find all states reachable after an (optional) maximum path length; similar to bounded model checking. However, Bakker does not seem to deal with infinite cycles which could cause an infinite execution. From this point of view, Bakker's method is theoretically exhaustive if there is no maximum path length but in practice the tool might have a long execution time with specifications such as the example of chapter 3.2.7.

3.6 Conclusion

For our related work we looked at a number of existing verification techniques and tools to see how suitable they are for our case. We found that:

- **Mathematical Induction** - Mathematical induction is a proof technique which may be used to show some property holds for a node in a cycle. We could use mathematical induction as part of our approach considering that mathematical induction is not complete and we will have to propose the property to be proven.

- **nuXmv** - nuXmv shows promise as it is an existing tool to verify safety properties for STS specifications using various strategies. However, the tool does not accept non-linear arithmetic constraints. Also, the IC3 strategy theoretically might exhibit long execution times when proving properties in STSs containing cycles, the backwards reachability strategy isn't exhaustive and the mathematical induction strategy is difficult to use as we may need to supply properties stronger than the ones we want to verify.

- **Z3** - Z3 with the PDR strategy is very promising. We have shown an example of how to verify safety properties for arbitrary STS specifications. The PDR strategy is the same as the IC3 strategy from nuXmv. As with IC3, PDR may exhibit long execution times when proving properties in STSs containing cycles. Z3 has already support for non-linear arithmetic with its PDR strategy.

- **MCMT** - MCMT is an existing tool designed for specifications with arrays but can be leveraged for symbolic transition systems. Unfortunately it isn't possible to specify arbitrary STSs in MCMT input language and MCMT does not support non-linear arithmetic.

- **Reachability analysis of STS -** Bakker's work gives a way to reason about the reachability within a STS. It reasons if there exists a path upto some maximum length which reaches some state. We have shown that a STS may contain an infinite path. Therefore, when no maximum length is supplied, the algorithm may execute infinitely. If a maximum length is supplied it will terminate inevitably but is non-exhaustive.

Chapter 4

Goose

With this chapter we introduce our approach to verifying properties for a Symbolic Transition System (STS) specification. We named this verification procedure Goose. Chapter 4.1 gives an overview of the procedure. We will use constraints called reachable state constraints (RSC) to symbolically summarize which states are reachable per node. The procedure consists of three parts in two phases. The first phase consists of the parts to generate and verify the RSCs and is described in chapter 4.2. The second phase and final part is described in chapter 4.3 and introduces which verification properties we shall verify with our approach and how we may verify them using the RSCs.

We will show in chapter 4.1.3 how we rename the variable names of any constraints within a STS to avoid variable name clashes so the Goose verification procedure may be performed. In the rest of this chapter we shall assume the first scoping strategy has already been performed which distinguishes any variables in constraints to the state variables of the node it is referencing. Any variable name in a RSC, guard or relation is replaced by the variable name joined with the node or transition label it is referencing. This will allow us to combine RSCs, guards and relations into new RSCs without any naming issues.

4.1 Overview Verification Procedure

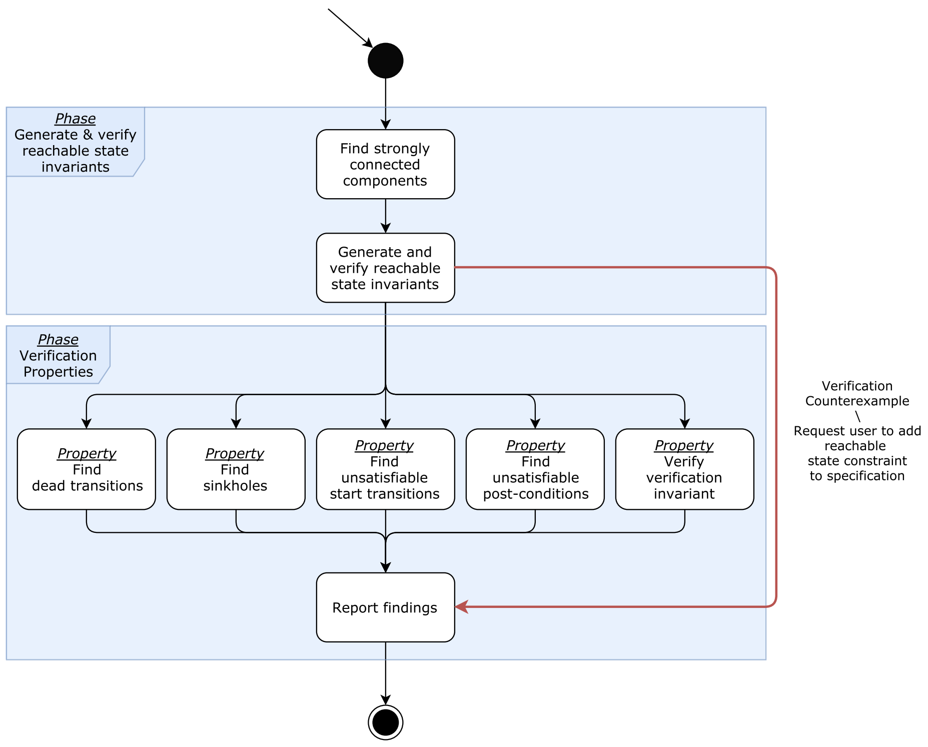
We propose a verification procedure for STS specifications which consists of two phases:

1. Find all reachable states for all nodes (chapter 4.2).
2. Prove or disprove any verification properties (chapter 4.3)

The first phase will consist of generating missing RSCs for each node and verifying the RSCs per strongly connected component (SCC). We will propose one way to generate missing RSCs in chapter 4.2.1 based on the deduction rule (introduced in chapter 4.2.1). We shall see that this rule is not complete in the sense it is unable to find a RSC in every situation. Therefore, we will allow the user to supply RSCs for nodes in the specification of the STS to help the procedure to generate and verify all RSCs. We will propose one way to verify RSCs in chapter 4.2.2 based on the triple verification rule (introduced in chapter 4.2.2).

A simple overview of the algorithm may be seen in figure 4.1. Again, it consists of the two main phases: 1) To generate and verify reachable state constraints for each node and 2) To check any of the verification properties.

To successfully finish phase 1 the user may need to specify additional reachable state constraints as our generation rule set is not complete. In this case, the algorithm will ask the user to add a reachable state constraint for a specific node to the specification and the tool may be run again.



Phase
Generate & verify reachable state invariants
Find strongly connected components
Generate and verify reachable state invariants
Phase
Verification Properties
Property
Find dead transitions
Property
Find sinkholes
Property
Find unsatisfiable start transitions
Property
Find unsatisfiable post-conditions
Property
Verify verification invariant
Report findings
Verification Counterexample \ Request user to add reachable state constraint to specification

Figure 4.1: Simple overview of different phases and individual steps of the algorithm.

Given phase 1 has successfully completed, Goose adds no restrictions to the termination of phase 2. Phase 2 may still not terminate if the SMT does not terminate for some query.

With phase 2 each of the individual properties are checked. They do not have a dependency on each other. Finally, the results are collected and reported.

A more detailed overview of the algorithm is shown in figure 4.2. The first step again consists of finding the sets of nodes which make up the strongly connected components in the STS. This result is used for the first phase of generating and verifying the RSCs for each node. The STS used in this first phase may contain RSCs specified by the user. Generation for missing RSCs and verification of RSCs is done per SCC and at any point we may find that the procedure is unable to generate a RSC for some node or a counterexample is found for some RSC. We follow the condensation graph of the STS to decide which SCC is processed when. The condensation graph shows the dependency between SCCs and is acyclic so it gives an order of work which is always finite and we know which SCCs are dependent on which. When each node has a verified RSC, the second phase begins of proving or disproving the verification properties. Finally, the procedure reports if each of the verification properties hold or gives counterexamples for the respective verification properties.

4.1.1 counterexamples

By first finding the reachable states, we gain an intermediate representation which represents all reachable states for each node. This information may be used to verify all verification properties in phase 2 and to give concrete counterexamples for any of the verification properties. With nuXmv's induction approach(described in chapter 3.2.8) it is unclear if any counterexamples are a failing of the inductive approach or if it is a concrete counterexample. With the information of all reachable states we are able to give concrete counterexamples in phase 2.

We do, however, have the same issue with potentially, spurious counterexamples in phase 1. Our verification rule is also based on induction and may produce counterexamples which are spurious and are a result of the induction proving method failing.

4.1.2 Reachable State Constraint

We shall represent reachable states as a property specified for a certain node. This property should be true for all reachable states for that node; in other words, the property summarizes all states reachable in that node. Therefore, we name these properties reachable state constraints (RSC).

Definition 17. *A reachable state constraint RSC_n for n is a constraint so that for any reachable state ς_n, $\varsigma_n \models RSC_n$.*

Lemma 1. *A reachable state constraint RSC_n for some node n may be an over-approximation of all reachable states for n: There may exist some variable assignment σ so $\sigma \models RSC_n$ but σ is not a reachable state for node n.*

Lemma 1 states that a reachable state constraint may be an over-approximation; it may model states as reachable which are not reachable. Our definition does not state that a reachable state constraint must be false or true for any states which are not reachable in a node. They may be true and therefore also unreachable states may be represented by a reachable state constraint.

The motivation to allow over-approximations is to allow verification for STSs where a node may contain a set of reachable state which may not or is difficult to be represented with a strict reachable state constraint. An example of the fibonacci sequence for the specification of listing C.1 of appendix C. A strict reachable state constraint does not exist although an over-approximated reachable state constraint is given.

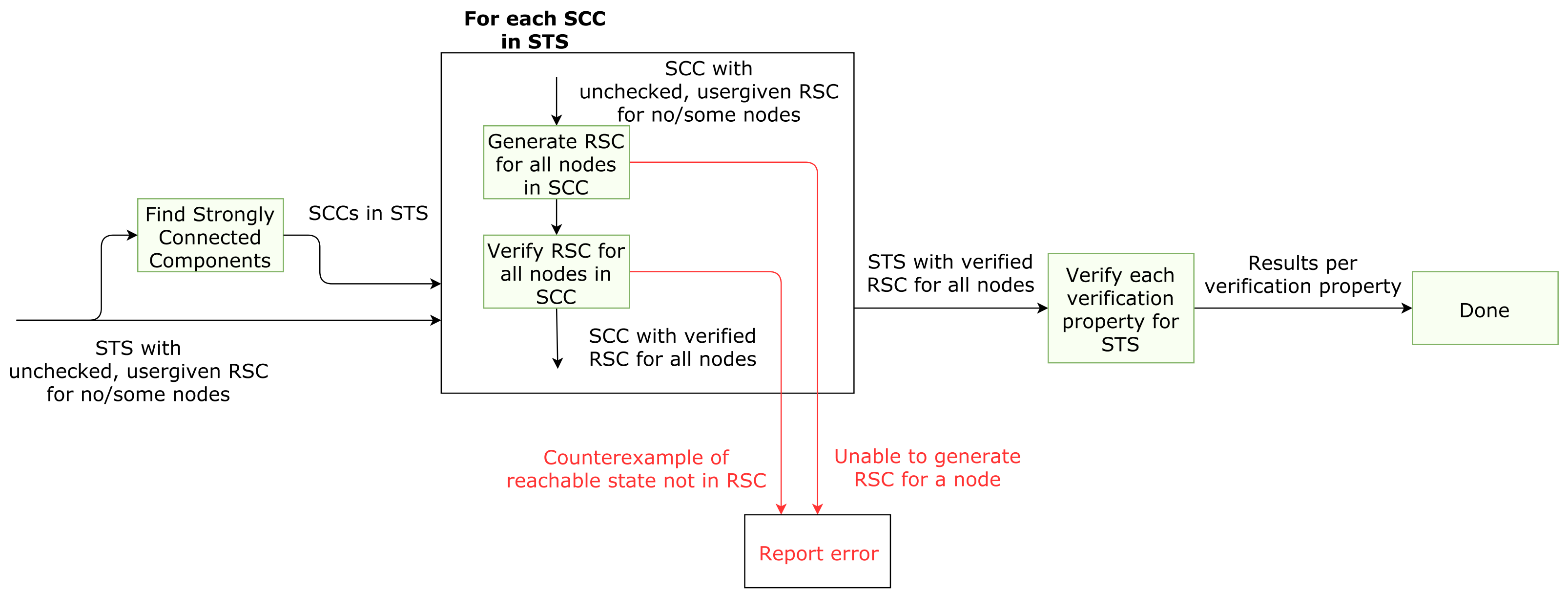

For each SCC in STS
SCC with unchecked, usergiven RSC for no/some nodes
Generate RSC for all nodes in SCC
Verify RSC for all nodes in SCC
SCC with verified RSC for all nodes
Find Strongly Connected Components
SCCs in STS
STS with unchecked, usergiven RSC for no/some nodes
STS with verified RSC for all nodes
Verify each verification property for STS
Results per verification property
Done
Counterexample of reachable state not in RSC
Unable to generate RSC for a node
Report error

Figure 4.2: Detailed overview of individual steps of the algorithm.

4.1.3 Scoping

With the procedure described in this chapter we will combine constraints. If done without care, variables of reachable state constraints, transition guards and relations referencing different state variables may overlap erroneously. We shall therefore define two scoping tactics to keep variables distinct through alpha-conversion. We shall always combine the same four building blocks into new constraints or queries: the origin RSC, transition guard, transition relation and the destination RSC. Considering a RSC for some node, we are always interested in the (free) variables representing the state variables for that node. Considering the transition guard and relation, we are always interested in the (free) variables representing the state variables of the origin and destination node and the transition variables. Also, we shall create RSCs by combining the RSCs of previous nodes and transition guards and relations. These RSCs may contain (free) variables referencing the state variables for other nodes and transition variables. This may be shown visually in figure 4.3.

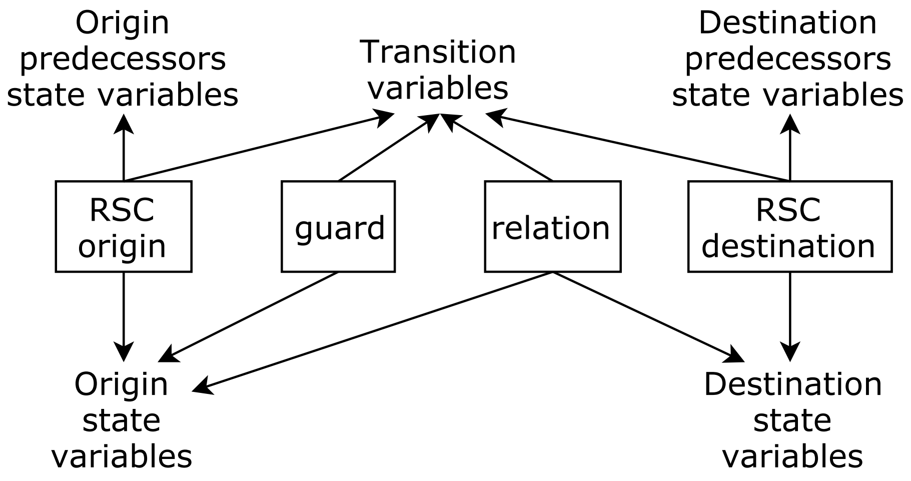

Origin predecessors state variables
Transition variables
Destination predecessors state variables
RSC origin
guard
relation
RSC destination
Origin state variables
Destination state variables

Figure 4.3: The four constraints to be used as buildings blocks (origin RSC, transition guard, transition relation and destination RSC) which we will use to create new properties. The arrows show which variable sets may be encountered in which of the constraints.

Scoping of Variables with Generating RSCs (Strategy 1)

With the definition of a STS (definition 10) there exist no RSCs for nodes and each of the transitions have guard and relations constraints which contain transition and state variables with and without the $'$ symbol. Our first scoping strategy is a naming strategy so all these variables have a label referencing the state variable of a specific node or a transition. This scoping strategy may be used when generating RSCs.

We assume a STS from definition 10. We change the names for the variables in the guards and relations of each transition. Any variable name in a guard which is a state variable label is joined with the label of the origin node. Any variable name in a guard which is not a state variable label is assumed a transition variable. This variable label is joined with the label of the transition.

The renaming of names for the relation is the same as the guard with the addition that any variable label that ends with the $'$ symbol is replaced with the variable label without the $'$ symbol and joined with the label of the destination node.

In the rest of this chapter we will assume this first scoping phase has already been performed.

Scoping of Variables with Verification Queries (Strategy 2)

The second scoping strategy is used when verifying RSCs and checking if a verification property holds. We assume the first scoping strategy has already been done so it is clear with any variable label which node or transition it references. As seen in figure 4.3, we will use RSCs from at most two nodes which may contain state variables referencing the same (predecessor) node. This overlap is erroneous so we shall discuss a naming strategy to separate all variable sets from figure 4.3.

The variable names in the origin RSC, guard, relation or destination RSC may be divided in a number of sets: 'origin predecessors state variables', 'origin state variables', 'transition variables', 'destination predecessors state variables' and 'destination state variables'. How to divide each of the variable names into any of the sets is rather straightforward.

The variable names in any of the sets are joined with a postfix referencing to which set this variable belongs. These are respectively: *prev_prev*, *prev*, *t*, *next_prev* and *next*. This naming strategy still allows transition variables in the RSC of the origin and destination to overlap. We shall see that for our queries this is not the case.

4.2 Finding Reachable State Constraints for STS

Our goal is to find reachable state constraints for all nodes in an arbitrary STS. We shall use an iterative approach where we continuously search for reachable state constraints and verify if they are valid. We are done when we are unable to verify a reachable state constraint, unable to generate a reachable state constraint for some node or when all nodes have a verified reachable state constraint. Reachable state constraints will be represented by a single constraint as shown with definition 17 in chapter 4.1.2.

With our approach we shall not only generate reachable state constraints automatically, we might also inquire the user to supply a reachable state constraint for some node. In the last case, we also have to verify if the given reachable state constraint is valid. Therefore, when considering the reachable state constraint for a node, it may be absent, unverified or verified. This leads to two different rule sets: 1) A rule set to generate reachable state constraints and 2) A rule set to verify unverified reachable state constraints.

Chapter 4.2.1 details how reachable state constraints may be deduced from previous reachable state constraints and chapter 4.2.2 details how reachable state constraints may be verified using mathematical induction over paths.

4.2.1 Generate Reachable State Constraints

Reachable state constraints represent the reachable state for some node. In order to reach some state for a node n, it depends on all previous nodes and their reachable state. From the recursive definition of reachable state (definition 14), we may see that the reachable state for some node n depends fully on the reachable state of all direct predecessor nodes N_{pred} and the transitions between n and N_{pred}. We also specify this in lemma 2.

Lemma 2. *The reachable state for some node depends only on the incoming transitions and the reachable state of the previous node.*

Deduction Rule

With lemma 2, we see how one may deduce a number of reachable state constraints. Figure 4.4 shows a diamond of nodes. The reachable state of *n1* depends fully on just the start transition. Therefore, the strongest RSC for *n1* is $i_{n1} == 0$. Using the RSC for *n1*, we are able to deduce a RSC for *n2* and *n3*; we know the RSC for the origin nodes of all incoming transitions for both nodes *n2* and *n3*. Therefore, continuing our intuïtive approach, the RSC for *n2* is the RSC of *n1* 'sequenced' with the transition between *n1* and *n2*. With sequenced, we mean to join the RSC of *n1* with the guard and relation of the transition as a conjunction. We should use conjunction as both the RSC of *n1* and the guard must hold before the transition (otherwise the transition may not be taken) and the relation must hold in *n2* after the transition. Using this intuïtive approach, we reach the RSC for *n2*: $i_{n1} == 0 \land true \land i_{n2} == i_{n1} + 2$. Analogously, the RSC for *n3* is $i_{n1} == 0 \land true \land i_{n3} > 3$. Finally, we can extend our deduction to *n4* as the disjunction of the deductions for both transitions to *n4* as either one transition may be used to reach *n4*: $[i_{n1} == 0 \land true \land i_{n2} == i_{n1} + 2 \land true \land i_{n4} == i_{n3}] \lor [i_{n1} == 0 \land true \land i_{n3} > 3 \land i_{n3} < 100 \land i_{n4} == i_{n3} - 100]$.

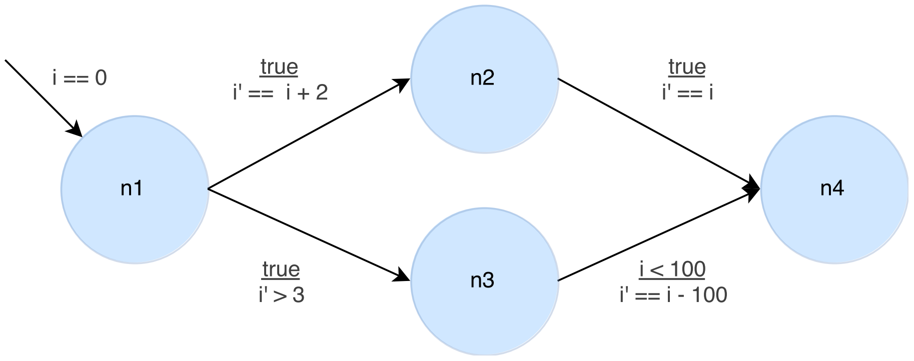


Figure 4.4: STS to show reachable state for some node depends fully on reachable state of previous nodes.
Guards are above the line, relations below the line for transitions

With this example we have shown a deduction rule to generate a RSC for some node depending fully on the RSCs of the direct predecessors and the transitions between the nodes. We may formalise this rule in definition 18.

Definition 18. ***Deduction rule.*** *Given a STS with nodes N and some node $n \in N$ where n is not in a cycle, all entering (scoped) transitions $T_{entering,n}$ and (scoped) start transitions $T_{start,n}$ with n as destination:*
$RSC_n = [\bigvee_{t \in T_{entering,n}} RSC_{t_{origin}} \wedge t_{guard} \wedge t_{relation}] \vee [\bigvee_{t \in T_{start,n}} t_{relation}]$

Theorem 1. *The deduction rule is sound.*

Proof. The deduction rule is a reformulation of the definition of reachable state (definition 14) as a property. We take n as the destination node for any start transition t_{start} and m as the origin node and n as the destination node for any transition t. RSC_n is the reachable state constraint for node n. We have to show that any ς_n should $\varsigma_n \models RSC_n$. We know from definition 14 that for any reachable state ς_n either there is a ς_m for transition t so $\varsigma_m, \varsigma_n \models \exists v_t \in V_t[C_t^{guard} \wedge C_t^{relation}]$ or there is a start transition t_{start} so $\varsigma_n \models C_{t_{start}}$. We assume RSC_m is known and therefore $\varsigma_m \models RSC_m$. Combining the statements before leads to $\varsigma_m, \varsigma_n \models [\exists v_t \in V_t[C_t^{guard} \wedge C_t^{relation}] \wedge RSC_m]$ for any transitions to n. Again either a start transition or transition may lead to n and therefore $[\bigvee_{t \in T_{entering,n}} \varsigma_{t_{origin}}, \varsigma_n \models [\exists v_t \in V_t[C_t^{guard} \wedge C_t^{relation}] \wedge RSC_{t_{origin}}]] \vee [\bigvee_{t \in T_{start,n}} \varsigma_n \models C_{t_{start}}]$. If we leave the variable assigned with ς_n, $\varsigma_{t_{origin}}$ and v_t free, we reach $[\bigvee_{t \in T_{entering,n}} [C_t^{guard} \wedge C_t^{relation} \wedge RSC_{t_{origin}}]] \vee [\bigvee_{t \in T_{start,n}} C_{t_{start}}]$ which evaluates to true for some variables assignments for $\varsigma_{t_{origin}}$ and $v_t \in V_t$ and any reachable state ς_n. Therefore if $RSC_n = [\bigvee_{t \in T_{entering,n}} RSC_{t_{origin}} \wedge t_{guard} \wedge t_{relation}] \vee [\bigvee_{t \in T_{start,n}} t_{relation}]$ then for any reachable state ς_n we know $\varsigma_n \models RSC_n$.

If n is in a cycle, we know the definition of RSC_n is recursive without end and therefore invalid. However, we assume n is not in a cycle. Due to these conclusions, this theorem must be valid.

Lemma 3. *The deduction rule is incomplete.*

Proof. n may be in a cycle and definition 18 does not define what the reachable state constraint RSC_n is for n when n is in a cycle. Therefore, this lemma is valid.

Optimization

Resulting constraint properties from our deduction rule may be optimized / flattened in some cases. The reachable state constraint for node *n2* introduces a term in the conjunction of just *true*. Any *true* in a conjunction may just be removed as $(true \wedge P) \leftrightarrow P$ (tautology). There is also the case where, from example 4.4, the variable i_{n3} in the reachable state constraint for node *n3* has no dependency on variable i_{n1}. In this case, the reachable state constraint for node *n3* may be reduced to just $i_{n3} > 3$ as variable i_{n3} has no relation to variable i_{n2} and we are interested in defining the reachable state for variable i_{n3}. These optimizations are not further explored and left to future work.

Strongest property relation

The deduction rule respects the strongest property relation. With the strongest property relation we mean that a reachable state constraint is as strong or strict as possible; only reachable states are represented by the reachable state constraint.

Definition 19. *A reachable state constraint RSC_n for some node n is strongest when for any reachable state ς_n it holds that $\varsigma_n \models RSC_n$ and for any unreachable state $\varsigma_{unreachable,n}$ it holds that $\varsigma_{unreachable,n} \models \neg RSC_n$.*

The RSCs we deduced for figure 4.4 are the strongest. Node *n1* only depends on the start transition. Therefore the strongest reachable state is represented by just the start transition. By deducing the RSC for *n2*, we actually also obtain a strongest RSC for *n2*. Any reachable state in *n1* may be transitioned to *n2* if it satisfies the guard and the new reachable state for *n2* is constrained by the relation. Our deduction rule is just the sequence of the RSC for the previous node, the guard and relation to represent all reachable states after the transition. Therefore it must respect the strongest property relation.

Lemma 4. *Our deduction rule respects the strongest property.*

Proof. Say we want to apply the deduction rule to some node n and we have all direct predecessor nodes $N_{previous}$. We have to show that the deduction rule respects the strongest property for both transitions and start transitions. Given that the RSCs for all $n_{previous} \in N_{previous}$ are the strongest RSC, the disjunction of all RSCs sequenced with the transition to n must be a strongest reachable state constraint RSC_n for n. Otherwise, there is a ς_n so $\varsigma_n \models RSC_n$ and $\neg[\varsigma_n \models RSC_{n_{previous}} \wedge t_{guard} \wedge t_{relation}]$ for some node $n_{previous} \in N_{previous}$. This is not possible by definition of the deduction rule as RSC_n contains $... \vee [RSC_{n_{previous}} \wedge t_{guard} \wedge t_{relation}] \vee ....$ Analogously, there exists no ς_n and start transition t_{start} so $\varsigma_n \models RSC_n$ but not $\varsigma_n \models t_{relation,start}$ as RSC_n contains $... \vee t_{relation,start} \vee ....$ Therefore, the deduction rule must respect the strongest property.

Incomplete

While we have shown that the deduction rule is sound and it respects the strongest property relation, we have also shown it is not complete (lemma 3; for any STS with a cycle we will not be able to deduce a RSC. This is because the RSC for the node starting the cycle is dependent on itself. As an example, look at figure 4.5 which contains a self loop on node *n1*. Our deduction rule states that we need to know the RSC for node *n1* to deduce the RSC for node *n1* as there is a transition from *n1* to *n1*. Therefore, we will not know the RSC for all direct predecessors of *n1* until we know the RSC for *n1*. This is recursion without end and it is the reason we cannot deduce a RSC for STSs with a cycle using the deduction rule.

We shall show in chapter 4.2.3 how we involve the user to find a RSC for (some) nodes in a cycle.

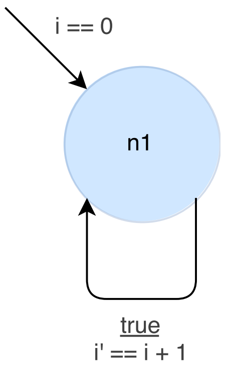


Figure 4.5: STS with a cycle.
Guards are above the line, relations below the line for transitions

Mechanize Deduction Rule

We can mechanize the deduction rule with the scala-like code of listing 4.1. It assumes:

- All variables of all reachable state constraints and transitions are scoped using scoping strategy 1.
- RSCs for nodes with a transition to `node` are known.

The result is that the RSC for `node` is deduced and set. It is correct and verified by design.

Listing 4.1: Scala-like code for mechanizing deduction rule

```
function deduce(Node node, STS sts) {
    ingoingSequences = node.entering.map(
        (t) => And(t.guard, t.relation, t.origin.rsc)
    )
    startTransitions = sts.startTransitions.filter(
        (s) => s.destination.equals(node)
    )
    ingoingSequences ++= startTransitions.map((s) => s.relation)

    newReachableStateConstraint = ingoingSequences.size match {
      case 0 => error // Should not happen. Disconnected node
      case 1 => ingoingSequences.head
      case _ => ingoingSequences.drop(1).foldLeft(
          ingoingSequences.head,
          (result, e1) => Or(result, e1)
      )
    }

    node.rsc = newReachableStateConstraint
}
```

4.2.2 Verify Reachable State Constraints

In some cases, a RSC will be proposed and we need some mechanism to verify if the property represents at least all reachable states for that node. We propose an incremental verification rule which is based on the triple of each originating node, transition and destination node for each transition. This coincides with a single path segment. We shall show that it is enough to verify a RSC for some node by verifying all triples of origin nodes and transitions to this node. In order to verify a RSC for some node, we have to show that the states after all paths to this node are models of the RSC.

Verification Rule

Corollary 2 shows that after any path to some node, the reachable state constraint (RSC) for that node holds. A proof is not necessary as this corollary is a reformulation of the definition of reachable state constraint (definition 17).

Corollary 2. *Given a reachable state constraint RSC_n for some node n, a state ς_n so $\varsigma_n \models RSC_n$ is the result of all possible paths Π to n.*

The theoretical value of this corollary may be explained using figure 4.6. Say we are interested in verifying a reachable state constraint RSC_{n3} for node *n3*. There are a number of nodes $N_{previous}$ which have a transition to *n3* including nodes *n1* and *n2* and assume we know a (unverified) reachable state constraint for those nodes. Any path to *n3* surely contains a node from $N_{previous}$ as the node just before the transition to *n3*. Now, corollary 2 allows us to summarize the resultant state $\varsigma_{n_{previous}}$ of all possible paths to a predecessor $n_{previous} \in N_{previous}$ of *n3* as the reachable state constraints for that node. Now to check if RSC_{n3} is valid for all reachable states ς_{n3}, we only have to check if the RSC_{n3} is implied after the last transition from a direct predecessor; in other words $[RSC_{n_{previous}} \wedge t_{guard,n_{previous},n3} \wedge t_{relation,n_{previous},n3}] \rightarrow RSC_{n3}$. This shows that to verify the reachable state constraint for some node *n3*, we have to show that the reachable state constraints of all previous nodes sequenced with the transition maintains the reachable state constraint to verify.

Definition 20. ***Triple verification rule.*** *To verify a property RSC_n is a reachable state constraint for some node n, we must show for all previous nodes $n_{previous} \in N_{previous}$ and any transition t from $n_{previous}$ to n with guard t_{guard} and relation $t_{relation}$ that $[RSC_{n_{previous}} \wedge t_{guard} \wedge t_{relation}] \rightarrow RSC_n$. Also show that for any start transitions $t_{start,n} \in T_{start}$ to n that $t_{relation,start,n} \rightarrow RSC_n$. Also show that $RSC_{n_{previous}}$ is a reachable state constraint for node $n_{previous}$.*

Theorem 2. *The triple verification rule is sound*

proof. We have to prove that when our theorem holds, any path to a node results in a state which is a model of the reachable state constraint for that node. We shall use proof by mathematical induction over the path. Therefore we need to provide the base case proof and the induction step proof.

- **Base case:** There is no path so we start at a start transition t_{start} with destination n. For the reachable state constraint RSC_n it should hold that $\varsigma_n \models RSC_n$ for any state ς_n for which $\varsigma_n \models C_{t_{start}}$. This is checked directly by the theorem. Therefore the base case holds.

- **Induction step:** Assume we are in a node n_{i-1} with a transition t to node n_i and a path P_{i-1} to node n_{i-1} with a resultant state $\varsigma_{n_{i-1}}$ which $\varsigma_{n_{i-1}} \models RSC_{n_{i-1}}$. With the transition for any state ς_{n_i} and some reachable state $\varsigma_{n_{i-1}}$ it should hold that $\varsigma_{n_{i-1}}, \varsigma_{n_i} \models [RSC_{n_{i-1}} \wedge C_t^{guard} \wedge C_t^{relation}]$. We are proving that for any reachable state ς_{n_i} it holds that $\varsigma_{n_i} \models RSC_n$. Therefore, we know this is true if there is an assignment $\varsigma_{n_{i-1}}$ so $[RSC_{n_{i-1}} \wedge C_t^{guard} \wedge C_t^{relation}] \rightarrow RSC_n$. This is checked by the triple verification rule. Therefore, the induction step holds.

We have shown that our theorem holds as the base case and induction step of our mathematical induction over path proof holds.

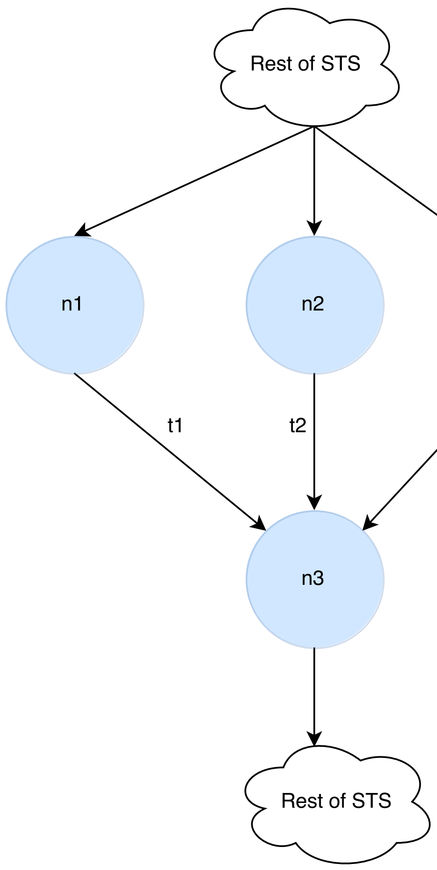


Figure 4.6: Figure to show a number of nodes (including *n1* and *n2*) have a transition to node $n3$ as a part of some STS.

Strongly Connected Components

The verification of reachable state constraints for STS nodes in general is dependent on the previous nodes as is stated by lemma 2. When cycles exist in a STS this causes cyclic dependencies to verify the RSCs of the nodes in the cycle as with the triple verification rule we know a RSC is only verified to be correct when the previous RSCs are verified to be correct; similar to the cyclic dependencies due to which we cannot use the deduction rule for nodes in a cycle. To resolve this, we propose to verify the reachable state constraints per strongly connected component (SCC) instead of verifying it per individual nodes. A SCC contains all nodes which are reachable from each other or in other words, the RSC of one node in the SCC may depend on the RSC of another node in the SCC. Any nodes outside of a SCC is only reachable one-way and there is no bidirectional dependency on the reachable state. The condensation graph of the STS shows which SCCs are dependent on each other. The condensation graph is acyclic and it shows an ordering of verifying the RCSs of the nodes in the SCCs. If we follow the condensation graph and only verify the nodes in the SCC when the RCSs of the nodes of the previous SCCs are verified, we will know that all RCSs of previous nodes are verified for correctness as assumed by the triple verification rule. This is formalised in lemma 5.

Lemma 5. *Verification of reachable state constraints for nodes in a STS must be done per strongly connected component.*

Incomplete & Spurious Counterexamples

While the triple verification rule may serve as a basis to verify reachable state constraints, it is not complete. The verification rule is based on mathematical induction; it assumes the correctness of previous reachable state constraints to prove the next one. An example where more information is needed is given by Wahl [27]. Wahl shows that a property for the fibonacci sequence may be proven using k-induction [15]. k-induction is an iterative proof form where we assume previous results respect some property and we have to show that subsequent results still respect the property. The k is the amount of previous answers which are used to proof the property. Wahl used the property $a > n$ as an example where a is the n'th fibonacci number (only when $n > 5$). Wahl used this example to show 2-induction is needed and 1-induction is not enough to verify the property. We can model the fibonacci sequence into a STS with the appropriate start state and use the property $a > n$ as the reachable state constraint. The triple verification rule would not be enough to verify this reachable state constraint.

When a SMT-solver is used to check this property with the triple verification rule, a counterexample is produced. This counterexample is a spurious counterexample due to the mathematical induction proof failing. We know it is spurious as the property is proven by Wahl. This shows that we do not know if a counterexample given by the triple verification rule is actually reachable by the STS or it is because mathematical induction is not enough to proof the reachable state constraint.

Depending on the SMT-solver, it is also possible for a SMT-solver to return unknown as we have described in chapter 2.3. In this case, we are unable to say what went wrong.

Mechanize Triple Verification Rule

We use lemma 5 to mechanize the triple verification rule with the scala-like code of listing 4.2. It assumes:

- Each of the direct predecessor nodes in the SCC have a RSC.
- Each of the nodes in the SCC have a RSC.
- The condensation of the strongly connected components is known and used as the order to verify each strongly connected component.

In order for our query to not have any erroneous variable overlap we have to scope the variables of the origin RSC, guard, relation and destination RSC accordingly using scoping strategy 2. This is done on lines 18 to 23. The result of the algorithm is that all reachable state constraints for the nodes in the SCC are verified or a (spurious) counterexample is given.

Listing 4.2: Scala-like code for mechanizing triple verification rule

```
1  function verify(StronglyConnectedComponent scc, STS sts) {
2      for(node <- scc.nodes) {
3          counterexamples: Set[Counterexample] = Set()
4          startTransitions = sts.startTransitions.filter(
5            (s) => s.destination.equals(node)
6          )
7
8          // Check each start transition
9          for(trans <- startTransitions) {
10           counterexample = SMTSolver.isUnsat(
11              trans.relation,
12              Not(start.destination.rsc)
13           )
14
15           counterexamples += counterexample
16         }
17
18         // Check each transition to this node
19         for (trans <- node.entering) {
20             from = trans.origin
21             counterexample = SMTSolver.isUnsat(
22               scopeFromRSC(from.rsc),
23               scopeGuard(trans.guard),
24               scopeRelation(trans.relation),
25               ForAll( to.rsc.predecessorVariables ++ to.rsc.transitionVariables
26                     , Not(to.rsc)
27                     )
28               )
29
30             counterexamples += counterexample
31         }
32
33         if(counterexamples.nonEmpty) {
34             throw new CounterexampleException(node, counterexamples)
35         }
36     }
37
38      for(node <- scc.nodes) {
39        node.rsc.get.status = Verified
40      }
41  }
```

4.2.3 Combining Generating and Verifying Reachable State Constraints

So far we have created the deduction and triple verification rule to generate and verify reachable state constraints for nodes in an arbitrary STS. We have proven their soundness and seen that they are both incomplete. We shall show how to combine them into one algorithm.

Due to lemma 2 we have seen that both generation and verification of reachable state constraints depend solely on the RSCs of predecessor nodes. For verification we therefore have to verify RSCs for nodes per strongly connected component (SCC) and not per individual node in order to guarantee that a RSC is verified in general. We propose to use the condensation of the SCCs for a STS be the order for which nodes we generate and of which nodes we verify the RSC. A correct ordering would be all nodes per SCC when doing a breadth-first search (BFS) [12] on the condensation graph.

This results in the scala-like code of listing 4.3. It assumes:

- All guards and relations of transitions are scoped using scoping strategy 1.

The listing shows the BFS over the condensation graph of the STS from line 7. We keep a queue `pending` which are the strongly connected components to be processed. We choose to process the SCCs in FIFO-ordering by choosing the first SCC in the queue on line 8. We also remove this SCC from the `pending` queue on line 9.

For the chosen SCC we first generate all missing RSCs for the nodes of the SCC on lines 12 through 29. We find a node for which all predecessor nodes have a RSC on line 13. If such a node exists, we deduce the RSC on line 16. If no such node exists but there are nodes without a RSC, we know there exists at least one node which does not have a RSC and where there are predecessors also missing a RSC. These nodes are found on lines 21 to 25 and an exception is thrown on line 27 to ask the user to provide a RSC for one or more of the nodes for which no RSC could be generated.

Assuming each node in the SCC has a RSC, we continue to lines 31 to 41. On line 32 we use the triple verification rule to verify all RSCs on the current SCC. We then add the SCC to the processed SCCs on line 34. Finally, we add all SCCs for which all predecessor SCCs are all `processed` to `pending` on lines 37 to 41.

Listing 4.3: Scala-like code for mechanizing generation and verification of reachable state constraints.

```
1  function generateAndVerify(STS sts) {
2      sccCondensation = SCCCondensation.findSCCCondensation(sts)
```

```
    pending = sccCondensation.findIndependentSCCs()
    processed = Set()

    // BFS over condensation
    while(pending.nonEmpty) {
      currentSCC = pending.head
      pending -= currentSCC

      // Generate
      while(currentSCC.nodes.exists((n) => n.rsc.isEmpty)) {
        val nodeToGenerate = DeductionRule.findNext(currentSCC)

        if(nodeToGenerate.nonEmpty) {
          DeductionRule.deduce(nodeToGenerate.get, sts)
        } else {
          // Find nodes which does not have a RSC and for which atleast
          // one previous node has RSC or a start transition
          nodesForUserInput = scc.nodes.filter(
            (n) =>
              n.rsc.isEmpty &&
              (n.entering.map((t) => t.origin).exists((n_) => n_.rsc.nonEmpty) ||
               sts.startTransitions.map((s) => s.destination).contains(n)
              )
          )
          throw new UserInputException(nodesForUserInput)
        }
      }

      // Verify
      TripleVerificationRule.verify(currentSCC, sts)

      processed += currentSCC

      // Find next SCC to add to pending
      for(scc <- currentSCC.next) {
        if(scc.previous.forall((prev) => processed.contains(prev))) {
          pending += scc
        }
      }
    }
  }
```

This algorithm assumes that all variables on transitions and user-given reachable state constraints are scoped using scoping strategy 1. The result is either a (spurious) counterexample for some RSC, a request for the user to provide a RSC for some node(s) or a STS where each node contains a verified RSC.

Introducing New Rules

So far we have implemented the deduction and triple verification rule which are not complete. Other generation and verification rules may be added to allow for more RSCs to be automatically generated and more RSCs to be verified. For example, Wahl [27] has shown that k-induction is a more powerful proving-technique than mathematical induction and we have shown that Wahl's example may be encoded in a STS for which our triple verification rule will have a spurious counterexample. We might create a verification rule using k-induction to prove RSCs for nodes in a cycle.

These new generation and verification rules may be freely added to our algorithm in listing 4.3. We have shown which assumptions and considerations have to be taken into account when creating and adding new rules in respectively chapters 4.2.1 and 4.2.2.

Multithreading

We might alter the algorithm of listing 4.3 to add multithreading. The `pending` set on line 3 contains all SCCs which may be processed. This set may grow to more than one SCC. Each SCC may be a job for some thread pool so each SCC is processed in parallel. This is allowed as the condensation of the SCCs shows the dependency between SCCs as is given by lemma 2. For all SCCs in pending, all previous SCCs are already processed and therefore we may process these independently. We leave this research to future work.

Soundness

The algorithm of listing 4.3 is sound. It is based on the deduction and triple verification which we already have deemed sound. We use an ordering of generation and verification of RSCs for nodes per SCC. While this changes the order of evaluation per node, this does not effect soundness. What does effect soundness is if the necessary assumptions for each rule are fulfilled.

Proof. What we need to prove is if the combination of the deduction and verification rule to the algorithm listed in listing 4.3 is sound. In order to prove this algorithm sound, we have to show that each of the assumptions of each rule are fulfilled. Therefore:

- Deduction rule assumptions
  - **All variables of all RSCs and transitions are scoped using strategy 1:** This is also an assumption of the combined algorithm and is therefore fulfilled.
  - **RSCs for direct predecessor nodes must be known:** A node is chosen where for each predecessor node a RSC is known on line 13. This chosen node is used directly by the deduction rule and therefore this assumption is fulfilled.
- Triple verification rule assumptions
  - **Each of the nodes in the SCC have a RSC:** We know this assumption is fulfilled as we generate RSCs for empty nodes until every node in the SCC has a RSC (line 12).

- **Each of the direct predecessor nodes in the SCC have a RSC:** A direct predecessor of a node in the SCC is either in the SCC or a predecessor SCC. If the predecessor node is in the SCC, see the previous assumption as it is fulfilled. We know that the RSCs of the nodes in the predecessor SCCs have been generated and verified. Therefore, if the predecessor node is in a predecessor SCC we know it also has a RSC. Therefore, this assumption is also fulfilled.
- **Verification of RSC per node must be done per SCC:** The SCCs the condensation graph is found on line 2. The RSCs of the nodes are generated and verified per SCC. The order of each SCC is chosen using BFS where we only add a SCC to `pending` if we know each of the direct predecessor SCCs have been `processed` (lines 37 to 40). Therefore, we know each of the predecessor SCCs have been processed and we follow the dependencies of the condensation graph. Therefore this assumption is fulfilled.

Each of the assumptions of the used algorithms are fulfilled. Therefore, we know the combined algorithm is also sound.

Completeness

The combined algorithm is able to generate and verify RSCs for nodes in a STS. It is not able to generate a RSC or verify a RSC in general as we have shown in chapters 4.2.1 and 4.2.2. However, we are able to use this combined algorithm for a STS with an arbitrary graph structure.

Proof. We have to proof that the combined algorithm of listing 4.3 may be used on a arbitrary graph structure and will always have an answer. The algorithm combines the deduction rule, triple verification rule, Gabow's path-based SCC finding algorithm and BFS. Neither the deduction or triple verification rules have a restriction on the graph structure of the STS and will always some answer. Finding the condensation graph of a STS as well as BFS is complete [17] [12]. There are no other features of the algorithm. Therefore, we may conclude that the combined algorithm always returns an answer regardless of the graph structure of a STS and it is therefore complete in this regard.

Termination

The combined algorithm may not terminate in general. We know our ordering through the condensation graph is finite as the condensation graph is acyclic. We also know the deduction rule and verification terminate when considering the algorithms of listings 4.1 and 4.2. They contain iterators only over finite collections and must therefore terminate. However, our algorithm accepts non-linear arithmetic and is dependent on a SMT-solver. Depending on this SMT-solver the algorithm will either terminate in general or not. For our prototype we have chosen the SMT-solver Z3 and we know it does not terminate in all cases when accepting non-linear arithmetic [14].

4.3 Verifying Verification Properties

Part of our task is to find useful properties to verify based on a STS specification. Safety guarantees (something that will never happen) and liveness guarantees (something that will (eventually) happen) are two of the main types of properties which are found throughout the literature [5]. While we recognize many different properties are useful to verify, we have chosen to focus on five for this research:

1. **Safety properties** To verify states violating a safety property are not reachable (see definition 21) in the specification (Safety guarantee).
2. **Dead transitions** To verify there exists a reachable state in the origin node of every transition which satisfies the guard of that transition. A transition that can never be taken will be referred to as a dead transition. (Reachability guarantee)
3. **Sinkholes** To verify there for every reachable state for some node there is is at least one transition which may be taken if a leaving transition exists. A state reachable in a node for which no transitions may be taken will be referred to as a sinkhole. (Liveness guarantee)
4. **Unsatisfiable Start Transitions** To verify if there exists a state for the relation of each start transition. If the relation is unsatisfiable the start transition may never be taken. (Reachability guarantee)
5. **Unsatisfiable Relation** To verify that for each reachable state that satisfies the guard of a leaving transition there is at least one reachable state in the destination node which together satisfy the relation of that transition. If there is not such a reachable state then there exists a reachable state for which a transition may be taken but there is no reachable state related to in the destination node. This is erroneous and should not happen. (Liveness guarantee)

These five properties together allow a specification to be proven to never violate certain properties (safety property), to show all transitions may be used (no unsatisfiable start transitions and dead transitions) and to show that it is possible to always make progress (no sinkholes and no unsatisfiable relations).

We will formalise the five verification properties in the context of STSs in first-order logic and show how they may be checked using a SMT-solver. In order to describe the translation from the formal specification of the properties to a SMT query, we first assume that the first part of our approach was successful; we know the reachable state constraint for each node in the STS.

The reachable state constraint might be an over-approximation and this has consequences for the validity of the answer when checking the five verification properties. In some cases this might result into false negatives (an issue exists but is not found) or false positives (an issue is found which is not an issue). We shall show these consequences to the soundness of our checks.

Throughout this chapter we shall use the STS in figure 4.7 as an example.

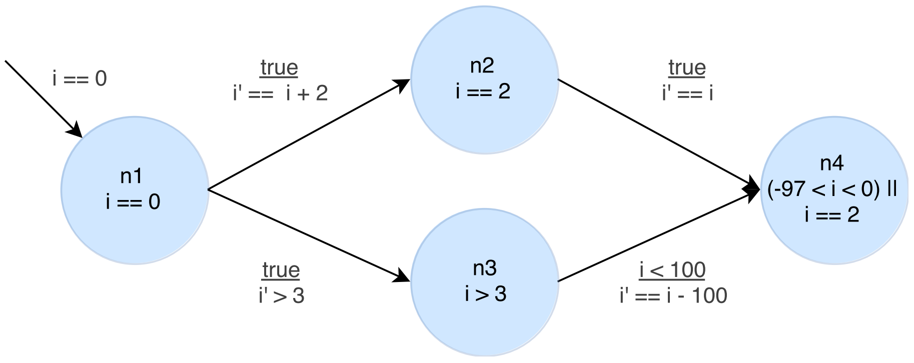


Figure 4.7: STS where RSCs are included in the labels. Used as a running example to explain the verification properties.
Guards are above the line, relations below the line for transitions

4.3.1 Safety Property

A safety property is a property modelling states which should not be reachable. A safety property for the bank account example in figure 1.1 would be to check if the balance is above zero. This safety property should hold for all reachable states in all nodes of the specification. If there is a state reachable in some node which violates/falsifies the property then we have found a counterexample.

Definition 21. *For a STS with state variables V a safety property P_{safe} with free variables $V_{safe} \subseteq V$ holds if for all reachable states ς_n $\varsigma_n \models P_{safe}$.*

Checking a Safety Property as a SMT Query

Proving there exists no reachable state which violates a safety property is to show a safety property holds. As we have a set of RSCs for each node which describe all reachable states, we can use the RSCs to show none of them violate any safety properties. In other words, each RSC should imply the safety property. This is formalised in corollary 3.

Corollary 3. *For some STS a safety property P_{safe} holds if for each reachable state constraints RSC of a node in the STS $RSC \rightarrow P_{safe}$ is valid.*

We have shown in chapter 2.3.3 that to prove the validity of some formula with a SMT-solver we have to show the abscence of falsifying models. Therefore, we have to check the negation of the formula in corollary 3 and to see if this formula is unsatisfiable. We rewrite the negation of the formula in corollary 3 such that:

$$\begin{array}{ll}
\neg[RSC \rightarrow P_{safe}] & \\
\neg[\neg RSC \vee P_{safe}] & \textit{(Rewrite implication)} \\
\neg\neg RSC \wedge \neg P_{safe} & \textit{(De Morgan)} \\
RSC \wedge \neg P_{safe} & \textit{(Double negative)}
\end{array}$$

Now we are able to write the definitive SMT query as corollary 4.

Corollary 4. *For some STS a safety property P_{safe} holds if for all reachable state constraints RSC $RSC \land \neg P_{safe}$ is unsatisfiable.*

Say we want to prove the safety property $i == 0 \lor i == 2$ as a safety property for the STS in figure 4.7. For the example, the RSCs are given as the label of the nodes. Our example has 4 nodes. We would send the SMT-solver 4 queries:

1. Node *n1*: $\neg(i == 0 \lor i == 2) \land (i == 0)$
2. Node *n2*: $\neg(i == 0 \lor i == 2) \land (i == 2)$
3. Node *n3*: $\neg(i == 0 \lor i == 2) \land (i > 3)$
4. Node *n4*: $\neg(i == 0 \lor i == 2) \land ((-97 < i \land i < 0) \lor i == 2)$

In this case, the safety property does not hold. While the queries for nodes n1 and n2 are unsatisfiable, the queries for nodes n3 and n4 both have counterexamples. Examples of counterexamples are respectively: $\varsigma_{n3}[4 \leftarrow i]$ and $\varsigma_{n4}[-1 \leftarrow i]$.

Soundness with over-approximated RSCs

We have defined the concept of a safety property and have shown how to check a safety property using the RSCs of each node and a SMT-solver. The RSCs may be over-approximations and this has consequences for the soundness of our method.

That a RSC RSC_n is an over-approximation means there may exist a state ς_n so $\varsigma_n \models RSC_n$ but ς_n is not reachable. We check if $RSC \land \neg P_{safe}$ is unsatisfiable for some safety property P_{safe}. However, there may be a state $\varsigma_{falsepositive}$ so $\varsigma_{falsepositive} \models RSC$ but $\varsigma_{falsepositive} \models \neg P_{safe}$ and $\varsigma_{falsepositive}$ is unreachable. In other words, there may exist a state which is not reachable but is a counterexample for the safety property and is a state modelled by the RSC. This is considered a false positive and may happen when the RSC may be an over-approximation.

In chapter 4.2.1 we have shown that the deduction rule respects the strongest property. Therefore, we know that for any STS where only the deduction rule is used to find all RSCs that none of the RSCs are over-approximations. In this case, there may be no false positives and any counterexample is valid.

While a RSC may be an over-approximation, it may not be an under-approximation. Therefore, at least all reachable states are checked if they violate the safety property and no false negatives are reported.

4.3.2 Dead Transitions

Dead transitions are transitions which may never be taken. There exists no reachable state in the origin node which satisfies the guard. Dead transitions are indications that there exists a part of the specification which may have some semantic meaning but there is no reachable state to take the transition there.

Definition 22. *Given a STS with transitions T and some transition $t \in T$ then t is a dead transition if for all reachable state $\varsigma_{t_{origin}}$ in node t_{origin} $\varsigma_{t_{origin}} \models \neg t_{guard}$.*

Checking for Dead Transitions as a SMT Query

Proving there is a dead transition is to show that for each reachable state for the origin node of the transition it does not satisfy the guard. We have a set of RSCs per node which model all reachable states. With the RSC of the origin node of a transition we can check if there is a reachable state which satisfies the guard. In other words, we are checking if there exists a state which satisfies the RSC of the origin node and the guard of the transition.

Definition 23. *Given a transition t with guard t_{guard} and the reachable state constraint $RSC_{t_{origin}}$ for the origin node t_{origin}, t is considered a dead transition if $RSC_{t_{origin}} \wedge t_{guard}$ is unsatisfiable.*

We can use a SMT-solver to directly check this query if it is unsatisfiable.

In the case of figure 4.7, we have 4 outgoing transitions. We will focus on the transition from *n1* to *n2*. We would send the SMT-solver a query checking is that transition is a dead transition:

1. $i == 0 \wedge true$

The answer to our query would be satisfiable (namely $[0 \leftarrow i]$). This would make this specific transition not a dead transition as the reachable state $\varsigma_n[0 \leftarrow i]$ would allow the transition to be taken.

Soundness with over-approximated RSCs

We have defined the concept of dead transitions and how to check if a given transition is a dead transition. However, as with the other verification properties there are consequences for the fact that a RSC may be an over-approximation. There may be a state $\varsigma_{unreachable}$ so $\varsigma_{unreachable} \models RSC$, $\varsigma_{unreachable} \models t_{guard}$ and $\varsigma_{unreachable}$ is unreachable. In other words, there may exist a state which is not reachable but is a state satisfying the guard and the RSC of a given origin node and outgoing transition. This is considered a false negative and may happen when the RSC is an over-approximation.

As with chapter 4.3.1, we note that the deduction rule respects the strongest relation and if only the deduction rule is used to find all RSCs then all RSCs are not over-approximations. In this case there can be no false negatives.

A RSC may not be an under-approximation and therefore there can be no false positives. There can be no transition checked as a dead transition which is not a dead transition because there are no reachable states missing in the RSC so all reachable states included in the RSC do not satisfy the guard of the transition. In this case the missing reachable states would satisfy the guard of the transition which shows it is not a dead transition. However, the RSC may not be an under-approximation so there may exist no false positives with the check of dead transitions.

4.3.3 Sinkholes

Sinkholes are reachable states in a node which do not satisfy the guard of any of the outgoing transitions. Effectively, when you reach this state you are stuck when there are outgoing transitions. A sinkhole is an indication that there is a state reachable you cannot leave and might indicate an issue with the specification.

Definition 24. *Given some node $n \in N$ for a STS with nodes N and transitions T and transitions $T_n \subseteq T$ where $\forall t \in T_n.t_{origin} = n$ then some reachable state ς_n is a sinkhole if $\varsigma_n \models \bigwedge_{t \in T_n}[\forall v_{t,1}, v_{t,2}... \neg t_{guard}]$ where $v_{t,i}$ are the transition variables for t_{guard}.*

Checking for Sinkholes as a SMT Query

Proving there is a sinkhole is to show that there is a reachable state for which no transitions may be taken. We have a set of RSCs per node which model all reachable states for some node. We can check with the RSCs if there is a reachable state so the property in definition 24 holds.

Definition 25. *Given some node $n \in N$ with RSC RSC_n for a STS with nodes N and transitions T and transitions $T_n \subseteq T$ where $\forall t \in T_n.t_{origin} = n$ then there exists a sinkhole if $RSC_n \wedge \bigwedge_{t \in T_n}[\forall v_{t,1}, v_{t,2}... \neg t_{guard}]$ is satisfiable where $v_{t,i}$ are the transition variables for t_{guard}.*

An SMT-solver is able to check the query in definition 25 directly if it is satisfiable. If it is satisfiable, the SMT-solver will show an example of a model/reachable state which is a sinkhole as the SMT-solver existentially closes the query which are the state variables of the RSC. If it is unsatisfiable, we know there is no sinkhole in that node.

As an example, we shall use figure 4.7 and check if node *n3* has a sinkhole. For this node the RSC is $i > 3$. There is a single outgoing transition from *n3* to *n4* with the guard $i < 100$. We would then query the SMT solver:

1. $i > 3 \wedge \neg[i < 100]$

This query has a model which is the reachable state $\varsigma_{n3}[100 \leftarrow i]$. This reachable state is therefore a sinkhole.

Soundness with over-approximated RSCs

We have defined the concept of sinkholes and how to check if some node has a reachable state which is a sinkhole. However, as with the other verification properties there are consequences when a RSC is an over-approximation.

There may be a state $\varsigma_{unreachable}$ so $\varsigma_{unreachable} \models RSC$, $\varsigma_{unreachable} \models \bigwedge_{t \in T_n}[\forall v_{t,1}, v_{t,2}... \neg t_{guard}]$ and $\varsigma_{unreachable}$ is unreachable. In other words, there may exist a state which is not reachable but is a state which never satisfies a guard of any outgoing transition but does satisfy the RSC. This is considered a false positive and may happen when the RSC is an over-approximation.

As with chapter 4.3.1, we note that the deduction rule respects the strongest relation and if only the deduction rule is used to find all RSCs then all RSCs are not over-approximations. In this case there can be no false positives.

A RSC may not be an under-approximation and therefore there can be no false negatives. A false negative would be a reachable state which is a sinkhole but is not found by our sinkhole check. This is only possible when the RSC is a under-approximation and the SMT-solver is not able to verify if the missing reachable state is a sinkhole or not. As the RSC is not an under-approximation, we know that this is impossible.

4.3.4 Unsatisfiable Start Transitions

Unsatisfiable Start Transitions are the relations of start transitions which are unsatisfiable. There is not a state which satisfies the relation of a unsatisfiable start transition. This is an indication of a erroneous specified start transition as it doesn't add anything to the specification semantically.

Definition 26. *A start transition t_{start} is a unsatisfiable start transition if $t_{relation,start}$ is unsatisfiable*

Checking for Unsatisfiable Start Transitions as a SMT Query

Proving a start transition is an unsatisfiable start transition is straightforward. Definition 26 may be used without further rewriting with an SMT-solver as a SMT-solver is able to check if the relation is unsatisfiable or not.

As an example, we shall use figure 4.7. There is a single start transition with the relation $i == 0$. We would then query the SMT solver:

1. $i == 0$

This query has the model/reachable state $\varsigma_{n1}[0 \leftarrow i]$ and is therefore satisfiable. This start transition is therefore not a unsatisfiable start transition.

Soundness with over-approximated RSCs

The check for unsatisfiable start transitions does not use the RSCs of any nodes. Therefore, the approach is sound even if the RSCs are over-approximations.

4.3.5 Unsatisfiable Relations

Unsatisfiable relations are transition relations where, for certain reachable states from the origin node, there are no related reachable states in the destination node. In other words, there are no destination reachable states that, together with the origin reachable state which satisfies the guard, satisfy the transition relation. This is erroneous and should not happen.

Definition 27. *A transition t has a unsatisfiable relation if there exists a reachable state $\varsigma_{t_{origin}}$ so $\varsigma_{t_{origin}} \models t_{guard}$ but there exists no reachable state $\varsigma_{t_{destination}}$ so $\varsigma_{t_{origin}} \wedge \varsigma_{t_{destination}} \models t_{guard} \wedge t_{relation}$*

Checking for Unsatisfiable Relations as a SMT Query

Proving the relation of a transition may be unsatisfiable is to find a reachable state in the origin node which satisfies the guard but the relation is unsatisfiable so there is no reachable state in the destination node related. In order to check this with an SMT-solver, we want the SMT-solver to find a reachable state based on the RSC of the origin node which satisfies the guard but falsifies the relation in all cases. With in all cases we mean we also have to properly use the transition and states variables of the transition and the destination node.

With an unsatisfiable relation, we are interested in a state of the origin RSC and some assignment of the transition variables so the guard is satisfied. The assignment of the transition variables also have to be applied to the relation part of the query. Therefore, we let the SMT-solver existentially close the transition variables. The relation contains the state variables for the destination node and we have to show that for all possible destination states it falsifies the relation. Therefore, we have to quantify the state variables of the destination node using a universal quantifier. All in all we reach the SMT query of definition 28.

Definition 28. *A transition t has a unsatisfiable relation if there exists an assignment of state variables for node t_{origin} and transition variables of t so $RSC_{t_{origin}} \wedge t_{guard} \wedge \forall v_{d,1}, v_{d,2}...\neg t_{relation}$ where $v_{d,i}$ are the state variables for the node $t_{destination}$.*

If a model exists for the query in definition 28 then t has an unsatisfiable relation and the model shows for which reachable state in the origin node and assignment of transition variables. If no model exists for the query then transition t does not have an unsatisfiable relation.

In order to perform this query, we use scoping strategy 2 as opposed to the other verification properties.

For an example we shall use figure 4.7. We shall check if the transition from *n1* to *n2* has an unsatisfiable relation. Node *n1* has RSC *i_n1_prev* $==0$ and the transition has the guard *true* and the relation *i_n2_next* $==$ *i_n1_prev* $+2$. The query to the SMT-solver becomes:

1. *i_n1_prev* $== 0 \wedge$ *true* $\wedge \forall$*i_n2_next*.$\neg[$*i_n2_next* $==$ *i_n1_prev* $+ 2]$

The query returns unsatisfiable although the reasoning why is not straightforward. There is only one reachable state for node *n1* namely $\varsigma_{n1}[0 \leftarrow i]$. This satisfies the RSC of *n1* and the guard. However, there is a reachable state for node *n2* so the relation is satisfied, namely $\varsigma_{n2}[2 \leftarrow i]$. This means in the query there is an assignment so $\neg t_{relation}$ does not hold. Therefore the universal quantifier evaluates to false and we deem the relation satisfiable for all reachable states in the origin node and all assignments for the transition variables. Therefore, this transition does not have an unsatisfiable relation.

Soundness with over-approximated RSCs

We have defined the concept of unsatisfiable relations and how to check if some transition has an unsatisfiable relation. However, as with the other verification properties there are consequences when a RSC is an over-approximation.

There may be a state $\varsigma_{unreachable}$ so $\varsigma_{unreachable} \models RSC \wedge t_{guard}$, $\varsigma_{unreachable} \models \forall v_{d,1}, v_{d,2}...\neg t_{relation}$ and $\varsigma_{unreachable}$ is unreachable. In other words, there may exist a state which is not reachable, satisfies the guard but there is not a reachable state in the destination node so the relation is satisfied. This is considered a false positive and may happen when the RSC is an over-approximation.

As with chapter 4.3.1, we note that the deduction rule respects the strongest relation and if only the deduction rule is used to find all RSCs then all RSCs are not over-approximations. In this case there can be no false positives.

A RSC may not be an under-approximation and therefore there can be no false negatives. A false negative in this case would be a reachable state in the origin node which satisfies the query but does not satisfy the RSC of origin node. In this case, the RSC would be an under-approximation as reachable states are not modelled by the RSC. However, this is not possible as the RSC must not be an under-approximation.

Table 4.1: Overview of soundness and completeness for the different steps of the verification procedure

Step	**Sound**	**Complete**
Generation & verification of reachable state constraints	Generation of RSCs is sound. When verifying if RSCs are valid, triple verification rule may give spurious counterexamples.	Complete for STS graph structure. Incomplete for generation and verification of arbitary RSCs. Depends if SMT-solver is complete.
If a safety property holds	False positives when RSC is over-approximation.	Complete if SMT-solver is complete.
If a dead transition exists	False negatives when RSC is over-approximation.	Complete if SMT-solver is complete.
If a sinkhole exists	False positives when RSC is over-approximation.	Complete if SMT-solver is complete.
If a start transition is unsatisfiable	Sound.	Complete if SMT-solver is complete.
If a transition relation is unsatisfiable	False positives when RSC is over-approximation.	Complete if SMT-solver is complete.

4.4 Completeness & Soundness

In this chapter we have defined and shown a verification procedure to check if a safety property holds, dead transitions or sinkholes exist and if there are unsatisfiable start transitions or relations. This verification procedure is a two-step algorithm where in step one we deduce and verify reachable state constraints and in step two check for any of the verification properties using the reachable state constraints. We have shown that for each of these steps and properties if they are sound and/or complete but we have yet to discuss if the overall approach is sound and/or complete.

The culmination of the two steps of the verification procedure do not add any new elements. The dependency between the steps are the reachable state constraints for the nodes. Therefore the culmination will not change if the overall approach is sound and/or complete other than the individual steps. We have shown that the generation and verification of reachable state constraints is sound. It is not complete as it is not able to generate and verify arbitrary reachable state constraints, but the procedure is complete that it may be used on a STS with an arbitrary graph structure. The verification properties are sound depending on if the reachable state constraint may be over-approximations or not. The verification properties are, however, complete in the sense that they may be used on arbitrary STSs with arbitrary RSCs.

In both steps we have to check if certain formulae are satisfiable or unsatisfiable by using a SMT-solver. Depending on whether the SMT-solver is complete this will also change if any of the steps are complete. Currently we accept non-linear arithmetic constraints for all aspects of the procedure. It is known that any procedure for the models of non-linear arithmetic constraints cannot be complete [13] and therefore no decision procedure exists. Even though this is proven, the dependency of completeness may depend solely on the SMT-solver used. We note this dependency.

To conclude this section, we show table 4.1 which gives an overview of the separate steps and if they are sound and/or complete.

4.5 Limitations

With this chapter we have introduced a verification procedure which we have shown may or may not be sound and complete. In chapter 4.4 we have described in which cases we cannot guarantee soundness and/or completeness. Other than soundness and completeness, in this section we also note a number of practical limitations.

1. No trace counterexamples - When the triple verification rule or a verification property has a counterexample, we are unable to give a trace from some start transition to the reachable state which is the counterexample. Creating such a trace may be possible but is left to future work.

2. No quantifiers in transition guards, relations or RSCs - Currently we limit the constraints to not include quantifiers. While we use quantifiers for some of our SMT-solver queries, we do not accept them as transition guards, relations or RSCs. This limitation may be resolved with future work.

4.6 Conclusion

This chapter has been dedicated to explain the verification procedure Goose. Goose is a verification procedure for STS specifications and is able to check if a safety property holds, if dead transitions or sinkholes exist and if there are unsatisfiable start transitions or relations. In order to verify these verification properties, Goose first generate and verifies reachable state constraints. These reachable state constraints are properties summarizing the reachable states for a certain node. Using the reachable state constraints, Goose is able to verify if any of the verification properties are violated.

We have shown that Goose is sound when the RCSs are generated by just the deduction rule and verified by the triple verification rule. In this case the RCSs are not an over- or under-approximation of the reachable states. In some cases Goose is not able to generate or verify a RCS. Users may give a RCS for a specific node which may then be verified by Goose. This RCS is possibly an over-approximation. Goose may then present false positives and negatives for the various verification properties.

We have also shown that Goose is possibly never complete. The verification procedure depends on a SMT-solver to query if certain formulae are satisfiable or unsatisfiable. It is known that there will never exist a complete decision procedure for non-linear arithmetic; which we also accept for the transition guards and relations and RCSs. The triple verification rule is also not complete. There may exist other verification rules which may make Goose (more) complete when it comes to verifying RCSs. Generation of RCS is done with the deduction rule and is incomplete when considering cycles. There may exist other techniques which may make Goose (more) complete when it comes to verifying RCSs.

Finally, we have also discussed the practical limitations and complexity of Goose. Currently there is no mechanism to find a trace when a counterexample is found on how to reach the counterexample. Also, currently we do not accept constraints with quantifiers. Future work may solve both limitations.

Chapter 5

Evaluation

To compare Goose with Z3 and nuXmv we have done a number of experiments to look at the execution times and solvability. We have done the following experiments:

1. Can Goose, Z3 and nuXmv prove a valid safety property for a survey of eight ING specifications and how fast? (Chapter 5.3)
2. Can Goose, Z3 and nuXmv find a valid counterexample for an invalid safety property for a survey of eight ING specifications and how fast? (Chapter 5.4)
3. Given the simple bankaccount example (figure 1.1) and an invalid safety property to check if the balance is below some maximum, how fast can Goose, Z3 and nuXmv find a valid counterexample? (Chapter 5.5)
4. Can Goose, Z3 and nuXmv prove that some number is not a fibonacci number and how fast? (Chapter 5.6)
5. How fast is the algorithm part of the prototype for all verification properties for the ING survey? (Chapter 5.7
6. How does Goose scale when the number of nodes are increased linearly? (Chapter 5.8)

In order to perform these experiments, we have developed a prototype implementing the algorithms from chapter 4. We shall discuss the prototype in chapter 5.1. The environment of our experiment is detailed in chapter 5.2. We shall conclude this chapter with a summary of our findings in chapter 5.9.

5.1 Prototype

The prototype implements the algorithms of chapter 4. This prototype has served as the basis for any experiments and as a reference implementation.

The prototype has been written in the programming language Scala [1]. This is an object-oriented programming language with functional programming influences. It generates to Java Virtual Machine (JVM) bytecode which is a multi-platform object-oriented assembly-like language which is interpreted by the Java Virtual Machine. We have chosen to use Scala as ING prefers JVM-based languages. After this research, the prototype may be used or further researched by ING and they will be familiar with the programming language the prototype is written in.

The Goose algorithm relies on an SMT-solver. We have chosen to use Z3. Not only does it have the PDR engine as explained in chapter 3.3 but it is also an SMT-solver as explained in chapter 2.3.1. We have chosen for Z3 as it is being maintained by Microsoft Research and we expect it to be updated with the latest theories in the future. Also, Z3 offers a programmatic API for Java.

We have checked if the prototype performs as expected using unit and integration tests. Our test suite has 84.41% statement coverage and 83.78% branch coverage including the core Goose algorithm code but excluding any command-line user-interface code or generator code to generate Z3 and nuXmv specifications from Goose specifications. We have chosen to exclude the generator code from the test coverage check because the focus of our testsuite is to see if Goose functions correctly. The Z3 and nuXmv specifications generated with the code generators have been checked for correctness.

5.2 Experimental Setup

In order to perform our experiments we used the same machine for all experiments. The details of the machine and software used is detailed in chapter 5.2.3. We have chosen to use an average workstation to allow us to answer the research question if the procedure is fast enough to be used while developing specifications on an average workstation.

With our experiments we evaluate the execution times and the solvability of Goose, Z3 with PDR strategy and nuXmv with IC3 strategy. As all three tools accept different input languages, we need some way to express the STS specification directly in Z3 and nuXmv. We have already discussed this in chapters 3.3.2 and 3.2.5. We have created code generators to map the STS specifications from the Goose input language to the other languages. Any specifications mapped to another language have been checked to see if they are correct.

5.2.1 ING survey

In collaboration with ING we have created 8 STS specifications which we deem representable of the overall verification needs of ING. We call these specifications the ING survey and they are included in appendix A. We have designed these 8 specifications to match the overall complexity found in ING's verification needs. We matched the complexity in the amount and structure of cycles, complexity of constraints and amount of nodes and transitions from least complex to most complex.

5.2.2 Warm-up

Z3 and nuXmv are implemented in C++ and Goose is implemented in Scala. C++ is compiled directly to machine code while Scala is compiled to Java Virtual Machine (JVM) bytecode. The JVM is an execution engine available for Windows, various Linux distributions and macOS and allows the same JVM compiled program to be executed across all operating systems. The JVM interprets the JVM bytecode and compiles it to machine code on the fly. This is called Just-In-Time (JIT) compilation and has various optimizations such as optimizing a piece of JVM code which is executed more often than other parts of the program. This leads to issues when evaluating the execution time of a program which is executed repeatedly within the same JVM instance as the first execution may be considerably slower than the second one. This is countered with a warm-up where the program is executed multiple times without registering the execution time before executing the same program again multiple times and registering the execution time. We have used this warm-up technique for the experiments of chapters 5.7 and 5.8. With these experiments we are interested in the results of the execution time for just the algorithm without any interference of the JIT compiler. With the other experiments we are interested in the execution for the situation as if we would have used the tool ourselves. In this situation, we would not warm-up the program for a best case execution time and therefore the warm-up technique is not used.

5.2.3 Testing Environment

The experiments were performed within the same testing environment. This environment consisted of:

- **OS**: Fedora 27 with kernel 4.14.8-300
- **CPU**: i7-6700HQ @ 2.60GHz
- **RAM**: 2x8GB DDR4 2133Mhz
- **Storage**: 256Gb NVMe SSD.

The following software versions were used:

- **Java**: 1.8.0 update 144 Oracle
- **Scala**: 2.12.4
- **Z3**: 4.5.0
- **nuXmv**: 1.1.1
- **MCMT**: 2.5.2

5.3 ING specifications with Valid Safety Property

The goal of this experiment is to show if Goose is suitable to be used with ING specifications and if it can be used during development. We have used the specifications of the ING survey (chapter 5.2.1). We have included a valid safety property and queried the tools to prove this property. An graphical overview of the specifications and the safety properties tested may be seen in appendix A. We are interested in the total execution time of the programs as if we would have used the tools ourselves. Therefore, we have not used the warm-up technique.

Analysis

The results are shown in table 5.1. Some queries returned the result 'unknown' for Z3 and nuXmv and we are unable to say why they returned unknown. We do note the following:

- Z3 & nuXmv are $\sim$ 100 times faster than Goose.
- Differences in time of execution between specifications are near negligible for all tools.
- Z3 & nuXmv cannot prove the safety property for the specifications containing non-linear arithmetic.
- Goose is able to prove the safety property for the specifications containing non-linear arithmetic.

The small difference in time of execution between specifications is expected: The specifications contain a similar amount of nodes and transitions and none of the cycles are forced to be unfolded by the safety property.

While the execution time of Goose is between 0.887 and 1.062 seconds, we see that Z3 and nuXmv are between 0.015 and 0.047 seconds. Z3 and nuXmv are very fast compared to Goose for these specifications. The reason seems to be the overhead of Goose starting the JVM and compiling the specification that Goose is almost $\sim$ 100 times slower than Z3 and nuXmv as may be seen in table 5.6 (chapter 5.7) where we do not take these parts of the program into account while timing the execution time of the two phases of the algorithm. Goose's total execution time is than between 0.025 and 0.070 seconds. Still Goose is slower.

Z3 & nuXmv cannot prove the safety property for specifications containing non-linear arithmetic. For Z3 this is unexpected. Hoder et. al. [19] have shown how they extended the PDR algorithm with non-linear arithmetic. However, they have not shown that their approach is able to reason over specifications with arbitrary non-linear arithmetic constraints. We think the (lack of) result for the specifications containing non-linear arithmetic are examples to show that their current approach is not complete. For nuXmv this is expected. Cimatti et. al. [11] state in the conclusion that they will look at non-linear arithmetic in the future stating indirectly that nuXmv currently does not work with non-linear arithmetic.

The goal of this experiment was to see if Goose returns results fast enough to be used while writing these specifications. In chapter 1.1 we have defined the bounds as execution time within a few seconds on an average workstation. Our test environment may be considered an average workstation and the execution time was always around a second for this prototype. We may therefore conclude that Goose may be used during the development of specifications to prove valid safety properties.

5.4 ING specifications with Invalid Safety Property

The goal of this experiment is again to show if Goose returns results fast enough to be used with ING specifications and if it can be used during development.

Table 5.1: Results for ING specifications with valid safety property. Average in seconds after 100 runs or unknown if tool could not prove the safety property.

Specification	**Contains non-linear arithmetic**	**Contains cycle**	**Goose**	**Z3**	**nuXmv**
Account Holder	-	-	1.046	0.023	0.015
Bank Account	-	Yes	1.000	0.022	0.016
Bank Account with Interest	Yes	Yes	1.018	Unknown	Unknown
Linear Loan	Yes	Yes	1.062	Unknown	Unknown
Money Movement	-	Yes	0.887	0.029	0.014
Non-Interest Loan	Yes	Yes	1.002	Unknown	Unknown
Transfer	-	-	0.911	0.022	0.014
TriangleOfNodes	-	Yes	0.973	0.047	0.023

The experiment is similar to the previous experiment of chapter 5.3. We have included an invalid safety property and queried the tools to find a counterexample for this property. A graphical overview of the specifications and the invalid safety property used may be seen in appendix A.

Analysis

The results are shown in table 5.2. Again some queries returned the result 'unknown' for Z3 and nuXmv. In this case we are also unable to say why they returned unknown. We note the following:

- Z3 & nuXmv are $\sim$ 100 times faster than Goose
- Differences in time of execution between specifications are near negligible for all tools
- nuXmv cannot find counterexamples for any safety property for the specifications containing non-linear arithmetic
- Z3 cannot find a counterexample for one specification

The difference in time of execution between specifications is expected: The specifications contain a similar amount of nodes and transitions and none of the cycles are forced to be unfolded by the safety property.

Again we see that Z3 & nuXmv are $\sim$ 100 faster than Goose. As we explained in the previous section, this seems to be due to start-up overhead of starting the JVM and compiling the specification. However, we have not ran a similar benchmark as for the previous section taking only the algorithm execution time into account so we are unable to say definitively if it is the same reason although it seems likely.

As explained in the previous section, Z3 should be able to reason (partially) about non-linear arithmetic while nuXmv is not able to reason about non-linear arithmetic. This is reflected in the results as Z3 is able to find counterexamples for 2 of the specifications containing non-linear arithmetic while nuXmv isn't able to find any counterexample for the 3 specifications containing non-linear arithmetic.

Table 5.2: Results for ING specifications with invalid safety property. Average in seconds after 100 runs or unknown if tool could not disprove the safety property.

Specification	**Contains non-linear arithmetic**	**Contains cycle**	**Goose**	**Z3**	**nuXmv**
Account Holder	-	-	1.041	0.016	0.017
Bank Account	-	Yes	1.009	0.015	0.016
Bank Account with Interest*	Yes	Yes	1.036	0.015	Unknown
Linear Loan*	Yes	Yes	1.057	Unknown	Unknown
Money Movement	-	Yes	0.894	0.014	0.015
Non-Interest Loan*	Yes	Yes	0.998	0.092	Unknown
Transfer	-	-	0.956	0.014	0.016
TriangleOfNodes	-	Yes	0.980	0.020	0.017

As explained in the previous and current experiment, the goal of these experiments is to find if Goose is suitable to be used during the development of specifications on an average workstation. The results for invalid safety properties are again within the bounds of a few seconds as the execution time is around a second. We may therefore conclude that Goose is not only suitable to prove valid safety properties but also to disprove invalid safety properties during the development of specifications.

5.5 Forced Cycle Unfolding

The goal of this experiment is to show that the proposed approach nicknamed Goose does not require more resources exponentially due to forced cycle unfolding and to see if Z3 and nuXmv indeed show this problem in practice. We have explained this phenomenon in chapter 3.2.7. We have used a variation of the bank account specification from the previous section where we not only limit the amount you are able to deposit and withdraw to 50 but also set the initial balance to 50. We then query the tools if the balance of the bank account ever exceeds a set maximum. This forces the IC3/PDR algorithm to unfold the cycle until a balance higher than the maximum is reached. This maximum is the variable we change throughout the experiment. A graphical representation of this specification may be seen in appendix B

Analysis

The results are shown in table 5.3 and in figure 5.1. We note the following:

- Goose has a near constant execution time
- Z3 increases exponentially in execution time as the maximum balance is increased
- nuXmv increases exponentially in execution time as the maximum balance is increased

Table 5.3: Results for limited bank account specification with invalid safety property. Average in seconds after 100 runs.

Maximum balance	**Goose**	**Z3**	**nuXmv**
100	1.016	0.020	0.020
1000	1.004	0.069	0.576
2000	1.003	0.128	4.582
3000	0.997	0.198	17.284
4000	1.002	0.275	42.685
10000	0.998	0.846	-
20000	1.061	2.391	-
30000	0.998	4.825	-
100000	1.006	55.795	-

- nuXmv increases exponentially in execution time more rapidly than Z3 as the maximum balance is increased

It is expected for Goose to remain constant in execution time as the amount of work does not change. First all reachable state constraints are deduced and verified and then it is checked if all reachable state constraints imply the safety constraint; the value for the maximum balance does not change this amount of work. As we only change the maximum of the balance and not the safety constraint itself it solely depends on the SMT-solver when the counterexample is found. Our results show that the execution time does not significantly change as we change the maximum balance so it seems the SMT-solver is able to find the counterexample at roughly the same amount of time for each value of the maximum balance.

We expected both Z3 and nuXmv to rapidly increase in execution time when we change the value for maximum balance. The results reflect our hypothesis. What we did not expect is that nuXmv rapidly increases in execution time for a lower value for maximum balance then Z3. This might be explained by some implementation difference between the IC3 and PDR engines of nuXmv and Z3 but further research into the internals of Z3 and nuXmv is necessary to say for certain.

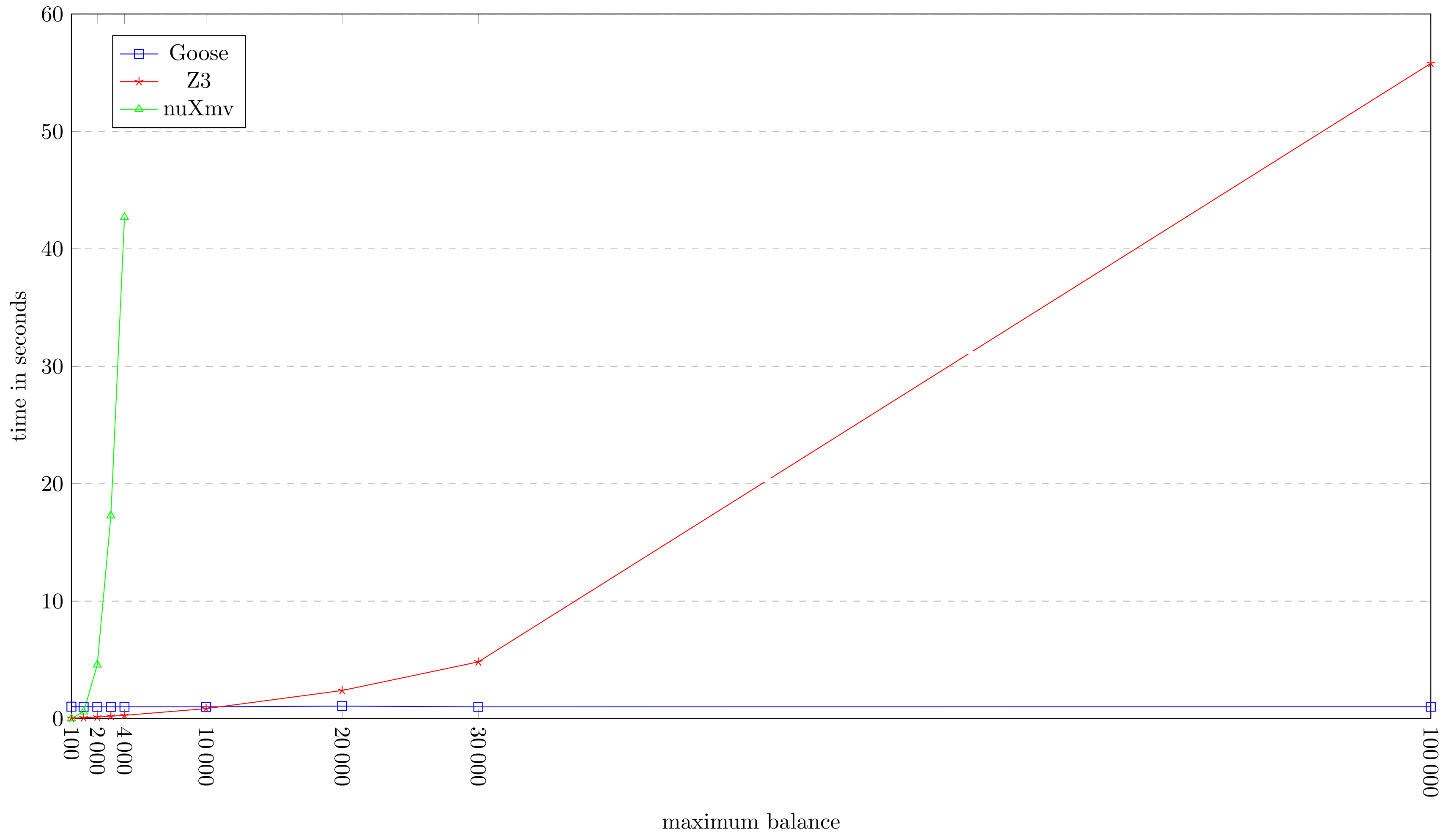


Figure 5.1: Limited bank account maximum counterexample

5.6 Fibonacci Numbers

The goal of this experiment is to show that Goose is unable to verify the safety property $a! = NUM$ where a is a variable fibonacci number and NUM is a constant number that should not be a fibonacci number. The specification used is included in appendix C. Goose is unable to (dis)prove the safety property as the query to verify the user-given RSC using the triple verification rule cannot be proven. The triple verification rule is not strong enough as Goose returns a spurious counterexample.

It is interesting to see that Z3 and nuXmv are able to (dis)prove the safety property. We have included the results in table 5.4.

Table 5.4: Results for fibonacci specification with invalid safety property. Average in seconds after 100 runs.

NUM	**Goose**	**Z3**	**nuXmv**
10	Spurious counterexample	0.027	0.021
100	Spurious counterexample	0.154	0.072
1000	Spurious counterexample	0.587	0.203
10000	Spurious counterexample	2.196	0.402
100000	Spurious counterexample	5.777	0.781
1000000	Spurious counterexample	11.78	1.413
10000000	Spurious counterexample	29.61	2.348
1000000000	Spurious counterexample	37.012	5.07

Analysis

The results are shown in table 5.4. We note the following:

- Goose is unable to (dis)prove the safety property as it is unable to verify the user-given RSC for the node in the specification.
- The execution time of Z3 & nuXmv increase as NUM increases.
- The execution time of Z3 increases more rapidly than the execution time of nuXmv when NUM increases.

It is expected that Goose is unable to (dis)prove the safety property. As we have noted before in chapter 3.1, mathematical induction is unable to prove properties for the fibonacci sequence as k-induction with $k >= 2$ is necessary. The triple verification rule is based on mathematical induction and is therefore not strong enough to prove the user-given RSC for the node in the specification.

It is also expected that the execution time increases for Z3 and nuXmv when NUM increases as we are forcing the cycle to unfold again.

It was not expected to see a large difference in the execution times between Z3 and nuXmv for the same value of NUM. We are unable to give a reason. It might have to do with the implementation differences between the tools and the way the counterexample information is used to refine the frames with the IC3 algorithm.

5.7 ING Specifications with all Verification Properties

The goal of this experiment is to show the execution times of the Goose algorithm for all verification properties specified in chapter 4.3 for the specifications of the ING survey of appendix A. To reiterate, we are interested if and how fast Goose is able to prove the valid safety properties, prove there are no sinkholes, prove there are no dead transitions, prove there are no unsatisfiable start transitions and to prove there are unsatisfiable relations.

We are interested in the execution time of just the algorithm so the compiling and setup phases of the prototype are not taken into account. We also use a warm-up to let the JVM optimize the JVM bytecode. We are also interested how the execution time is divided across the two phases of the verification procedure so we note the execution time of the two phases and the percentage of the total execution time of the two phases combined.

Analysis

The results are shown in tables 5.5, 5.6, 5.7, 5.8 and 5.9. We note the following:

- Goose has a false positive in the Linear Loan specification when checking for sinkholes and unsatisfiable relations.
- Goose presents an SMT query to Z3 which Z3 is unable to solve in the Non-Interest Loan specification when checking for dead transitions, unsatisfiable relations and unsatisfiable start transitions and returns unknown.
- The maximum time taken in the generation & verification phase is 0.046 seconds.
- The maximum time taken in the verification properties phase is 0.041 seconds.

We have noted that SMT solvers may return unknown in some cases. In the results we see that Z3 returns unknown when checking for dead transitions, unsatisfiable relations and unsatisfiable start transitions for the Non-Interest Loan specification. Upon further analysis, we see that this has to do with non-linear arithmetic and in all three cases it is a query with the same term that fails. In the query is the same term consisting of three *real* variables with an equality and multiplication.

The maximum time taken in the two phases is 0.046 and 0.041 seconds. This is unrealistic in a regular use-case as we have optimized the JVM byte code by warming up the JVM. However, we do note that the algorithm (without setup and compiler parts of the program) is below 100ms and therefore satisfies the requirement that the useful verification properties should be verified in reasonable time.

			Generation & Verification phase			Verification Properties phase		
Specification	**Contains non-linear arithmetic**	**Contains cycle**	**# SMT calls**	**% of to-tal time**	**Execution time**	**# SMT calls**	**% of to-tal time**	**Execution time**
Account Holder	-	-	5	57.37%	0.031	4	42.63%	0.023
Linear Loan	Yes	Yes	6	56.74%	0.041	5	43.26%	0.031
Money Movement	-	Yes	3	60.38%	0.019	2	39.62%	0.012
Bank Account	-	Yes	6	55.88%	0.039	5	44.12%	0.031
Bank Account with Interest	Yes	Yes	7	55.06%	0.046	6	44.94%	0.038
Non-Interest Loan	Yes	Yes	-	-	-	-	-	Unknown (Z3)
Transfer	-	-	3	61.29%	0.019	2	38.71%	0.012
TriangleOfNodes	-	Yes	7	54.36%	0.041	6	45.64%	0.035

Table 5.5: Results for ING specifications checking for dead transitions. Average in seconds after 50 warm-up runs and 100 timed runs. Only generation/verification phase and verification property phase of prototype is timed. Setup and compilation phases are skipped.

			Generation & Verification phase			Verification Properties phase		
Specification	**Contains non-linear arithmetic**	**Contains cycle**	**# SMT calls**	**% of total time**	**Execution time**	**# SMT calls**	**% of total time**	**Execution time**
Account Holder	-	-	5	52.32%	0.031	5	47.68%	0.028
Linear Loan	Yes	Yes	6	57.43%	0.040	5	42.57%	0.030
Money Movement	-	Yes	3	75.87%	0.019	1	24.13%	0.006
Bank Account	-	Yes	6	69.18%	0.039	3	30.82%	0.017
Bank Account with Interest	Yes	Yes	7	72.13%	0.045	3	27.87%	0.017
Non-Interest Loan	Yes	Yes	5	58.93%	0.034	4	41.07%	0.023
Transfer	-	-	3	51.44%	0.018	3	48.56%	0.017
TriangleOfNodes	-	Yes	7	70.14%	0.041	3	29.86%	0.017

Table 5.6: Results for ING specifications checking valid safety property. Average in seconds after 50 warm-up runs and 100 timed runs. Only generation/verification phase and verification property phase of prototype is timed. Setup and compilation phases are skipped.

			Generation & Verification phase			Verification Properties phase		
Specification	**Contains non-linear arithmetic**	**Contains cycle**	**# SMT calls**	**% of to-tal time**	**Execution time**	**# SMT calls**	**% of to-tal time**	**Execution time**
Account Holder	-	-	5	64.75%	0.031	3	35.25%	0.017
Linear Loan	Yes	Yes	6	69.32%	0.040	3	30.68%	0.018 (false pos-itive)
Money Movement	-	Yes	3	70.61%	0.018	1	29.39%	0.008
Bank Account	-	Yes	6	77.22%	0.040	2	22.78%	0.012
Bank Account with Interest	Yes	Yes	7	79.76%	0.046	2	20.24%	0.012
Non-Interest Loan	Yes	Yes	5	74.68%	0.034	2	25.32%	0.011
Transfer	-	-	3	76.63%	0.019	1	23.37%	0.006
TriangleOfNodes	-	Yes	7	70.62%	0.041	3	29.38%	0.017

Table 5.7: Results for ING specifications checking for sinkholes. Average in seconds after 50 warm-up runs and 100 timed runs. Only generation/verification phase and verification property phase of prototype is timed. Setup and compilation phases are skipped.

			Generation & Verification phase			Verification Properties phase		
Specification	**Contains non-linear arithmetic**	**Contains cycle**	**# SMT calls**	**% of total time**	**Execution time**	**# SMT calls**	**% of total time**	**Execution time**
Account Holder	-	-	5	56.88%	0.031	4	43.12%	0.024
Linear Loan	Yes	Yes	6	57.34%	0.040	5	42.66%	0.030 (false positive)
Money Movement	-	Yes	3	60.67%	0.019	2	39.33%	0.012
Bank Account	-	Yes	6	53.00%	0.039	5	47.00%	0.034
Bank Account with Interest	Yes	Yes	7	52.31%	0.044	6	47.69%	0.041
Non-Interest Loan	Yes	Yes	-	-	-	-	-	Unknown (Z3)
Transfer	-	-	3	61.68%	0.019	2	38.32%	0.012
TriangleOfNodes	-	Yes	7	53.87%	0.040	6	46.13%	0.034

Table 5.8: Results for ING specifications checking for unsatisfiable relations. Average in seconds after 50 warm-up runs and 100 timed runs. Only generation/verification phase and verification property phase of prototype is timed. Setup and compilation phases are skipped.

			Generation & Verification phase			Verification Properties phase		
Specification	**Contains non-linear arithmetic**	**Contains cycle**	**# SMT calls**	**% of total time**	**Execution time**	**# SMT calls**	**% of total time**	**Execution time**
Account Holder	-	-	5	84.27%	0.031	1	15.73%	0.006
Linear Loan	Yes	Yes	6	86.94%	0.041	1	13.06%	0.006
Money Movement	-	Yes	3	75.49%	0.019	1	24.51%	0.006
Bank Account	-	Yes	6	86.25%	0.040	1	13.75%	0.006
Bank Account with Interest	Yes	Yes	7	87.89%	0.046	1	12.11%	0.006
Non-Interest Loan	Yes	Yes	-	-	-	-	-	Unknown (Z3)
Transfer	-	-	3	76.07%	0.019	1	23.93%	0.006
TriangleOfNodes	-	Yes	7	87.78%	0.041	1	12.22%	0.006

Table 5.9: Results for ING specifications checking for unsatisfiable start transitions. Average in seconds after 50 warm-up runs and 100 timed runs. Only generation/verification phase and verification property phase of prototype is timed. Setup and compilation phases are skipped.

5.8 Scaling of Goose, Z3 PDR and nuXmv

The goal of this experiment is to show how Goose, Z3 and nuXmv scale when increasing the number of nodes and transitions. As not all of the tools support non-linear arithmetic, we have generated specifications with linear arithmetic. We have generated two types of specifications: 1) STS with a single *integer* state variable which is given a new value after each transition with no relation to the previous value (see figure 5.2) and 2) STS with a single *integer* state variable which adds some number to the current value after each transition (see figure 5.3). The maximum number set/accumulated on each transition (j_1 through j_{m-1} in the figures) is between 1 and 10 and therefore i is not higher than $10 * (m - 1)$ where m is the number of nodes in the specification.

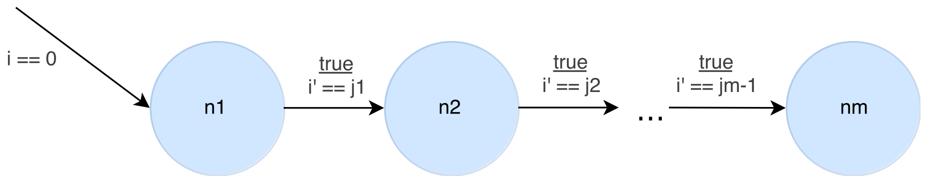


Figure 5.2: STS where the amount of nodes and transitions are scaled. Variable i is set a new value between 1 and 10 after each transition.
Guards are above the line, relations below the line for transitions

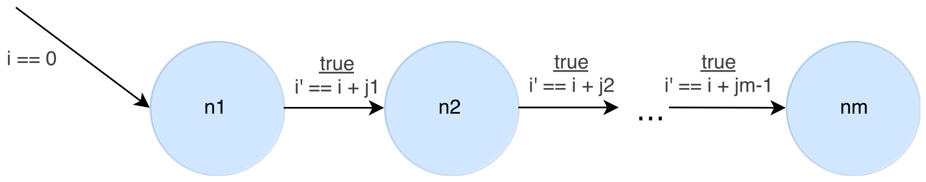


Figure 5.3: STS where the amount of nodes and transitions are scaled. Variable i is added a new value between 1 and 10 after each transition.
Guards are above the line, relations below the line for transitions

The check we will perform is proving a valid safety property valid. We will check 1) if $i >= 0$ holds for the state variable i and 2) if $i <= 10000$ holds for the state variable i. We have chosen these two safety properties as we hypothesize that the IC3 algorithms will not have an issue with the first safety property but will perform poorly when trying to verify the second safety property when the i is accumulated throughout the specification. This hypothesis is based on the fact that IC3 will probably use $i >= 0$ as a frame for all reachable states while the second safety property forces the creation of a significant number of frames as there is no abstraction for the safety property. We shall use the results to confirm or deny this hypothesis.

Analysis

The results are shown in tables 5.10, 5.11, 5.12 and 5.13. We note the following:

- The execution time for Z3 and nuXmv rise quickly for the STS where i is accumulated where safety property 2 $i <= 10000$ is checked. (table 5.13)
- The execution time of Goose rises more quickly for the STSs where i is accumulated than the STSs where i is set a new value after each transitions.
- The execution time of Z3 rises more slowly than the execution time of nuXmv for the results of tables 5.10, 5.11 and 5.12.
- The execution time of nuXmv rises more slowly than the execution time of nuXmv for the results of table 5.13.
- The execution time of Goose rises more-than-linearly as the number of nodes rises linearly.

The first analysis point confirms our theory. Z3 and nuXmv do indeed perform poorly when i is accumulated and the safety property $i <= 10000$ is checked.

The second analysis point is unexpected. When i is accumulated, a long RSC is accumulated with many variables which have to be assigned. When i is set a new value after each transition, the SMT solver should only look at the term with the state variable for that node to determine if it implies the safety property. Further analysis is needed to determine the cause for this and is left to future work.

The third and fourth analysis points are interesting. Z3 performs better for the result sets with a low execution time while nuXmv performs better for the result set where the execution time quickly rises.

The fifth and final analysis point is interesting and is the goal of this experiment. Goose seems to scale more-than-linearly when the amount of nodes increases. Further analysis is needed to determine if this is due to the longer RSCs and the execution time of the SMT solver or due to the added complexity of the verification procedure. This is left to future work.

	Generation & Verification phase			**Verification Properties phase**			**Z3**	**nuXmv**
# of nodes	**# SMT calls**	**% of total time**	**Execution time**	**# SMT calls**	**% of total time**	**Execution time**	**Execution time**	**Execution time**
1	1	50.59%	0.006	1	49.41%	0.006	0.011	0.013
10	10	51.39%	0.059	10	48.61%	0.056	0.016	0.014
50	50	52.47%	0.316	50	47.53%	0.287	0.018	0.018
100	100	54.12%	0.683	100	45.88%	0.579	0.02	0.024
200	200	57.73%	1.646	200	42.27%	1.205	0.025	0.038
300	300	61.49%	2.956	300	38.51%	1.851	0.028	0.054
400	400	62.88%	4.338	400	37.12%	2.561	0.034	0.075
500	500	65.63%	6.282	500	34.37%	3.29	0.037	0.099
600	600	70.53%	9.882	600	29.47%	4.129	0.042	0.126
700	700	73.39%	13.654	700	26.61%	4.952	0.046	0.165
800	800	72.51%	15.298	800	27.49%	5.801	0.053	0.206
900	900	74.62%	19.53	900	25.38%	6.644	0.056	0.258
1000	1000	79.18%	29.073	1000	20.82%	7.643	0.061	0.299

Table 5.10: Results for checking for the safety property $i >= 0$ for the STS with arbitrary number of nodes of figure 5.2. Average in seconds after 50 warm-up runs and 100 timed runs for the phases of Goose. Setup and compilation phases are skipped. Z3 and nuXmv have been run 100 times and the execution time is averaged.

	Generation & Verification phase			**Verification Properties phase**			**Z3**	**nuXmv**
# of nodes	**# SMT calls**	**% of total time**	**Execution time**	**# SMT calls**	**% of total time**	**Execution time**	**Execution time**	**Execution time**
1	1	50.76%	0.006	1	49.24%	0.006	0.012	0.013
10	10	51.58%	0.06	10	48.42%	0.056	0.017	0.014
50	50	53.36%	0.332	50	46.64%	0.29	0.02	0.019
100	100	55.79%	0.754	100	44.21%	0.597	0.024	0.024
200	200	62.17%	2.134	200	37.83%	1.299	0.028	0.039
300	300	67.73%	4.318	300	32.27%	2.057	0.032	0.056
400	400	66.88%	5.887	400	33.12%	2.915	0.039	0.076
500	500	69.81%	8.924	500	30.19%	3.859	0.044	0.101
600	600	72.54%	12.805	600	27.46%	4.846	0.049	0.131
700	700	80.20%	24.411	700	19.80%	6.027	0.055	0.169
800	800	82.17%	33.302	800	17.83%	7.225	0.099	0.211
900	900	83.90%	44.402	900	16.10%	8.518	0.068	0.262
1000	1000	85.62%	59.066	1000	14.38%	9.918	0.076	0.306

Table 5.11: Results for checking for the safety property $i >= 0$ for the STS with arbitrary number of nodes of figure 5.3. Average in seconds after 50 warm-up runs and 100 timed runs for the phases of Goose. Setup and compilation phases are skipped. Z3 and nuXmv have been run 100 times and the execution time is averaged.

	Generation & Verification phase			**Verification Properties phase**			**Z3**	**nuXmv**
# of nodes	**# SMT calls**	**% of total time**	**Execution time**	**# SMT calls**	**% of total time**	**Execution time**	**Execution time**	**Execution time**
1	1	50.93%	0.006	1	49.07%	0.006	0.011	0.013
10	10	51.33%	0.059	10	48.67%	0.056	0.016	0.014
20	20	51.60%	0.12	20	48.40%	0.112	0.017	0.015
50	50	52.45%	0.314	50	47.55%	0.285	0.019	0.018
60	60	52.78%	0.382	60	47.22%	0.341	0.019	0.019
100	100	54.03%	0.683	100	45.97%	0.581	0.02	0.024
200	200	57.78%	1.627	200	42.22%	1.189	0.025	0.038
300	300	61.48%	2.934	300	38.52%	1.838	0.028	0.053
400	400	64.72%	4.672	400	35.28%	2.547	0.034	0.074
500	500	67.92%	6.984	500	32.08%	3.298	0.037	0.098
600	600	68.28%	8.81	600	31.72%	4.093	0.042	0.128
700	700	70.44%	11.642	700	29.56%	4.886	0.046	0.166
800	800	75.56%	17.736	800	24.44%	5.737	0.053	0.205
900	900	74.55%	19.435	900	25.45%	6.635	0.057	0.258
1000	1000	78.91%	28.491	1000	21.09%	7.614	0.061	0.303

Table 5.12: Results for checking for the safety property $i <= 10000$ for the STS with arbitrary number of nodes of figure 5.2. Average in seconds after 50 warm-up runs and 100 timed runs for the phases of Goose. Setup and compilation phases are skipped. Z3 and nuXmv have been run 100 times and the execution time is averaged.

	Generation & Verification phase			**Verification Properties phase**			**Z3**	**nuXmv**
# of nodes	**# SMT calls**	**% of total time**	**Execution time**	**# SMT calls**	**% of total time**	**Execution time**	**Execution time**	**Execution time**
1	1	50.86%	0.006	1	49.14%	0.006	0.011	0.013
10	10	51.60%	0.061	10	48.40%	0.057	28.43	0.081
20	20	51.90%	0.121	20	48.10%	0.112	50.681	1.449
50	50	53.28%	0.329	50	46.72%	0.288	-	32.906
60	60	53.99%	0.409	60	46.01%	0.349	-	73.453
100	100	55.69%	0.758	100	44.31%	0.603	-	-
200	200	60.80%	1.988	200	39.20%	1.282	-	-
300	300	67.58%	4.249	300	32.42%	2.038	-	-
400	400	66.91%	5.844	400	33.09%	2.89	-	-
500	500	69.81%	8.846	500	30.19%	3.825	-	-
600	600	79.18%	18.465	600	20.83%	4.856	-	-
700	700	80.15%	23.952	700	19.85%	5.93	-	-
800	800	82.06%	33.067	800	17.94%	7.227	-	-
900	900	84.09%	45.125	900	15.91%	8.538	-	-
1000	1000	85.26%	57.727	1000	14.74%	9.982	-	-

Table 5.13: Results for checking for the safety property $i <= 10000$ for the STS with arbitrary number of nodes of figure 5.3. Average in seconds after 50 warm-up runs and 100 timed runs for the phases of Goose. Setup and compilation phases are skipped. Z3 and nuXmv have been run 100 times and the execution time is averaged.

5.9 Conclusion

With this chapter we have presented a number of experiments and analysed the results. The goal of these experiments were to find how Goose performs in terms of solveability and execution time as compared to Z3 and nuXmv. We have found that:

- The execution time of Goose scales more-than-linearly with the number of nodes and transitions as is shown by the experiments of chapter 5.8
- The execution time of Goose remains constant with safety properties that force cycle unfolding with IC3/PDR algorithms as is shown by arguments made and the experiment of chapter 5.5
- Goose has a higher solveability for specifications containing non-linear arithmetic as is shown by the experiments of chapter 5.3 and chapter 5.4.
- Goose is usable during development of specifications by ING as is shown by the experiments of chapters 5.3, 5.4 and 5.7.
- Goose is unable to verify safety property for the fibonacci specification while Z3 and nuXmv are able to verify the safety property as is shown by the experiments of chapter 5.6.
- The overall execution time is higher than the execution time of Z3 and nuXmv except when we force a significant number of frames for the IC3 algorithm as is confirmed by the experiments in chapters 5.3, 5.4, 5.5 and 5.8.

We have discussed if the current results for Goose satisfy the requirements for ING and they have confirmed that Goose will fit in their use-case.

Chapter 6

Future Work

Throughout this thesis we have left certain questions to future work. This chapter serves as an overview of the different tasks which may be researched in the future.

1. **Reduce reachable state constraints when possible** - We have shown in chapter 4.2.1 that the generated RSC in some cases may be reduced when certain parts of the RCS are not necessary to model all the reachable states. With future work we may look at techniques and tools to optimize these constraints to simple constraints while preserving the states modelled by the RCS.

2. ***k*-induction & other proving techniques** - We have shown in chapter 4.2.2 that we use the triple verification rule to verify RSCs. This proving-technique is based on mathematical induction over paths. Stronger proving-techniques exist such as k-induction. Future work may research which proving-techniques or rules may be added to the verification procedure to allow for more RCSs to be verified.

3. **Parallelisation of algorithm** - We have shown in chapter 4.2.3 that the order of work for the verification procedure is per strongly connected component (SCC) using the condensation graph. We have also shown that it is possible to verify multiple SCCs concurrently while preserving soundness. Future research may improve the algorithm to leverage concurrency to speed up the algorithm.

4. **Z3 PDR and verification properties** - In chapter 3.3 we have shown how Z3 with the PDR strategy may be used to verify safety properties of STSs. Opposed to nuXmv, with Z3 and PDR it is possible to reference all reachable states for some node. Future research may be done to create a mapping from the verification properties to a Z3 PDR specification and evaluate how well Z3 with PDR is able to verify the same properties Goose is able to verify.

5. **References between specifications** - With Goose in its current form we are able to verify STS specifications without referencing other STS specifications. This may be used to put the different symbolic transitions systems in parallel for some transitions. NuSMV [10] and nuXmv [11] are

able to express relations between multiple specifications using modules. Future work may look into how the verification procedure and properties may be modified to allow for references between specifications.

6. **Quantifiers in language** - Currently we allow users to specify constraints for guards, transition relations and RSCs without quantifier. We have not researched if any of the procedure steps have to change when allowing quantifiers. Future research may look into allowing quantifiers in the input language and what consequences this has.

7. **Trace for counterexamples** - Currently counterexamples for verifying a RCS and the verification property may be given. This entails a reachable state for some node. We have not yet provided a mechanism to find a path from a starting transition to this counterexample. Future research may look into creating such a mechanism.

8. **Further analysis of complexity Goose** - We have seen in chapter 5.8 that Goose scales exponentially when the amount of nodes increases linearly. We are interested in why it increases exponentially.

9. **Reachable state constraint generation for cycles** - We have seen in chapter 4.2 that the deduction rule is unable to generate RSCs for nodes in a cycle. There exist invariant generation techniques which may help to generate RSCs for nodes in cycles in some cases.

Chapter 7

Conclusion

This research has started with the goal to define useful verification properties to verify for STSs which is the formalism used in the domain specific language Rebel. Also a technique, existing or new, had to be found/designed to verify these useful verification properties; preferably one that ran in reasonable time (a few seconds) on an average workstation so it could be just when developing the Rebel specifications. This has led to the a number of research questions in chapter 1.1.

We have defined five useful verification properties: safety properties, existence of sinkholes, existence of dead transitions, existence of unsatisfiable relations and existence of unsatisfiable start transitions. These verification properties have been defined in chapter 4.3.

In order to solve the research questions, a literature study has been conducted on existing techniques to solve one or more of the useful verification properties we have defined. In chapter 3 we have described the tools MCMT, nuXmv and Z3. We have shown that MCMT may not be used for arbitrary STSs while nuXmv and Z3 both implement a variant of the IC3 algorithm to verify safety properties for arbitrary STSs. We have shown in chapter 3.2.7 that the amount of work done by IC3 may blow up when considering STSs with cycles and specific safety properties which force the algorithm to unfold the cycle for a large number of iterations. We have also confirmed this with the experiment of chapter 5.5. This has led to the development of a new approach as these existing technique did not fit the requirements of ING.

This new approach is nicknamed Goose. As shown in chapter 4, this approach uses properties called reachable state constraints (RSC) to summarize all reachable states for a node. The approach uses the deduction rule to deduce the RSC for nodes of a STS when the node is not in a cycle and otherwise requests a RSC from the user. Any unverified RSC is verified with the triple verification rule.

We have also evaluated Goose with Z3 and nuXmv in terms of solveability and execution time in chapter 5. We have shown that Goose is able to solve more specifications containing non-linear arithmetic and it scales more-than-linearly when increasing the number of nodes linearly. Z3 and nuXmv are able to solve certain safety properties which Goose is unable to solve as is shown with the evaluation of the fibonacci specification (chapter 5.6). The reverse is also true as is shown with the non-linear arithmetic specifications of the ING survey (chapters 5.3 and 5.4). Finally, we have also shown that Goose is usable for the ING use case by checking the solveability and execution time for a survey of ING specifications.

The motivation for this research started at ING. They wanted some way to verify properties for symbolic transition systems. We have presented our findings in the form of a presentation and discussed if the research in this thesis solves the main research questions for them and if the prototype is usable in their work.

ING has answered that they find that the research is relevant for their work and solves many of the research questions they had. As we currently do not allow references between specifications in Goose, the research is not yet ready to be used for every specification ING is currently working on. We do, however, have integrated the Goose prototype in ING's work and have shown that the prototype is able to solve their verification needs for specifications containing no references to other specifications. They consider the research a success.

All in all, we may answer the research questions as follows:

1. What useful properties may be verified for a STS? (chapter 4.3)

    - We have defined the verification properties: safety property and the existence of sinkholes, dead transitions, unsatisfiable relations and unsatisfiable start transitions.

2. What verification techniques & tools exist which may help to verify properties about a STS? (chapter 3)

    - We have found the existing techniques Z3 and nuXmv and have defined the new approach Goose.

3. What approach may be taken to decrease the resources needed and to increase the solvability to verify the chosen useful properties for a STS over existing techniques? (chapter 4)

    - So far we have only shown that Goose is able to verify all verification properties we have defined. We have also noted in chapter 6 that Z3 may be able to verify other verification properties. This is to be researched in future work.

4. Is this approach sound, complete, does it always terminate and what are the practical limitations? (chapter 4)

    - Goose is sound when none of the RSCs are over-approximations. When a user supplies a RSC this may be an over-approximation and some dead transitions may not be reported (false negatives) while some reachable states may be reported to violate a safety property or Goose reports that certain sinkholes or unsatisfiable relations exist (false positives). Goose is not complete as in some cases we have to ask the user to supply RSCs and not all RSCs may be verified to be correct. Goose always terminates although the SMT solver, which is used by Goose, may not terminate in all cases. Finally, it is also not able to give a trace to the violating reachable state when a counterexample for some verification property or RSC is found.

5. Using a suite of Rebel specifications commonly used by ING, how do the selected tools perform in terms of solvability and execution time to verify on an average workstation and is the execution time within the practical limit of a few seconds? (chapter 5)
    - As we have shown with the various experiments of chapter 5, Goose is able to verify all verification properties within reasonable time for the ING survey.
    - We have shown that Goose is able to solve more specifications with non-linear arithmetic of the ING survey while Z3 and nuXmv are able to verify the safety property for the fibonacci specification of chapter 5.6. Z3 and nuXmv are usually a lot faster than Goose except when we force Z3 and nuXmv to create a significant number of frames.

Appendices

Appendix A

ING Specifications

A graphical representation of the ING specifications used in the evaluation. Results for valid safety properties are shown in chapter 5.3 and result for invalid safety properties are show in chapter 5.4.

Table A.1 shows which valid safety property we have proved and which invalid safety property has been refuted for each of the specifications. Table A.2 shows which (if any) reachable state constraints have been added to verify the specification with Goose.

Table A.1: Name and reference of each specification in graphical form with the valid and invalid safety property evaluated in chapter 5

Specification	**Reference**	**Valid safety property**	**Invalid safety property**
Account Holder	A.1	$archivedAt == -1 \lor archivedAt == 100000$	$archivedAt == 2$
Bank Account	A.2	$balance \geq 0$	$balance < 1000$
Bank Account with Interest	A.3	$balance \geq 0$	$balance < 1000$
Linear Loan	A.4	$outstandingAmount \geq 0 \land remainingTerms \geq 0$	$(outstandingAmount == 0) \rightarrow (remainingTerms > 0)$
Money Movement	A.5	$balance1 + balance2 == 400 \land balance1 \geq 0 \land balance2 \geq 0$	$balance1 == 401$
Non-Interest Loan	A.6	$(outstandingAmount == 0 \rightarrow remainingTerms == 0) \land (remainingTerms == 0 \rightarrow outstandingAmount == 0) \land remainingTerms \geq 0 \land outstandingAmount \geq 0$	$(outstandingAmount == 0) \rightarrow (remainingTerms > 0)$
Transfer	A.7	$balanceFrom \geq 0 \land balanceTo \geq 0$	$balanceFrom < 100000$
Triangle of Nodes	A.8	$i \geq 0 \land i \leq 3$	$i \geq 0 \land i \leq 2$

Table A.2: Name and reference of each specification in graphical form with the reachable state constraints needed to verify the specification with Goose.

Specification	**Reference**	**Node**	**Reachable State Constraint**
Bank Account	A.2	*Open*	$balance \geq 0 \land status == OPEN$
Bank Account with Interest	A.3	*Open*	$balance \geq 0 \land status == OPEN$
Linear Loan	A.4	*Indexation*	$outstandingAmount \geq 0 \land repayment \leq outstandingAmount \land repayment \geq 0 \land remainingTerms \geq 0 \land interestRate \geq 0.05 \land interestRate < 1.0$
Money Movement	A.5	*Drafted*	$balance1 \geq 0 \land balance2 >= 0 \land balance1 + balance2 == 400$
Non-Interest Loan	A.6	*Agreement*	$(outstandingAmount == remainingTerms * repayment) \land outstandingAmount > 0 \land remainingTerms > 0$
Triangle of Nodes	A.8	*i is 1*	$i == 1$
		i is 2	$i == 2$

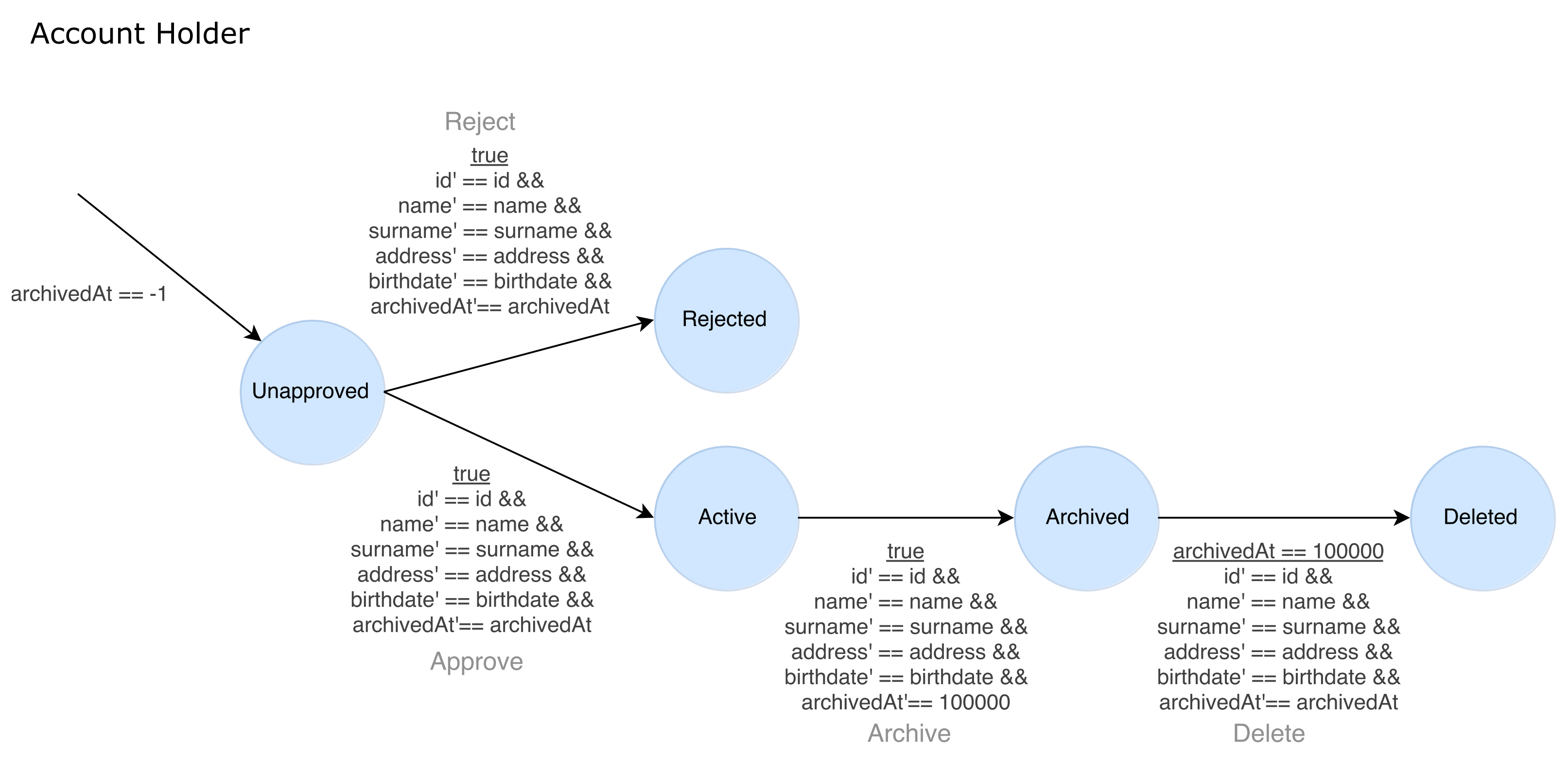


Figure A.1: STS for a client in the system.
Guards are above the line, relations below the line for transitions

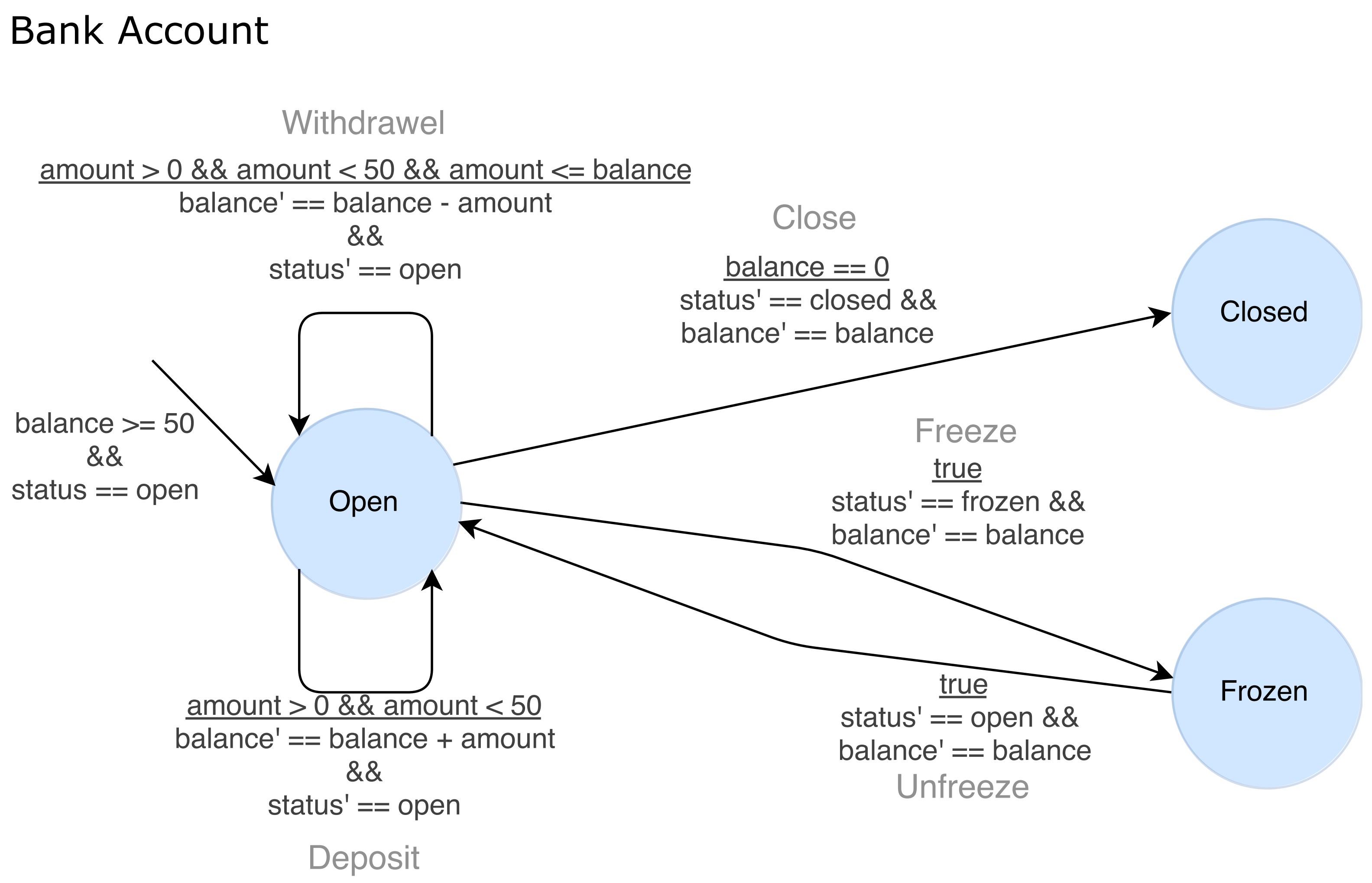


Figure A.2: STS for a bank account with withdrawal, deposit, freeze, unfreeze and close account transitions.
Guards are above the line, relations below the line for transitions

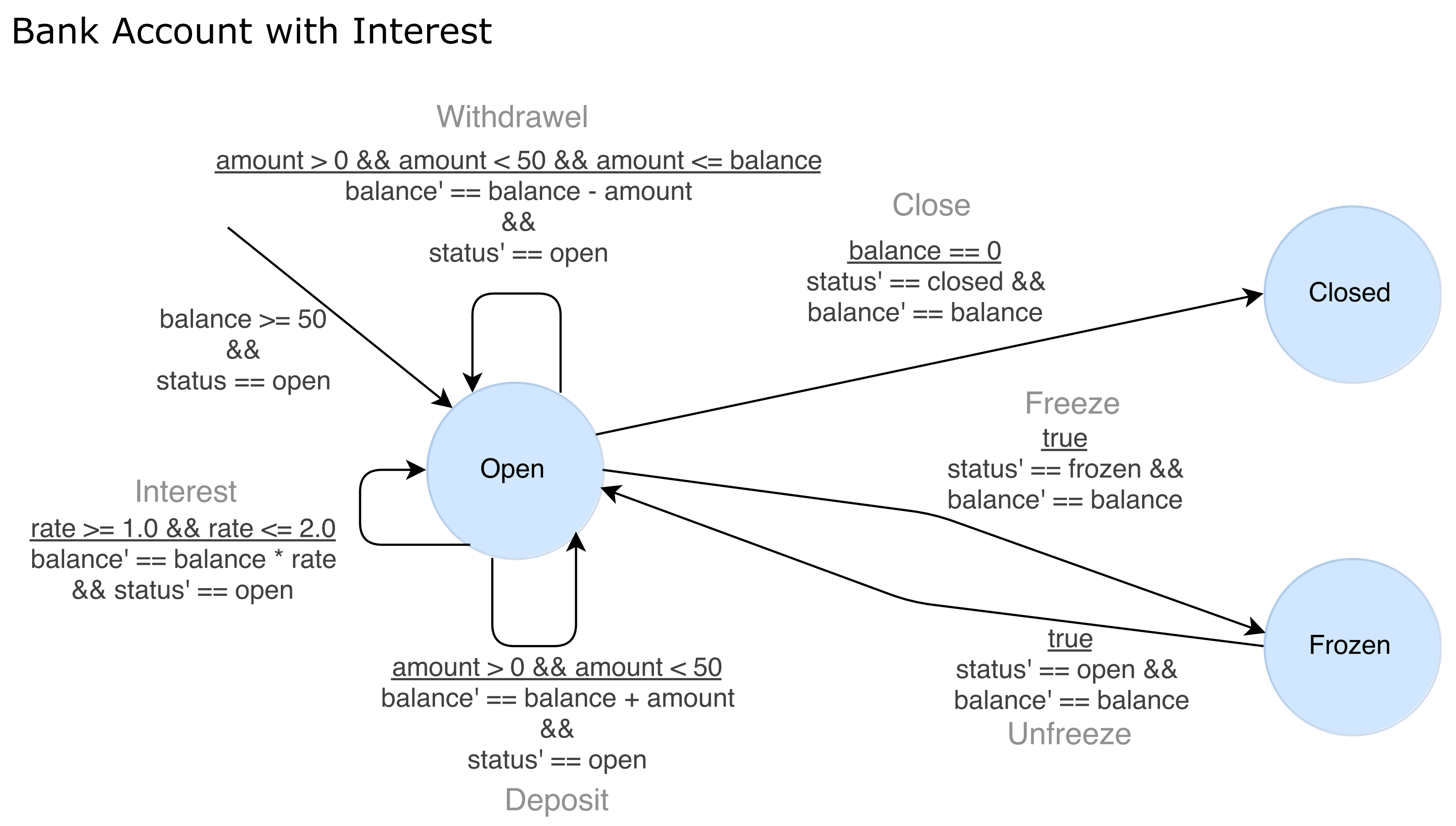
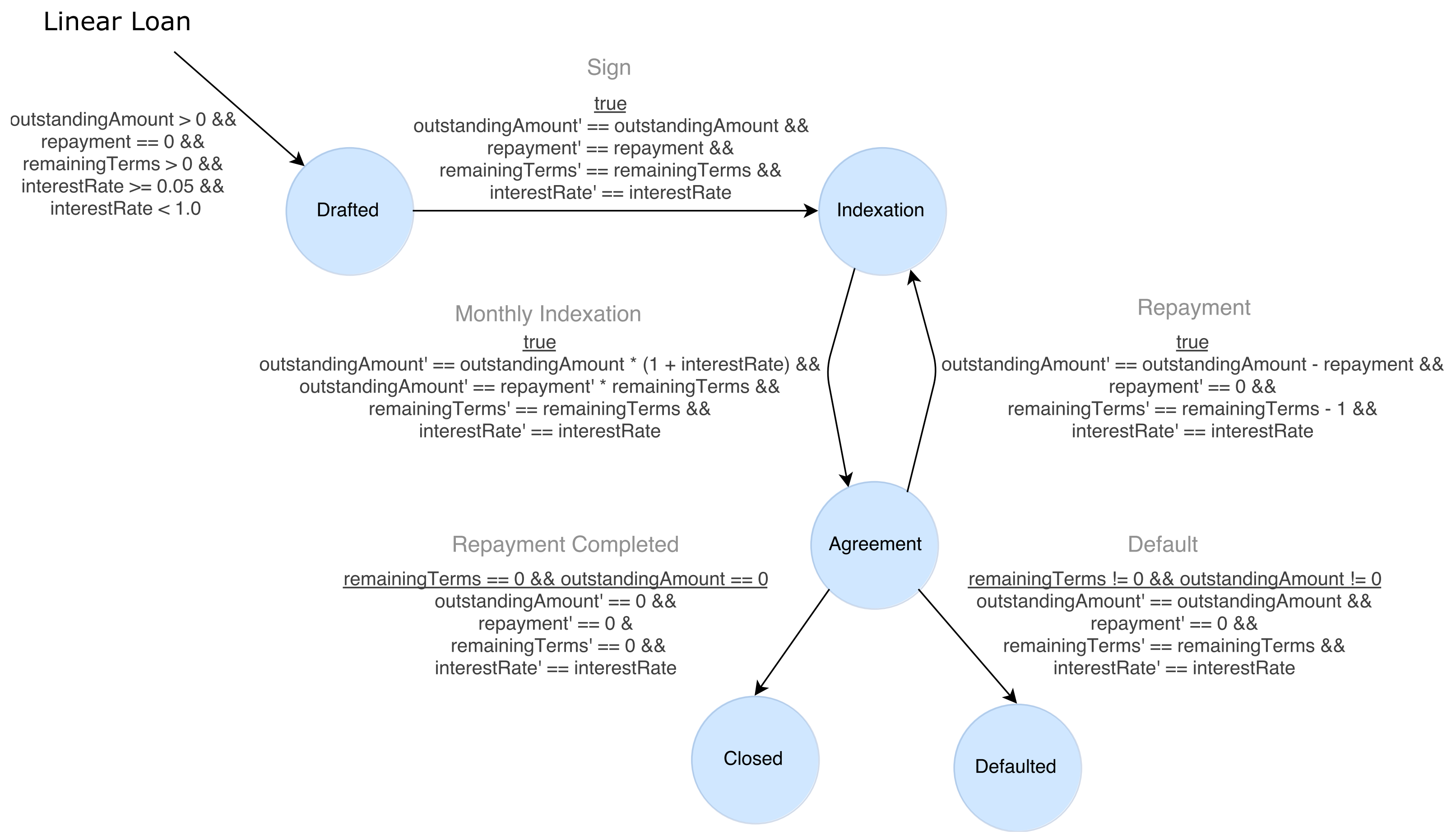


Figure A.3: STS for a bank account with withdrawal, deposit, freeze, unfreeze, close and interest account transitions.
Guards are above the line, relations below the line for transitions



Linear Loan
outstandingAmount > 0 && repayment == 0 && remainingTerms > 0 && interestRate >= 0.05 && interestRate < 1.0
Drafted
Sign
true
outstandingAmount' == outstandingAmount && repayment' == repayment && remainingTerms' == remainingTerms && interestRate' == interestRate
Indexation
Monthly Indexation
true
outstandingAmount' == outstandingAmount * (1 + interestRate) && outstandingAmount' == repayment' * remainingTerms && remainingTerms' == remainingTerms && interestRate' == interestRate
Repayment
true
outstandingAmount' == outstandingAmount - repayment && repayment' == 0 && remainingTerms' == remainingTerms - 1 && interestRate' == interestRate
Agreement
Repayment Completed
remainingTerms == 0 && outstandingAmount == 0
outstandingAmount' == 0 && repayment' == 0 & remainingTerms' == 0 && interestRate' == interestRate
Default
remainingTerms != 0 && outstandingAmount != 0
outstandingAmount' == outstandingAmount && repayment' == 0 && remainingTerms' == remainingTerms && interestRate' == interestRate
Closed
Defaulted

Figure A.4: STS for a linear loan which may be defaulted.
Guards are above the line, relations below the line for transitions

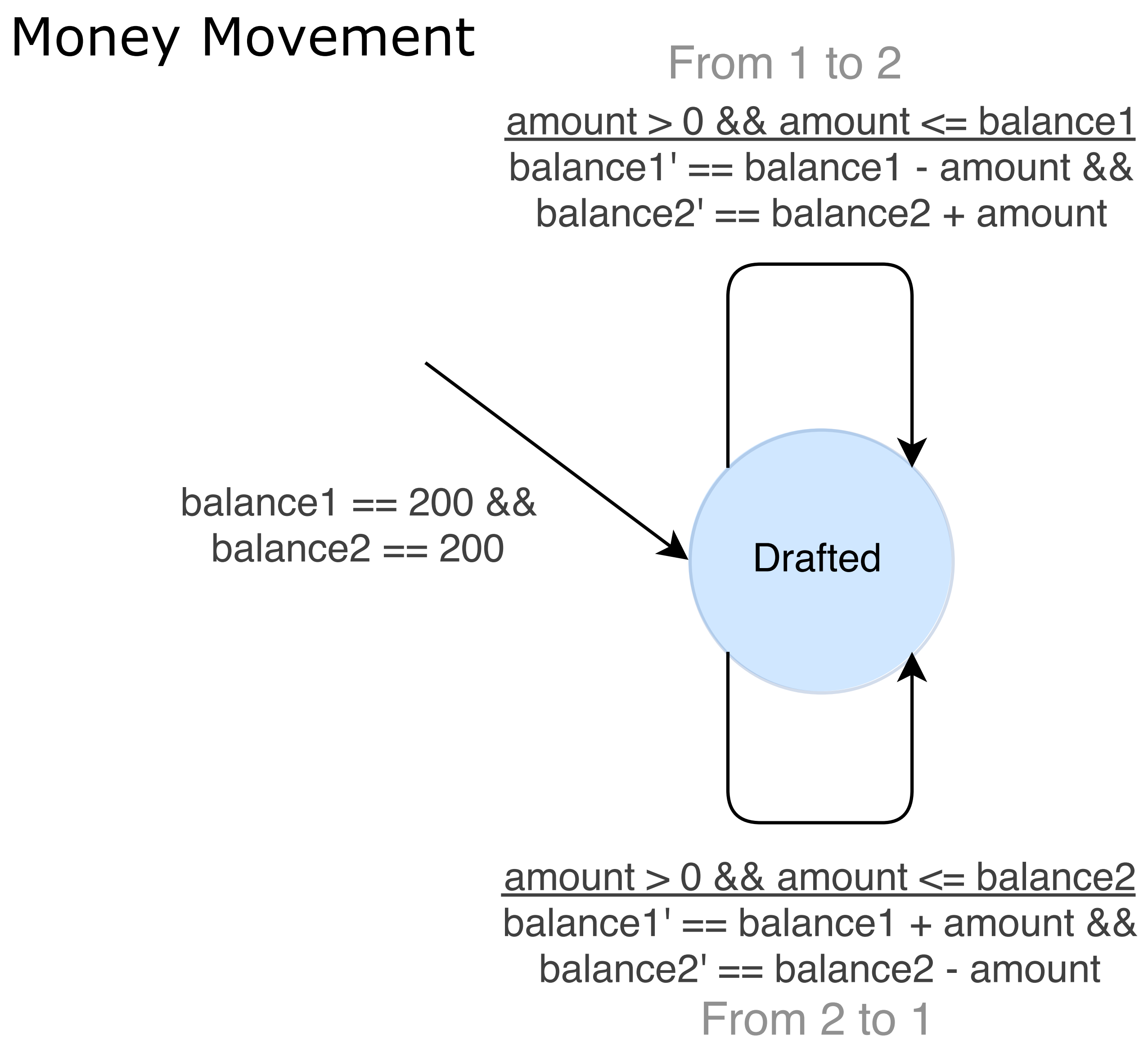

Money Movement
From 1 to 2
amount > 0 && amount <= balance1
balance1' == balance1 - amount &&
balance2' == balance2 + amount
balance1 == 200 &&
balance2 == 200
Drafted
amount > 0 && amount <= balance2
balance1' == balance1 + amount &&
balance2' == balance2 - amount
From 2 to 1

Figure A.5: STS for where a set amount of money is moved between two balances.
Guards are above the line, relations below the line for transitions

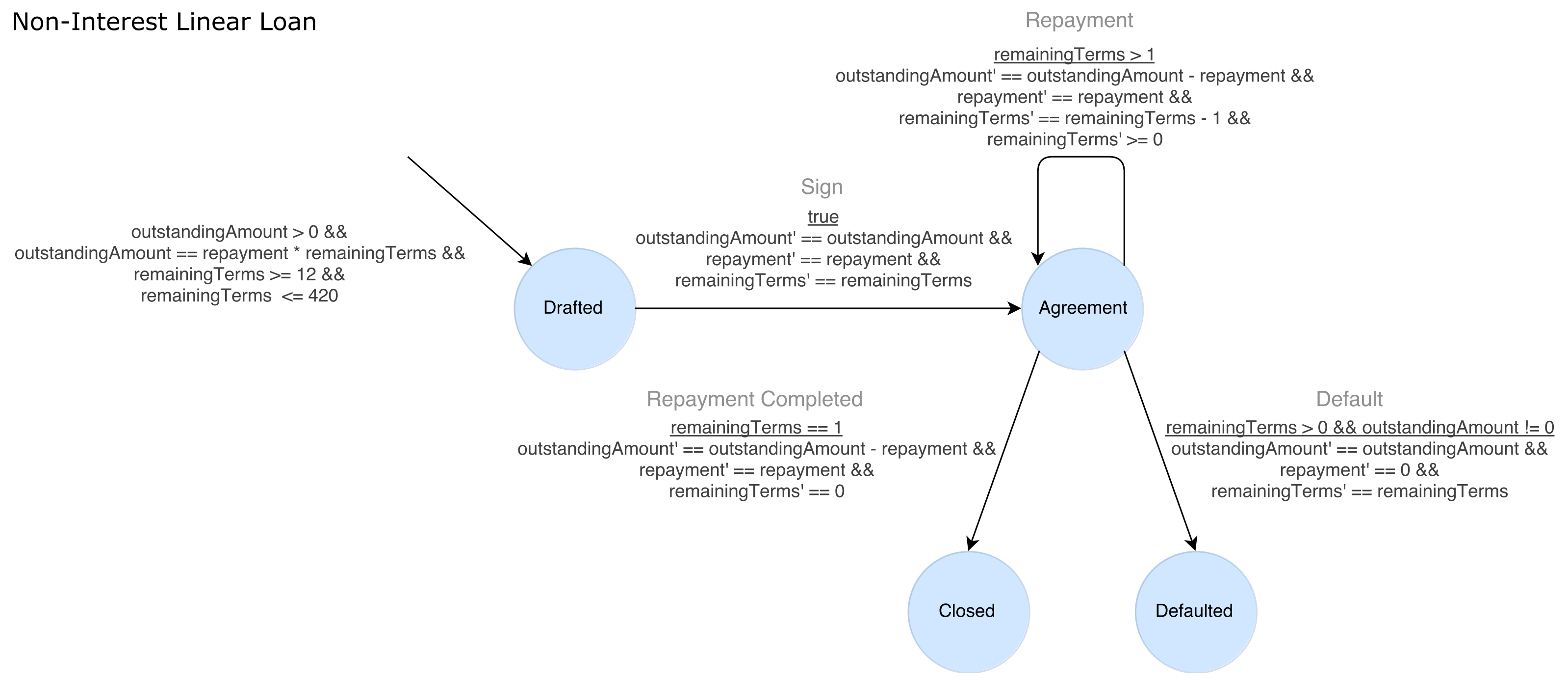

Non-Interest Linear Loan
outstandingAmount > 0 &&
outstandingAmount == repayment * remainingTerms &&
remainingTerms >= 12 &&
remainingTerms <= 420
Drafted
Sign
true
outstandingAmount' == outstandingAmount &&
repayment' == repayment &&
remainingTerms' == remainingTerms
Agreement
Repayment
remainingTerms > 1
outstandingAmount' == outstandingAmount - repayment &&
repayment' == repayment &&
remainingTerms' == remainingTerms - 1 &&
remainingTerms' >= 0
Repayment Completed
remainingTerms == 1
outstandingAmount' == outstandingAmount - repayment &&
repayment' == repayment &&
remainingTerms' == 0
Default
remainingTerms > 0 && outstandingAmount != 0
outstandingAmount' == outstandingAmount &&
repayment' == 0 &&
remainingTerms' == remainingTerms
Closed
Defaulted

Figure A.6: STS for a loan without interest which may be defaulted.
Guards are above the line, relations below the line for transitions

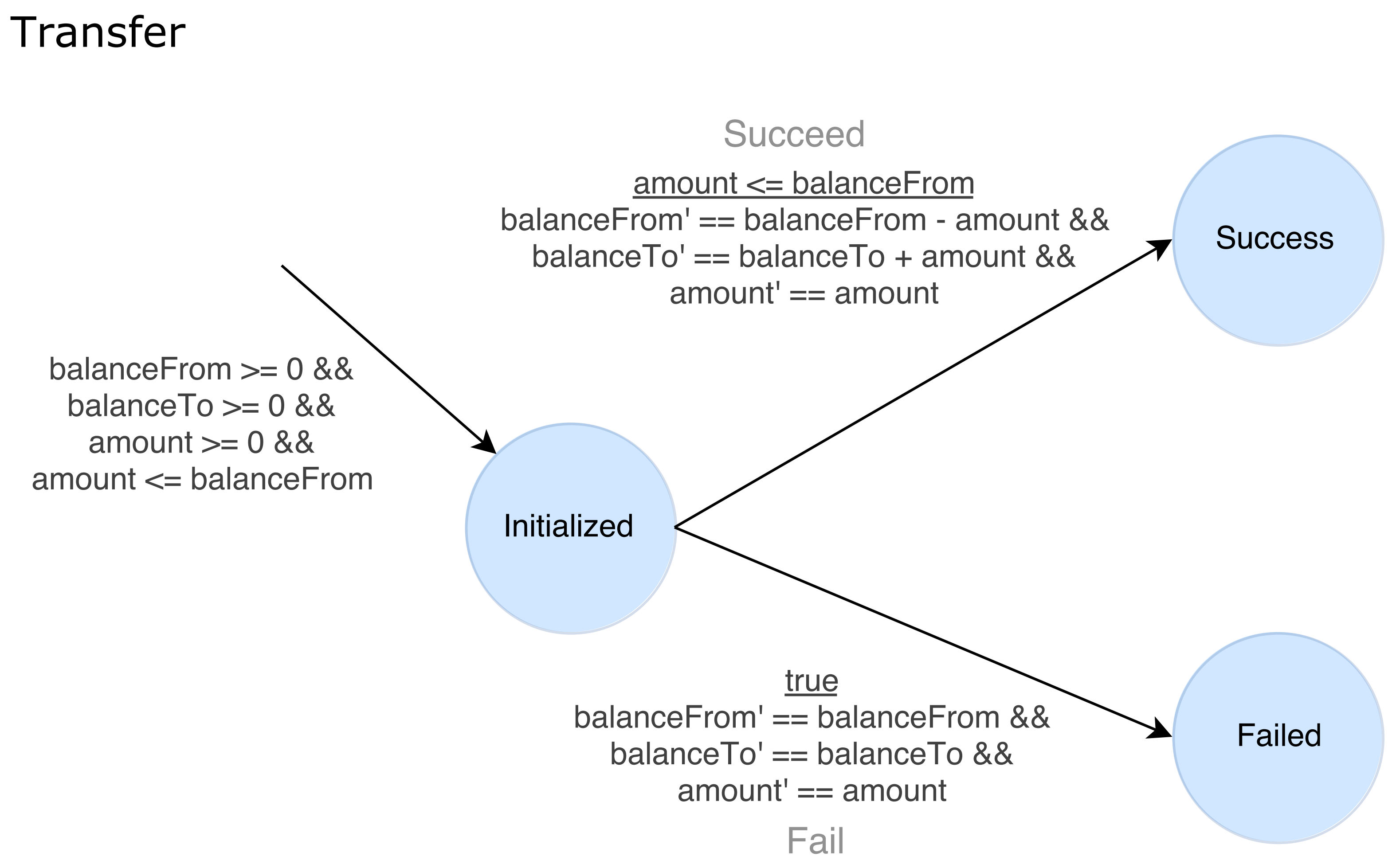


Figure A.7: STS for a single transfer between two balances.
Guards are above the line, relations below the line for transitions

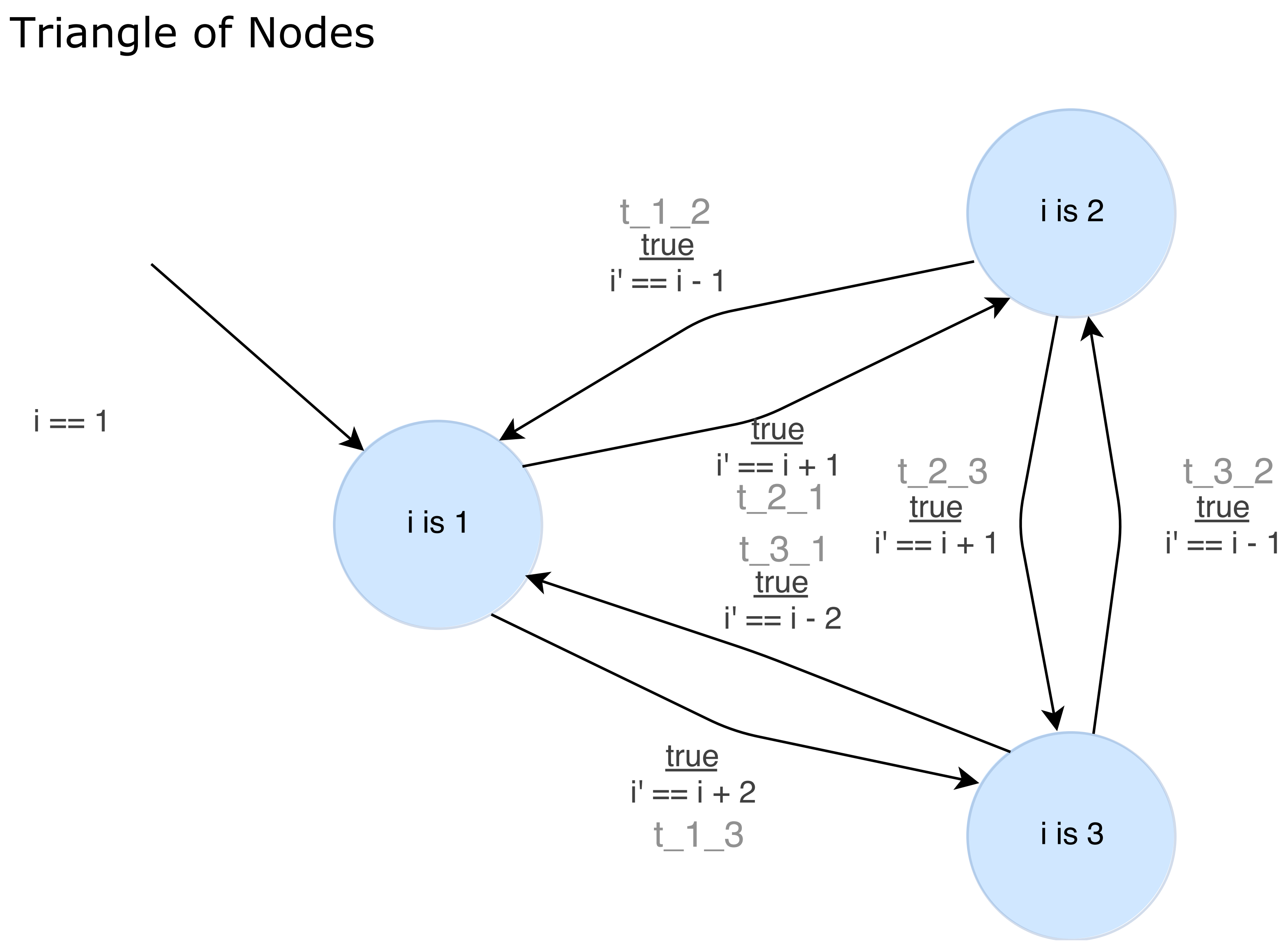


Figure A.8: STS with a complex strongly connected component.
Guards are above the line, relations below the line for transitions

Appendix B

Limited Bank Account

A graphical representation of the bank account specification (see figure B.1) of the experiment explained in chapter 5.5. The experiment uses the invalid safety property $balance < MAXIMUM$ where $MAXIMUM$ is the variable changed in the experiment to some integer $MAXIMUM > 0$. We have added the reachable state constraint $balance \geq 0 \wedge status == OPEN$ to node *Open* in order to verify the specification with Goose.

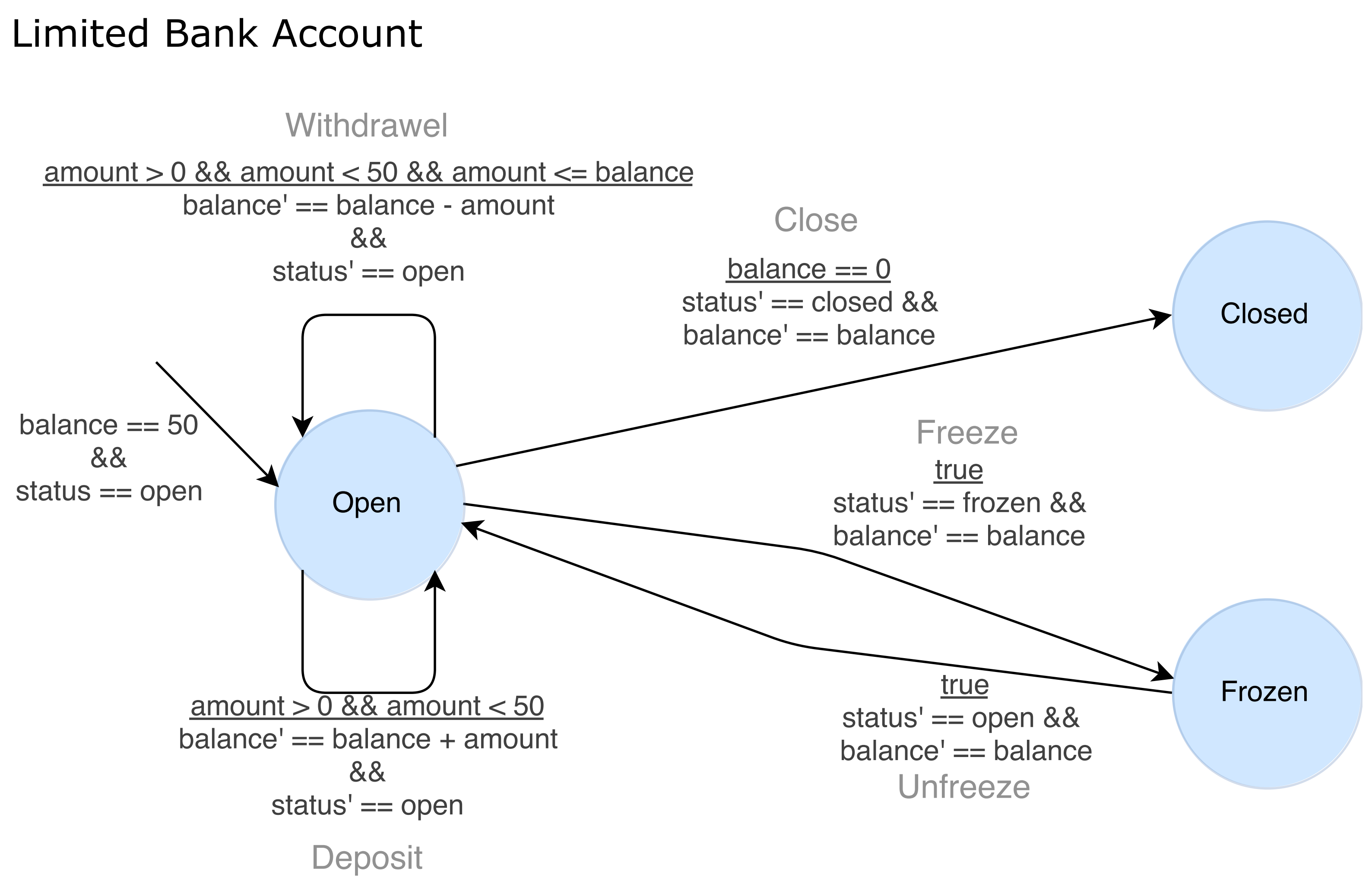


Figure B.1: STS for a bank account with withdrawal, deposit, freeze, unfreeze, close and interest account transitions where the maximum amount to deposit and withdraw is limited.
Guards are above the line, relations below the line for transitions

Appendix C

Fibonacci

A graphical representation of the fibonacci specification (see figure C.1) of the experiment explained in chapter 5.6. The experiment uses the invalid safety property $a \neq NUM$ where NUM is some constant number which should not be a fibonacci number. We have added the reachable state constraint $a == i_1 + i_2 \wedge a \geq 5 \wedge i_1 \geq 3 \wedge i_2 \geq 2$ to the node in order to try to verify the specification with Goose.

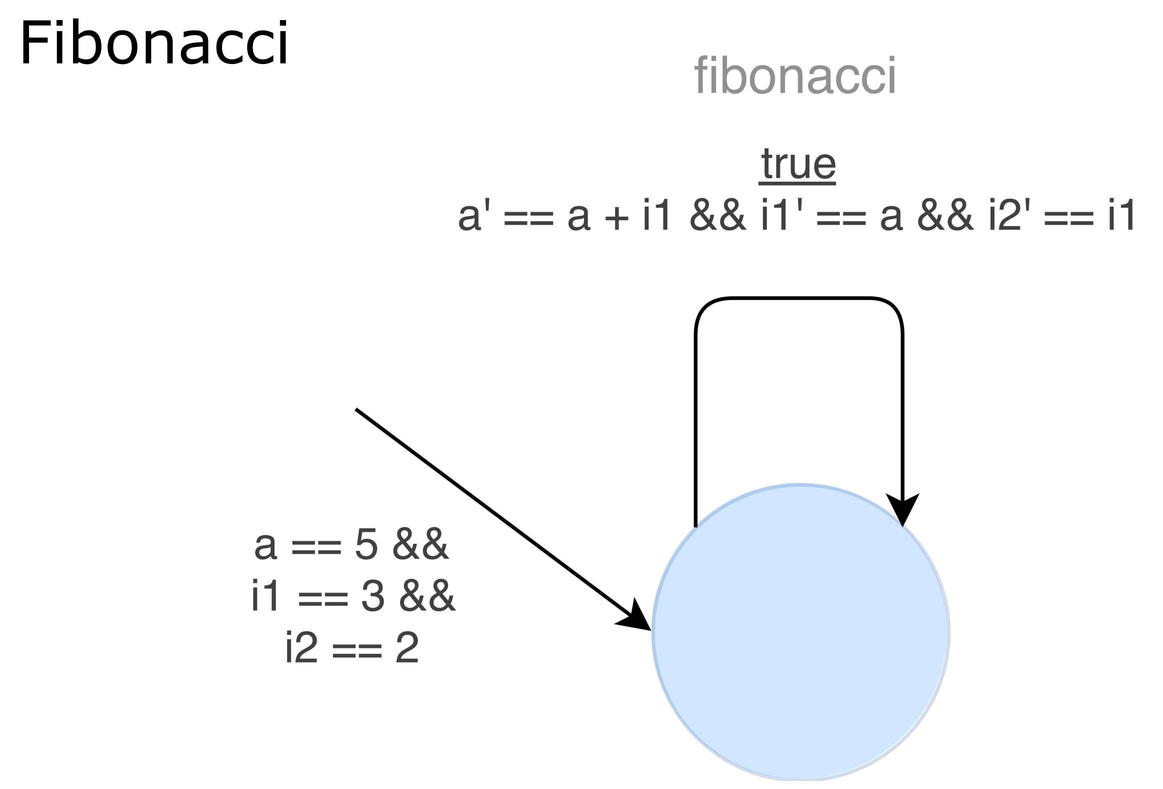


Figure C.1: STS to iteratively express the fibonacci sequence.
Guards are above the line, relations below the line for transitions

Appendix D

Specifying STS in nuXmv & Z3

How to specify an arbitrary STS in nuXmv is shown in listing D.1.

Listing D.1: Pseudo code on how to specify an arbitrary STS in nuXmv input language.

```
MODULE main

VAR node : { [foreach node]
                node.label [node isNotLast] , [end isNotLast]
              [end foreach]
             };

[foreach state variable as var]
  VAR var.label : var.type;
[end foreach]

[foreach transition variable as var]
  VAR var.label_var.transition.label_t : var.type;
[end foreach]

INIT [foreach start transition as trans]
       ( (node = trans.destination)
         & trans.relation
       ) [trans isNotLast] | [end isNotLast]
     [end foreach];

[foreach node as n]
  TRANS (node = n.label -> (
    [foreach n.leaving_transitions as trans]
      (trans.guard) & (trans.relation) |
    [end foreach]
  ));
[end foreach]
```

```


[foreach safety property as prop]
INVARSPEC (prop);
[end foreach]
```

How to specify an arbitrary STS in Z3 with the PDR strategy is shown in listing D.2.

Listing D.2: Pseudo code on how to specify an arbritary STS in Z3 PDR input language.

```
(declare-datatypes () ((Node
    [foreach node as node]
      node.label
    [end foreach]
)))

[foreach enumeration as enum]
  (declare-datatypes () ((enum.label enum.values)))
[end foreach]

; The collection of reachable, faulty states
(declare-rel faulty_states (Node
[foreach state variable as var]
  var.type
[end foreach]
))
; The collection of reachable states
(declare-rel states (Node
[foreach state variable as var]
  var.type
[end foreach]
))

(declare-var node (Node))
(declare-var next_node (Node))

[foreach state variable as var]
  (declare-var var.label (var.type))
  (declare-var next_var.label (var.type))
[end foreach]


[foreach transition variable as var]
  (declare-var var.label_var.transition.label_t (var.type))
[end foreach


(define-fun initial () Bool
  (or
    [foreach start transition as trans]
      (and (= node trans.destination) (trans.relation))
    [end foreach]
  )
)

[foreach transition as trans]
  (define-fun trans.label () Bool
    (and
      (= node trans.origin)
      (= next_node trans.destination)
      (trans.guard)
      (trans.relation)
    )
  )
[end foreach]

; Any of the following transitions
(define-fun transition () Bool
  (or
    [foreach transition as trans]
      trans.label
    [end foreach]
  )
)

(define-fun safety_property () Bool
  (and
    [foreach safety property as prop]
      prop
    [end foreach]
  )
)

; Any state satisfying the initial function is a reachable state
(rule (=> initial (states node
[foreach state variable as var]
  var.label
[end foreach]
)))

; For any origin state in reachable states, any destination state
; that satisfies any transition is in reachable states
(rule
  (=>
    (and
      (states node
        [foreach state variable as var]
          var.label
        [end foreach]
      )
      transition
    )
    (states next_node
      [foreach state variable as var]
        next_var.label
      [end foreach]
    )
  )
)

; Any reachable state that counters the safety property
; is a faulty state
(rule
  (=>
    (and
      (states node
        [foreach state variable as var]
          var.label
        [end foreach]
      )
      (not safety_property)
    )
    (faulty_states node
        [foreach state variable as var]
          var.label
        [end foreach]
    )
  )
)

; Ask for all faulty states
; Should be empty if safety property is preserved
(query faulty_states)
```

Bibliography

[1] Scala. http://www.scala-lang.org/.

[2] C Operator Precedence. http://en.cppreference.com/w/c/language/operator_precedence, 2017.

[3] Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina. SAFARI: SMT-based abstraction for arrays with interpolants. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7358 LNCS, pages 679–685, 2012.

[4] Francesco Alberti, Silvio Ghilardi, and Natasha Sharygina. A framework for the verification of parameterized infinite-state systems. *Fundamenta Informaticae*, 150(1):1–24, 2014.

[5] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[6] Kai Bakker. *Reachable States for Symbolic Transition Systems.* PhD thesis, University of Amsterdam, 2015.

[7] Clark Barrett, Aaron Stump, and Seshia Cesare Tinelli. The SMT-LIB Standard Version 2.0. *8th International Workshop on Satisfiability Modulo Theories*, page 85, 2010.

[8] Mordechai Ben-Ari. *Mathematical Logic for Computer Science*, volume 2. Springer, 2012.

[9] Armin Biere, Alessandro Cimatti, Edmund Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, 2003.

[10] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.

[11] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Infinite-state invariant checking with IC3 and predicate abstraction. *Formal Methods in System Design*, 49(3):190–218, 2016.

[12] Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms-3rd Edition.* 3 edition, 2009.

[13] Martin Davis. Hilbert's Tenth Problem is Unsolvable. *The American Mathematical Monthly*, 80(3):233, 1973.

[14] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT Solver. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 4963 LNCS, pages 337–340, 2008.

[15] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. Software Verification Using k-Induction. In *SAS 2011: Static Analysis*, pages 351–368, 2011.

[16] Bruno Dutertre and Leonardo de Moura. The YICES SMT Solver. *Computer science lab SRI*, pages 1–5, 2006.

[17] Harold N. Gabow. Path-based depth-first search for strong and biconnected components. *Information Processing Letters*, 74(3):107–114, 2000.

[18] Silvio Ghilardi, Enrica Nicolini, Silvio Ranise, and Daniele Zucchelli. Towards SMT model checking of array-based systems. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5195 LNAI(Section 5):67–82, 2008.

[19] Krystof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 7317 LNCS, pages 157–171, 2012.

[20] Julian Huijbregts. Tax Authority concludes that the collection of taxes is in jeopardy due to IT-issues. https://tweakers.net/nieuws/128493/belastingdienst-concludeert-dat-innen-belasting-gevaar-loopt-door-ict-problemen.html, 2017.

[21] K.L. McMillan. Interpolation and SAT-Based Model Checking. In *International Conference on Computer Aided Verification*, 2003.

[22] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*, volume 13. Cambridge University Press, 1999.

[23] Jan Ouwens. Mortgage DSL at the Rabobank. http://www.nljug.org/databasejava/een-dsl-kweken/, 2016.

[24] Jouke Stoel, Tijs Van Der Storm, Jurgen Vinju, and Joost Bosman. Solving the Bank with Rebel. 3(Industry Track for Software Language Engineering):13–20, 2016.

[25] Thomas A. Sudkamp. *Languages and machines: An introduction to the theory of computer science*, volume 13. Addison Wesly, 3 edition, 2005.

[26] Machiel Van der Bijl, Arend Rensink, Jan Tretmans, and Machiel Van Der Bijl. Compositional testing with ioco. *... Approaches to Software Testing*, pages 86–100, 2004.

[27] Thomas Wahl. The k-Induction Principle. *Northeastern University, College of Computer and Information Science*, pages 1–2, 2013.