

# Checking universally quantified temporal properties with three-valued analysis

Björn Wachter

**Diplomarbeit**

bei Professor Reinhard Wilhelm

Betreuung: Jörg Bauer

Universität des Saarlandes

2005



## ERKLÄRUNG

Hiermit erkläre ich, Björn Wachter, an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen verwendet habe.

Saarbrücken, 24. Februar 2005



## ABSTRACT

Establishing correctness of systems with replicated components, UML models, and heap-manipulating programs requires showing universally quantified temporal properties. We present a symbolic quantifier elimination which is independent of a particular abstract representation of states. Based on the symbolic quantifier elimination, we give a three-valued logical analysis for quantified temporal properties. We compare the analysis with a method that employs finite instantiation and data type reduction. For this purpose, we have conducted a case study using the tools TVLA and SMV. Finite instantiation relies on symmetry arguments. We discuss how the framework for three-valued logical analysis relates to symmetry.

## ZUSAMMENFASSUNG

Um Korrektheit von Systemen mit replizierten Komponenten, UML-Modellen und heap-manipulierenden Programmen zu zeigen, benötigt man universell quantifizierte temporale Eigenschaften. Wir stellen eine symbolische Quantorelimination vor, die unabhängig von einer speziellen Representation der abstrakten Zustände ist. Wir präsentieren eine dreiwertige logische Analyse für quantifizierte temporale Eigenschaften, die auf dieser symbolischen Quantorelimination beruht. Wir vergleichen die Analyse mit einer Methode, die auf endlicher Instanziierung und Datentypreduktion aufbaut. Zu diesem Zweck haben wir eine Fallstudie mit den Verifikationsprogrammen TVLA und SMV durchgeführt. Endliche Instanziierung beruht auf Symmetrieargumenten. Wir erläutern die Beziehung des Frameworks für dreiwertige logische Analyse zur Symmetrie.



## ACKNOWLEDGEMENTS

Ich hatte grosses Glück Jörg Bauer als Betreuer zu haben. Sein Enthusiasmus und das Interesse, das er an meiner Arbeit zeigte, waren immer wieder inspirierend für mich. Professor Wilhelm danke ich für das spannende Thema, sein Vertrauen in meine Fähigkeiten und die Chancen, die er mir gibt, mich als Forscher zu entwickeln.

Diskussionen mit meinem Kollegen Jan Reineke verdanke ich viele Anregungen zur Gestaltung der Arbeit und neue Sichtweisen jenseits davon. Ich will auch meine sehr guten Freunde Lijun Zhang und Andreas Meyer nicht vergessen. Lijuns Geradlinigkeit und sein Sinn für das Praktische haben mich stark beeinflusst. Andreas wurde am Ende meiner Studienzeiten ein enger Freund und geschätzter Kooperationspartner bei Projekten. Auch jetzt half er mir tatkräftig und las meine Diplomarbeit.

Ich bin dankbar für die Förderung durch die Deutsche Forschungsgemeinschaft (DFG) im Rahmen des Projektes AVACS (DFG Transregio-Sonderforschungsbereich 14, Teilprojekt S2). Die Kooperation innerhalb von S2 hat mich sehr vorangebracht. Besonders Bernd Westphal gab mir Denkanstösse während des letzten AVACS Workshops und kommentierte frühe Entwürfe meiner Arbeit.

Die Mitarbeiterinnen und Mitarbeiter am Lehrstuhl Wilhelm haben mir immer wieder bei kleineren Problemen geholfen.

Diese Aufzählung wäre unvollständig, würden darin nicht meine Eltern erwähnt. Ihre emotionale Unterstützung kann man nicht hoch genug einschätzen. Darüberhinaus gab mir mein Vater die Möglichkeit mich ganz meinem Studium zu widmen. Ohne sein Zutun hätte es diese Arbeit nicht geben können.





# Contents

<b>1</b>	<b>Introduction</b>	<b>11</b>
1.1	Verification Problem . . . . .	11
1.2	Existing Techniques . . . . .	13
1.2.1	Three-valued Logical Analysis . . . . .	13
1.2.2	Finite instantiation and data type reduction . . . . .	13
1.2.3	Predicate Abstraction . . . . .	14
1.3	Summary of the thesis . . . . .	15
1.4	Results . . . . .	17
1.5	Overview of the thesis . . . . .	18
<b>2</b>	<b>Models</b>	<b>19</b>
2.1	First-Order Logic . . . . .	19
2.1.1	Syntax . . . . .	19
2.1.2	Semantics . . . . .	21
2.2	Syntax and Semantics of Models . . . . .	22
2.3	Predicate Logic . . . . .	23
2.4	Example . . . . .	24
2.5	Discussion . . . . .	25
<b>3</b>	<b>Properties</b>	<b>28</b>
3.1	Syntax . . . . .	29
3.2	Semantics . . . . .	30
3.3	Discussion . . . . .	32
<b>4</b>	<b>Quantifier Elimination</b>	<b>34</b>
4.1	Skolemization . . . . .	35
4.2	Predicate Logic Skolemization . . . . .	38
4.3	Discussion . . . . .	41
<b>5</b>	<b>Analysis</b>	<b>44</b>
5.1	Three-Valued Analysis . . . . .	44
5.2	Implementation . . . . .	50
5.3	Case Study . . . . .	55
5.4	Related 3-valued analyses . . . . .	58
5.5	Discussion . . . . .	63

<b>6</b>	<b>Symmetry</b>	<b>65</b>
6.1	Intuition. . . . .	65
6.2	Canonical Abstraction and Symmetry . . . . .	67
<b>7</b>	<b>Finite Instantiation and Data Type Reduction</b>	<b>70</b>
7.1	Case Study: SMV . . . . .	73
<b>8</b>	<b>Conclusion</b>	<b>77</b>
8.1	Future Work . . . . .	77
<b>A</b>	<b>Extended Embedding Order</b>	<b>83</b>
<b>B</b>	<b>Proofs</b>	<b>89</b>
B.1	Galois connection induced by representation function . . . . .	89
B.2	Skolemization for predicate logic . . . . .	89
B.3	Preservation by Simulation . . . . .	92
B.4	Extended Embedding . . . . .	95
B.5	Symmetry Lemma . . . . .	96
<b>C</b>	<b>Sources</b>	<b>99</b>
C.1	Case Study with TVLA . . . . .	99
C.2	Case Study with SMV . . . . .	102

Table of meta-variables	
$K = \langle S, I, R \rangle \in \mathcal{K}_S$	transition system over state space $S$
$S$	set of states
$I$	set of initial states
$R$	transition relation
$\Sigma = \langle \mathcal{B}, \mathcal{F}, P, \mathcal{V}, r \rangle$	signature
$T \in \mathcal{B}$	base type
$f \in \mathcal{F}$	function symbol
$p \in P$	predicate
$x \in \mathcal{V}$	variable
$r$	rank function
$s \in Struct[\Sigma, \Delta]$	logical structure
$\Delta(T)$	semantic domain of a base type $T$
$e \in \mathcal{FO}_\Sigma$	expression over signature $\Sigma$
$M = \langle S, \theta, \rho \rangle \in \mathcal{M}_\Sigma$	model
$\theta \in \mathcal{FO}_\Sigma$	expression denoting the initial states of a model
$\rho \in \mathcal{FO}_{\Sigma \cup \Sigma'}$	expression denoting the transitions of a model
$\mathfrak{r} \in \mathfrak{R}$	compatibility constraint
<hr/>	
$\mathcal{P} = \langle P, \mathcal{V}, r \rangle$	signature of predicate logic
$s \in 3Struct[\mathcal{P}]$	three-valued logical structure
<hr/>	
$\phi$	FCTL* state formula
$\Phi$	FCTL* path formula
<hr/>	
$\zeta$	Skolem constant
$\eta$	Skolem predicate

# Chapter 1

## Introduction

Systems which manipulate unbounded data and have a large number of components are ubiquitous. They occur in the shape of software, or dynamic communicating systems. Dynamic communicating systems consist of a collection of entities that interact via communication links. They are dynamic in that communication links can change, and entities can be created or destroyed. Examples are traffic guidance systems and ad-hoc networks. Formal methods are needed to guarantee correctness of designs, in order to avoid accidents, data loss, or security breaches.

### 1.1 Verification Problem

Formal verification methods provide means for verifying a model of a system against a property. Both models and properties are denoted in an unambiguous formal description language and have a precise semantics. Using formal verification, one can *guarantee* that a system shows exactly the behavior specified by the property. Testing or simulation can only partially explore the behavior of a program and can only consider a restricted set of inputs. This is sufficient to find some, but not necessarily all, bugs, however, it is *insufficient* to ensure that a system meets the specification given by the property. We assume that models are denoted in first-order logic. Their semantics are first-order transition systems, i.e. transition systems with a state space consisting of first-order structures. Temporal properties incorporate the aspect of time into the specification. This is necessary in order to express the reactions of a system to stimuli: *when  $x$  happens then after  $k$  steps  $y$  will happen*, liveness properties: *something good will happen eventually*, and safety properties: *something bad will never happen*. The semantics of a temporal property is a set of logical structures in a transition system.

We are interested in verifying systems with infinite data domains and an unbounded number of processes. A simple example for such a system is the Ticket Protocol depicted in Figure 1.1. Processes compete for a resource that can only be used by one process at a time. The goal of the Ticket Protocol is to make sure that no two processes ever become critical simultaneously, i.e. use the resource simultaneously. Mutual exclusion is ensured with counter variables. Counter values are called tickets. Each process has a ticket, which is modeled by the local process variable  $a$ . There is always a winner ticket, the value of global variable  $s$ ; global variable  $t$  holds a fresh ticket that is by one larger

than the greatest ticket issued so far. Formally, mutual exclusion can be expressed by the quantified safety property, *for all processes  $p, q$  holds invariantly that if  $p$  and  $q$  are critical they must be identical*, formally this can be expressed by the quantified temporal property  $\phi_{MUTEX} = \forall p. \forall q. \mathbf{AG}(at(p) = crit \wedge at(q) = crit \Rightarrow p = q)$ .  $p$  and  $q$  stand for process indices.  $\mathbf{AG}$  means that the property is a safety property and  $at(p) = crit$  means that process  $p$  is in its critical section.

Unbounded systems often require quantified temporal properties, i.e. temporal properties with first-order quantification in front. One obtains an instance of  $\phi_{MUTEX}$  by inserting symbols or values, e.g. if 0 and 1 are process indices, the temporal property  $\phi_{MUTEX}[0][1] = \mathbf{AG}(at(0) = crit \wedge at(1) = crit \Rightarrow 0 = 1)$  is an instance. A quantifier elimination technique will be presented which instantiates quantified formulas with symbols. Generally, we consider properties where the first-order quantification domain is unbounded. Therefore, there is an unbounded number of instances to be shown.

Program TICKET	Program ADD
<pre> int s = 0,     t = 0;  void P[i] () {   int a = 0    while(true) {     think: atomic{a=t; t=t+1;}     wait:  if(a==s) {     crit:   ...            s=s+1           }         } } </pre>	<pre> input int x; input int y; output int z; while(true)   z=x+y </pre>

Figure 1.1: Running examples. The program on the right is called the *Ticket Protocol*. The index  $i$  in  $P[i]$  ranges over an unbounded domain of process indices. The Ticket Protocol is the parallel composition of the processes  $P[i]$ . Variable  $s$  stands for the *winner ticket*, and variable  $t$  contains a fresh ticket. Local process variable  $a$  contains the ticket of the corresponding process. The program on the right constantly adds its inputs  $x, y$  and returns the result in  $z$ . Using our method, we showed mutual exclusion for the Ticket Protocol. Except for the modeling, the process of verification was completely automatic.

## 1.2 Existing Techniques

In the last decade, much attention has been given to modeling and analyzing systems with infinite data domains; e.g. [SRW02, SS99, BPR01, McM00, MQS00, NNH99, GHJ01]. Formal methods, and static analysis in particular, play a key role in verifying such systems. Static analysis explores the states of a program in *all* possible situations without actually running it. Instead, the program is run on abstract descriptors that represent collections of states. Tools based on static analysis have been successfully used to verify systems and find bugs [BCR, SRW<sup>+</sup>, McMb, HM, CCG].

### 1.2.1 Three-valued Logical Analysis

In three-valued analysis, the abstract descriptors are bounded three-valued structures. A three-valued structure is a logical structure where predicates can evaluate to a third truth value  $1/2$  which stands for *unknown*. Boundedness means that the universes of those structures have bounded size. The static analysis framework [SRW02], implemented in the tool TVLA [SRW<sup>+</sup>], allows one to show and discover quantified invariants of both heap-manipulating programs *and* systems with an unbounded number of components. Quantified *temporal* verification, though, requires additional machinery. Verification of general first-order temporal properties has been addressed in [YRSW03]; a temporal first-order linear-time logic, called ETL, and a static analysis for verifying ETL properties were given. [YR04] describes an analysis based on decomposition of quantified safety properties. First-order safety properties are safety properties with first-order quantifiers in front. More general temporal properties are not considered.

### 1.2.2 Finite instantiation and data type reduction

Quantified temporal properties naturally occur in hardware verification; e.g. one wants to algebraically specify the outputs in terms of the inputs and the number of steps after which the result is available *for all input values*. [McM00] presents a methodology for hardware verification, implemented in the tool SMV [McMb], based on compositional reasoning and finite instantiation. The methodology is compositional in two senses: a verification problem is decomposed into subproblems with finite instantiation, and spurious counterexamples (cf. SMV tutorial [McMb]) are removed by compositional reasoning. Quantifiers are eliminated by finding a finite number of sufficient instances of the quantified property; this is called *finite instantiation*. Instances can be found automatically if model and specification fulfill certain syntactic criteria (scalarset criteria [ID96]). Symmetry means that data values can be permuted while preserving system behavior and properties of interest. For example, one can exchange the roles of the processes in the Ticket Protocol without changing the behavior of the program. Symmetry arguments are used for finding a finite sufficient number of instances of a quantified property in [McM00]. For the mutual exclusion property of the Ticket Protocol, the instances  $\phi_{MUTEX}[0][1]$  are  $\phi_{MUTEX}[0][0]$  are sufficient. This is because process indices can only be compared using equality and no arithmetic operations are allowed.

Data type reduction is parameterized by a set of values which are to be kept material, all other values are collapsed and all information about them is discarded. Data type

reduction is used in combination with finite instantiation. The values selected by decomposition are kept material. For example, when  $\phi_{MUTEX}[0][0]$  is verified the process indices are abstracted to an abstract process index which stands for 0 and one which stands for all process indices but 0.

UML models are another example for systems with large data domains and an unbounded number of components. [DH01] describes Live Sequence Charts, a specification formalism for UML models. Instance lines of sequence charts have the semantics of universal quantification [DPJ03]. [DW03] proposes to use the compositional methodology [McM00] for the verification of UML models against Live Sequence Chart specifications.

### 1.2.3 Predicate Abstraction

Predicate abstraction is used to compute a finite-state abstraction of a program. It is parameterized by a set of properties (*predicates*). A program is mapped to an abstract program, a so-called Boolean program [BPR01], which manipulates Boolean variables corresponding to the predicates. Automated reachability analyses and invariant checking for sequential C code (and concurrent programs with a bounded and fixed number of processes) have been addressed by predicate abstraction, e.g., [HM, BCR, CCG, MPC<sup>+</sup>]. However, these techniques cannot handle systems with an unbounded number of replicated components. Establishing correctness of such systems requires discovering quantified invariants and verifying quantified temporal properties. Classical predicate abstraction can only handle quantifier-free problems. There is a method [Lah04] which uses indexed predicates and can find quantified invariants of unbounded systems.

### 1.3 Summary of the thesis

We formulate models using first-order logic with function symbols, equality *and* transitive closure. The framework for three-valued analysis [SRW02] makes use of a predicate logic. This logic can be obtained by considering the sublogic *without function symbols* of our general first-order logic. The logic used in [DW03] lacks transitive closure. Apart from that, our models strongly resemble symbolic transition systems of [DPJ03, DW03]. The semantics of a model  $M$  is a first-order transition system  $K$ . If the states of a transition system have the same universe, it is termed a constant-domain transition system. A first-order transition system whose states can have different universes is called a varying-domain transition system. The framework for three-valued analysis [SRW02] considers first-order transition systems with varying domains. We give models a constant-domain semantics.

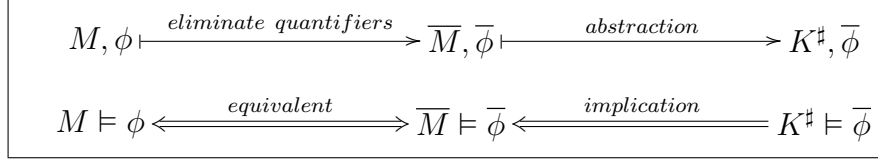
The constant-domain semantics has the advantage that quantified properties become more expressive (cf. Section 3.3). We specify properties in a branching temporal logic augmented with first-order quantification. A model fulfills a property if all of its initial states fulfill the property. The semantics of first-order properties is straightforward in a constant-domain setting. First-order quantification is handled by using environments which are passed into first-order expressions nested in the temporal property. The semantics of temporal operators is standard and taken from [CGP00]. Property preservation is ensured by a simulation preorder on first-order transition systems.

We want to get rid of quantifiers in front of temporal formulas. Skolemization is a quantifier elimination technique which syntactically transforms a model and a property. It is completely independent of an ensuing abstraction step and does not rely on a particular representation of abstract states. The quantifier elimination is not based on symmetry arguments; it originates from theorem proving, see e.g. [Ber02, OS03]. The idea is to instantiate a quantifier with a fresh uninterpreted symbol which is introduced into the model. The symbol has a free initial valuation and the valuation does not change over time. For example, the mutual exclusion property of the Ticket Protocol is transformed to the property  $\bar{\phi}_{MUTEX} = \mathbf{AG}(at(\zeta_1) = crit \wedge at(\zeta_2) = crit \Rightarrow \zeta_1 = \zeta_2)$ . We have instantiated two quantifiers with fresh symbols  $\zeta_1, \zeta_2$ . Predicate logic is a sublogic of the general first-order logic with function symbols. Skolemization introduces function symbols. So Skolemization applies to models denoted in predicate logic but it produces a model *with* function symbols. One has to adapt Skolemization to predicate logic and obtains similar syntactical transformation as in the work about decomposing first-order safety properties [YR04] (for more details see Section 5.4).

Combining predicate logic Skolemization with three-valued analysis yields a method for verifying models denoted in predicate logic against quantified temporal properties. The method is staged: it consist of a quantifier elimination step and an abstraction step. The abstraction is done by three-valued logical analysis [SRW02]. Abstraction makes the verification problem tractable at the cost of possible false alarms (also called *spurious counterexamples*). We are given a model described in predicate logic, denoted as  $M$  and a property  $\phi$ . Skolemization is a syntactical transformation on concrete models, and produces a model  $\bar{M}$  and a property  $\bar{\phi}$ . Skolemization guarantees strong preservation (symbolized by  $\Leftrightarrow$ ), i.e. we obtain an equivalent verification problem. An abstract transition system  $K^\#$  is computed by running the model  $\bar{M}$  on abstract descriptors. Canonical



abstraction [SRW02] ensures termination of the exploration. The figure is to be read from left to right and depicts the stages of the method.



The quantifier elimination introduces symbols. These symbols appear in properties resulting from Skolemization, e.g.  $\overline{\phi}_{MUTEX}$ . It is a straightforward idea to adapt the abstraction to these symbols. The individuals referred to by the Skolem symbols and individuals transitively reachable from those individuals should be distinguished from the other individuals in order to increase precision. This idea is called heterogeneous abstraction in [YR04]. We used our analysis to prove mutual exclusion for the Ticket Protocol. Additional lemmas or instrumentation predicates were not necessary.

As we want to compare three-valued analysis with finite instantiation and data type reduction, it is interesting to see whether the syntactic restrictions required for finite instantiation also hold for models of the three-valued analysis framework [SRW02]. Indeed, *models of [SRW02] are per se fully symmetric* in a similar way as expressed in [ID96] (except that the former approach uses a varying-domain and the latter a constant-domain semantics). This coincides with the idea of a heap semantics where isomorphic<sup>1</sup> structures are not distinguished because individuals are *anonymous*. The reasons for symmetry lie in the predicate logic of [SRW02]. It cannot distinguish between isomorphic structures. There is exactly one symbol with a fixed valuation: the equality symbol. Equality is preserved under permutation. Except for equality all symbols are given a valuation by the interpretation functions of logical structures. There are no symmetry-breaching operations in the sense of [ID96]. Canonical abstraction maps isomorphic structures to the same abstract structure. A characterization of canonical abstraction in terms of symmetry is given in Chapter 6. Since our method and [YR04] is related to [McM00], we compare the different approaches to verification of three-valued analysis (with decomposition) and [McM00]. Chapter 7 contains a report of our findings concerning this comparison and documents our practical experience with finite instantiation, data type reduction and the verification tool SMV. We model the Ticket Protocol with SMV.

---

<sup>1</sup> Two structures are isomorphic if they are identical up to a permutation of individuals.

## 1.4 Results

**A framework for quantifier instantiation.** Models are formulated in a many-sorted first-order logic with function symbols, equality and transitive closure. Temporal properties can be expressed in a branching temporal logic. A syntactic quantifier instantiation allows one to eliminate universal first-order quantification in front of temporal properties. Thus verification of quantified temporal properties can be reduced to the verification of temporal properties. The quantifier instantiation is a symbolic method which does not rely on a particular representation of abstract states. (see Chapters 2,3,4)

**An analysis for quantified temporal properties.** Using the framework for instantiation we construct an analysis for the special case of models formulated in predicate logic. Three-valued logical analysis is used to check quantified temporal properties. We describe implementations of state space exploration and give a correctness argument. We have implemented a prototype with TVLA and verified mutual exclusion of the Ticket Protocol using this implementation. (see Chapter 5)

**Finite instantiation and data type reduction.** We discuss how, our analysis in particular, and three-valued analysis in general relates to data type reduction and finite instantiation (see Chapter 7). Finite instantiation is based on symmetry arguments. The concept of anonymous individuals of [SRW02] is a form of symmetry. Canonical abstraction, the abstraction used in most work concerning three-valued logical analysis, collapses symmetries. (see Chapter 6)

## 1.5 Overview of the thesis

Chapter 2: Models.

Chapter 3: Properties.

Chapter 4: Quantifier Elimination.

Chapter 5: Analysis.

Chapter 6: Symmetry.

Chapter 7: Finite instantiation and data type reduction.

Chapter 8: Conclusion.

Appendix.

# Chapter 2

## Models

The models we consider can express programs with an unbounded number of processes, infinite data domains, and unbounded heap structures. Procedures or other complex control structures are not directly supported. Syntactically, we write models in first-order logic (our models closely resemble symbolic transition systems of [DPJ03, DW03]). The first-order logic  $\mathcal{FO}_\Sigma$  has equality symbols and an operator for transitive closure. The predicate logic of [SRW02] is the one-sorted sublogic of  $\mathcal{FO}_\Sigma$  without function symbols. The semantics of models are transition systems over a state space of first-order structures. The semantics is a constant-domain semantics which means that the states of transition systems induced by models share a common universe. We will discuss this choice later.

### 2.1 First-Order Logic

#### 2.1.1 Syntax

Signatures define a typed vocabulary with which we write first-order expressions. Symbols are typed by rank functions  $r$ . Types are tuples where the first component is tuple which gives the domain, the second component is the range, e.g. the successor function  $succ$  on the natural numbers  $Nat$  would have rank  $r(succ) = (Nat, Nat)$ . For a set  $X$ , we denote  $X^* = \bigcup_{0 \leq i \leq n} X^i$ .  $e$  stands for the empty tuple  $e \in X^0$ .

**Definition 2.1.1 (Signature).** *Let  $\mathcal{B} \cup \{Bool\}$  be a set of base types such that  $Bool \notin \mathcal{B}$  is the distinguished base type of truth values.*

- |                           |   |
|---------------------------|---|
| <i>variables.</i>         | <i>For every sort <math>T \in \mathcal{B}</math>, there is a countably infinite set of variables <math>V_T</math>. Each variable has rank <math>(e, T)</math>. The family of sets <math>V_T</math> is <math>\mathcal{V}</math>.</i> |
| <i>function symbols.</i>  | <i>Let <math>\mathcal{F}</math> be a set of function symbols. The type of each function symbol <math>f \in \mathcal{F}</math> is given by <math>r \in \mathcal{F} \rightarrow \mathcal{B}^* \times \mathcal{B}</math>.</i>          |
| <i>equality symbols.</i>  | <i>For each base type <math>T \in \mathcal{B}</math>, there is an equality symbol <math>=_T</math> with <math>r(=_T) = (TT, Bool)</math>.</i>   |
| <i>predicate symbols.</i> | <i>There is a set of predicate symbols <math>P</math> and the rank function <math>r \in P \rightarrow \mathcal{B}^* \times \{Bool\}</math>.</i>   |

$\mathcal{V}, P$ , and  $\mathcal{F}$  are disjoint. The tuple  $\Sigma = \langle \mathcal{B}, \mathcal{F}, P, \mathcal{V}, r \rangle$  is a **signature**.

We define the syntax of many-sorted first-order logic with transitive closure and equality. Ranks  $r$  are extended to the terms of the logic.

**Definition 2.1.2 (First-Order Logic).** *Let  $\Sigma = \langle \mathcal{B}, \mathcal{F}, P, \mathcal{V}, r \rangle$  be a signature. We define terms and expressions over  $\Sigma$  inductively.*

<i>terms <math>t \in \mathcal{T}_\Sigma</math>:</i>	
<i>truth values.</i>	<i>The literals <math>0, 1</math> are terms of rank <math>r(v) = (e, Bool)</math> without free variables.</i>
<i>variable.</i>	<i>A variable <math>v \in \mathcal{V}_T</math> is a term of rank <math>r(v) = (e, T)</math> with free variables <math>FV(v) = \{v\}</math>.</i>
<i>equality.</i>	<i>For terms <math>t_1, t_2</math> with <math>r(t_1) = r(t_2) = (e, T)</math> (<math>T \in \mathcal{B}</math>), <math>t_1 =_T t_2</math> is a term of rank <math>(e, Bool)</math> and has free variables <math>FV(t_1 = t_2) = FV(t_1) \cup FV(t_2)</math>.</i>
<i>compound term.</i>	<i>For <math>f \in \mathcal{F}</math> and <math>p \in P</math> with rank <math>r(f) = (T_1 \dots T_n, T)</math> and <math>r(p) = (T_1 \dots T_n, Bool)</math>, and for terms <math>t_1, \dots, t_n</math> of rank <math>r(t_1) = (e, T_1), \dots, r(t_n) = (e, T_n)</math>, the applications <math>t = f(t_1, \dots, t_n)</math> and <math>t' = p(t_1, \dots, t_n)</math> are terms of rank <math>r(t) = (e, T)</math> and <math>r(t') = (e, Bool)</math>, respectively, with free variables <math>FV(t) = FV(t') = \bigcup_{1 \leq i \leq n} FV(t_i)</math>.</i>
<i>expressions <math>e \in \mathcal{FO}_\Sigma</math>:</i>	
<i>term.</i>	<i>A term <math>t</math> of rank <math>r(t) = (e, Bool)</math> is an expression.</i>
<i>logical connectives.</i>	<i>For expressions <math>e_1, e_2</math>, conjunction <math>e_1 \wedge e_2</math> and negation <math>\neg e_1</math> are expressions. The free variables of a conjunction are <math>FV(e_1 \wedge e_2) = FV(e_1) \cup FV(e_2)</math> and of negation <math>FV(\neg e_1) = FV(e_1)</math>.</i>
<i>quantification.</i>	<i>For a variable <math>x \in \mathcal{V}_T</math> and an expression <math>e</math> with <math>x \in FV(e)</math> the universal quantification <math>\forall x : T. e</math> is an expression with free variables <math>FV(\forall x : T. e) = FV(e) \setminus \{x\}</math>.</i>
<i>transitive closure.</i>	<i>For variables <math>v_1, v_2, v_3, v_4 \in \mathcal{V}_T</math> and an expression <math>e</math> with <math>v_3, v_4 \notin FV(e)</math>, <math>v_1, v_2 \in FV(e)</math> the transitive closure <math>(TC v_1, v_2 : T.e)(v_3, v_4)</math> is an expression with free variables <math>FV((TC v_1, v_2 : T.e)(v_3, v_4)) = (FV(e) \setminus \{v_1, v_2\}) \cup \{v_3, v_4\}</math>.</i>

An expression  $e$  is closed if  $FV(e) = \emptyset$ .

Constants can be expressed as function symbols of rank  $r(e, T)$ .

**Notation.** Let  $e_1, e_2$  be two expressions. We define the abbreviations:

$$\begin{aligned}
e_1 \vee e_2 &\equiv \neg(\neg e_1 \wedge \neg e_2) \\
e_1 \Rightarrow e_2 &\equiv \neg e_1 \vee e_2 \\
e_1 \Leftrightarrow e_2 &\equiv (e_1 \Rightarrow e_2) \wedge (e_2 \Rightarrow e_1) \\
\exists x : T.e &\equiv \neg \forall x : T. \neg e
\end{aligned}$$

We omit the subscript  $T$  when writing down the equation symbol  $=_T$  and write  $=$  instead. This is not problematic because the semantic domains of the base types will be disjoint.

## 2.1.2 Semantics

First-order expressions are interpreted over logical structures. Logical structures provide a semantic domain  $\Delta(T)$  for each base type  $T \in \mathcal{B}$ . The universe  $U$  of a logical structure is the disjoint union of all semantic domains (the dot over a union operator  $\cup$  signifies disjoint union). The elements of the universe are called individuals. The base type  $Bool$  has always the semantic domain  $\mathbb{B} = \{0, 1\}$ . Function and predicate symbols are interpreted as functions over the universe. Let  $\Sigma = \langle \mathcal{B}, \mathcal{F}, P, \mathcal{V}, r \rangle$  be a signature.

**Definition 2.1.3 (Logical Structure).** *A logical structure is a tuple  $\langle U, \Delta, \iota \rangle$  with*

- universe.*  $U$  is a universe of values. The elements of the universe  $U$  are called individuals.
- semantic domains.*  $\Delta$  is a function which maps each base type  $T$  to a set of values  $\Delta(T) \subseteq U$ . The universe  $U = \dot{\bigcup}_{T \in \mathcal{B}} \Delta(T)$  is the disjoint union of all semantic domains.
- interpretation.* The interpretation  $\iota$  maps each function symbol  $f \in \mathcal{F}$  of rank  $r(f) = (T_1 \dots T_n, T)$  to a function

$$\iota(f) \in \Delta(T_1) \times \dots \times \Delta(T_n) \rightarrow \Delta(T) .$$

The interpretation  $\iota$  maps each predicate symbol  $p \in P$  of rank  $r(p) = (T_1 \dots T_n, Bool)$  to a function

$$\iota(p) \in \Delta(T_1) \times \dots \times \Delta(T_n) \rightarrow \mathbb{B} .$$

We denote the set of logical structures over  $\Sigma$  as  $Struct[\Sigma]$ . Sometimes we omit the universe of a logical structure and write  $\langle \Delta, \iota \rangle$ , since the universe is uniquely determined by  $U = \bigcup_{T \in \mathcal{B} \setminus \{Bool\}} \Delta(T)$ . Using this notation, we define:

$$Struct[\Sigma, \Delta] = \{s \mid s = \langle \Delta, \iota \rangle \in Struct[\Sigma]\} .$$

which is the the set of logical structures over  $\Sigma$  with fixed semantic domains  $\Delta$ .

**Definition 2.1.4 (Semantics of First-Order Logic).** *Let  $s = \langle U, \Delta, \iota \rangle \in Struct[\Sigma, \Delta]$  be a logical structure. An environment is a function  $Z \in \mathcal{V} \rightarrow U$ . We write  $Env_U$  for the set of environments. A complete assignment for an expression  $e$  is a function  $Z \in Env_U$  such that  $FV(e) \subseteq dom(Z)$ . The valuation  $[t]$   $s$   $Z$  of a term  $t$  in state  $s$  and with the complete assignment  $Z \in Env_U$  is an element of  $U \cup \mathbb{B}$ . We define the valuation of terms inductively:*

$$\begin{aligned} [x] \quad s \ Z &= (Z \ x) \\ [t_1 = t_n] \quad s \ Z &= ([t_1] \ s \ Z) = ([t_2] \ s \ Z) \\ [f(t_1, \dots, t_n)] \quad s \ Z &= \iota(f)([t_1] \ s \ Z, \dots, [t_n] \ s \ Z) \\ [p(t_1, \dots, t_n)] \quad s \ Z &= \iota(p)([t_1] \ s \ Z, \dots, [t_n] \ s \ Z) \end{aligned}$$

where  $f \in \mathcal{F}$ ,  $p \in P$ .

The valuation  $[e]$   $s$   $Z$  of an expression  $e$  in a state  $s \in Struct[\Sigma, \Delta]$  and with the complete assignment  $Z$  is an element of  $\mathbb{B}$ .

We define the valuation of expressions inductively:

$$\begin{aligned} [e_1 \wedge e_2] \quad s \ Z &= \min\{[e_1] \ s \ Z, [e_2] \ s \ Z\} \\ [\neg e] \quad s \ Z &= 1 - ([e] \ s \ Z) \\ [\forall x : T. e] \quad s \ Z &= \min\{[e] \ s \ (Z \cup \{x \mapsto v\}) \mid v \in \Delta(T)\} \\ [c] \quad s \ Z &= c \end{aligned}$$

$$[(TC \ v_1, v_2 : T, e)(v_3, v_4)] \ s \ Z = \max_{\substack{n \geq 1, w_1, \dots, w_n \in \Delta(T) \\ Z(v_3)=w_1, Z(v_4)=w_n}} \min_{1 \leq i \leq n} ([e] \ s \ Z_{w_i, w_{i+1}})$$

where  $Z_{w_i, w_{i+1}} = Z \cup \{v_1 \mapsto w_i, v_2 \mapsto w_{i+1}\}$ .

Let  $e \in \mathcal{FO}_\Sigma$  be a closed expression and  $S' \subseteq \text{Struct}[\Sigma]$ . The set denotation of  $e$  with respect to  $S'$  is the set  $\llbracket e \rrbracket_{S'} = \{s \in S' \mid ([e] \ s \ \emptyset) = 1\}$ .

When we mention assignments we will implicitly mean complete assignments from now on.

## 2.2 Syntax and Semantics of Models

In order to express transition systems, we also need to denote binary relations upon states. We denote relations as expressions over primed and unprimed symbols; hence, technically we are dealing with expressions over the signature  $\Sigma \cup \Sigma'$ , which is defined as:

$$\Sigma \cup \Sigma' := \langle \mathcal{B}, \mathcal{F} \cup \{f' \mid f \in \mathcal{F}\}, P \cup \{p' \mid p \in P\}, \mathcal{V}, r \cup \{f' : r(f)\} \cup \{p' : r(p)\} \rangle .$$

Given a pair of states  $s, s'$  and expression  $\rho \in \mathcal{FO}_{\Sigma \cup \Sigma'}$  the idea is that  $s$  evaluates unprimed and  $s'$  primed symbols. The set denotation of  $\rho$  is then the set of pairs that evaluate  $\rho$  to 1. This idea is realized in the ensuing definition.

**Definition 2.2.1 (Relations).** Let  $\rho \in \mathcal{FO}_{\Sigma \cup \Sigma'}$  be a closed first-order expression over primed and unprimed symbols. Let  $S'$  be a subset of  $\text{Struct}[\Sigma, \Delta]$ . We define the denotation  $\llbracket \rho \rrbracket_{S' \times S'} := \text{cross}_{\Sigma, \Delta}(\llbracket \rho \rrbracket_{\text{cross}_{\Sigma, \Delta}^{-1}(S' \times S')})$  where  $\text{cross}_{\Sigma, \Delta}$  is the one-to-one correspondence

$$\begin{aligned} \text{cross}_{\Sigma, \Delta} &\in \text{Struct}[\Sigma \cup \Sigma', \Delta] \rightarrow \text{Struct}[\Sigma, \Delta] \times \text{Struct}[\Sigma, \Delta], \\ \langle \Delta, \iota \rangle &\mapsto (\langle \Delta, \{c \mapsto \iota(c) \mid c \in \mathcal{F} \cup P\} \rangle, \langle \Delta, \{c \mapsto \iota(c') \mid c \in \mathcal{F} \cup P\} \rangle) . \end{aligned}$$

Let  $S'$  be a subset of  $\text{Struct}[\Sigma]$ . We set  $\llbracket \rho \rrbracket_{S' \times S'} = \bigcup_{\Delta} \llbracket \rho \rrbracket_{\text{Struct}[\Sigma, \Delta] \cap S'}$ .

**Definition 2.2.2 (Transition System).** Let  $S$  be a set,  $I \subseteq S$  and  $R \subseteq S \times S$ . Furthermore,  $R$  fulfills that  $\forall s \in S \ \exists t \in S : R(s, t)$  ("Every state has a successor"). Then the tuple  $K = \langle S, I, R \rangle$  is called a transition system. The elements of  $S$  are called states. For every state  $s$ , a state  $s'$  with  $R(s, s')$  is called a successor. The elements of  $I$  are called the initial states of  $K$ , and  $R$  is called the transition relation of  $K$ . We denote the set of transition systems with state space  $S$  as  $\mathcal{K}_S$ .

Models, as we will define them, are not entirely syntactic entities. A model consists of two expressions  $\theta, \rho$ , and a state space over which the expressions, which denote transitions and initial states, respectively, are to be evaluated. Alternatively, we could have used a state space  $Struct[\Sigma]$  or a state space with fixed semantic domains  $Struct[\Sigma, \Delta]$ . There is also the possibility to use constraints to describe the state space. Our motivation is that we want to have a framework which is flexible enough to account for all the choices just mentioned.

**Definition 2.2.3 (Model).** *Let  $\Sigma$  be a signature and  $\Delta$  semantic domains.*

*model.* *A tuple  $M = \langle S, \theta, \rho \rangle \in \mathcal{P}(Struct[\Sigma]) \times \mathcal{FO}_\Sigma \times \mathcal{FO}_{\Sigma \cup \Sigma'}$  is called (many-sorted) model if  $\rho, \theta$  are closed.  $S$  is called the state space of  $M$ . The initial states and the transitions are given as first-order expressions  $\theta$  and  $\rho$ . We write  $\mathcal{M}_\Sigma$  for the set of models over signature  $\Sigma$ .*

*constant-domain model.* *A model  $M$  with state space  $S \subseteq Struct[\Sigma, \Delta]$  (for some semantic domains  $\Delta$ ) is called constant-domain model.*

*semantics.* *The semantics of a model  $M$  is the transition system  $\llbracket M \rrbracket = \langle S, \llbracket \theta \rrbracket_S, \llbracket \rho \rrbracket_S \rangle \in \mathcal{K}_S$ .*

The notion of a constant-domain model is that all structures in the state space of the model share the same semantic domain  $\Delta$ . The term varying-domain means that structures may have different or even disjoint semantic domains.

## 2.3 Predicate Logic

An interesting special case of first-order logic is one-sorted (first-order) predicate logic. For this sublogic we introduce simplified notation (which matches the notation of [SRW02]).

**Definition 2.3.1 (Predicate Logic).** *We introduce a fixed base type  $\mathcal{U}$ . A predicate signature is a signature  $\Sigma = \langle \{\mathcal{U}, Bool\}, \emptyset, P, r \rangle$  (cf. 2.1.1). We write  $r(p) = k$  for  $r(p) = (\mathcal{U}^k, Bool)$ . Syntactically, we can simplify the signature to  $\Sigma = \langle P, \mathcal{V}, r \rangle$ . We use the meta-variable  $\mathcal{P}$  for predicate signatures.*

*Predicate logic expressions  $\mathcal{FO}_\mathcal{P}$  can syntactically be simplified by removing type annotations. This leads to the syntax:*

$$e \in \mathcal{FO}_\mathcal{P} ::= 0 \mid 1 \mid v_1 = v_2 \mid p(v_1, \dots, v_k) \mid e \wedge e \mid \neg e \mid \forall x. e \mid (TC \ v_1, v_2 : e)(v_3, v_4)$$

where  $p \in P$ ,  $u, v, v_1, \dots, v_k \in \mathcal{V}$  and  $r(p) = k$ .

*A structure of predicate logic is a logical structure over a predicate signature. We syntactically simplify the notation for logical structures of predicate logic  $\langle \Delta, \iota \rangle$  to  $\langle U, \iota \rangle$  where  $U$  is the meta-variable for the universe  $U = \Delta(\mathcal{U})$ . For a fixed universe  $U$  we define  $Struct[\mathcal{P}, U] = \{ \langle U, \iota \rangle \mid \langle U, \iota \rangle \in Struct[\mathcal{P}] \}$ .*

The definition of valuation and denotation of general first-order expressions (cf. Def. 2.1.4, the valuation agrees with [SRW02]), the definition of models (cf. Def. 2.2.3), and of properties (cf. Definitions 3.1.1, 3.2.2) carry over, since predicate logic is a sublogic of first-order logic.



## 2.4 Example

**Running Examples.** We consider two programs (both depicted in Figure 1.1). One program, called *ADD*, adds two numbers. The other program is the Ticket Protocol. The *Ticket Protocol* is a small program which has an infinite data domain, and an unbounded number of processes. The protocol is a solution to the problem of mutual exclusion (more about that in Chapter 3) between concurrent processes. Mutual exclusion is insured with counter variables. The counter values are called tickets. The analogy are the numbered tickets sometimes issued in bureaus of authorities. Each process has a ticket, which is modeled by the local process variable  $a$ . There is always a winner ticket, the value of global variable  $s$ ; global variable  $t$  holds a fresh ticket that is by one larger than the greatest ticket issued so far.

We produce a model  $M_{TICKET}$  corresponding to the Ticket Protocol, which is written down in a tentative C-like syntax in Figure 1.1. We start by finding an appropriate signature  $\Sigma$ . There are the following sorts: truth values  $Bool$ , process indices  $Proc$ , process locations  $Loc$ , counter values  $Nat$ , i.e., tickets. Thus the set of base types is  $\{Bool\} \cup \{Proc, Nat, Loc\}$ . We model the variables of the program as symbols: there are global variables  $s$ , for the winner ticket, and  $t$  for a fresh ticket. Both have rank  $(e, Nat)$ . There are local process variables  $a$  for each process. We model  $a$  as a function symbol  $a$  of rank  $(Proc, Nat)$ . The location of a process is modeled as a function symbol  $at : (Proc, Loc)$ . Implicitly, there is a variable  $act : (e, Proc)$ , the process that is currently active (we assume that processes are run in concurrent and interleaved fashion). Furthermore, we need equality predicates and a successor function on the tickets. As a set of variable we can choose  $\mathbb{N}$ , the set of natural numbers. We obtain the following signature:

$$\begin{array}{ll} \Sigma = \langle & \{Nat, Proc, Loc\}, & \text{(base types)} \\ & \{act, s, t, a, succ\}, & \text{(function symbols)} \\ & \emptyset, & \text{(predicate symbols)} \\ & \mathbb{N}, & \text{(variables)} \\ & \{act : (e, Proc), s : (e, Nat), t : (e, Nat), a : (Proc, Nat), & \text{(typing)} \\ & at : (Proc, Loc), succ : (Nat, Nat)\} & \text{(typing cont.)} \end{array}$$

Next we write down the informal program as a model. The semantic domains are  $\Delta(Nat) = \mathbb{N}$ ,  $\Delta(Loc) = \{think, wait, crit\}$ .  $\Delta(Proc) \subseteq \mathbb{N}$  is some fixed subset of the natural numbers. The state space

$$S = \{s = \langle \Delta, \iota \rangle \mid s \in Struct[\Sigma, \Delta], \iota(succ)(n) = n + 1\}$$

is simply the set of all structures such that the equality symbols have the "correct" valuation. The initial states are given by the first-order expression:

$$\theta \equiv s = 0 \wedge t = 0 \wedge \forall p : Proc. (a(p) = 0 \wedge at(p) = think)$$

and the transition relation by:

$$\begin{aligned}
\rho &\equiv a(\text{act}) = t \wedge at(\text{act}) = \text{think} \wedge \\
&\quad at'(\text{act}) = \text{wait} \wedge t' = \text{succ}(t) \wedge s' = s \wedge \\
&\quad \forall p : \text{Proc.} \neg(\text{act} = p) \Rightarrow a'(p) = a(p) \wedge at'(p) = at(p) \\
\vee &\quad a(\text{act}) = s \wedge at(\text{act}) = \text{wait} \wedge \\
&\quad t' = t \wedge s' = s \wedge \\
&\quad \forall p : \text{Proc.} \neg(\text{act} = p) \Rightarrow a'(p) = a(p) \wedge at'(p) = at(p) \\
\vee &\quad at(\text{act}) = \text{crit} \wedge \\
&\quad at'(\text{act}) = \text{think} \wedge s' = \text{succ}(s) \wedge t' = t \wedge \\
&\quad \forall p : \text{Proc.} \neg(\text{act} = p) \Rightarrow a'(p) = a(p) \wedge at'(p) = at(p) \quad .
\end{aligned}$$

## 2.5 Discussion

Note that the state space of a model is not given syntactically. Thus we can account for several possibilities. In [SRW02], models are considered where the state space is implicitly given by constraints over  $\text{Struct}[\mathcal{P}]$  (called: "Compatibility constraints"). This does in general not produce a constant-domain model. We discuss this aspect within this section.

Leaving the state space open allows one to smuggle symbols of a fixed valuation into the model, e.g. the successor function on natural numbers in the Example section. The only symbol of fixed valuation in [SRW02] is the equality symbol.

**Symbolic Transition Systems.** The models we consider are very similar to symbolic transition systems in [DPJ03, DW03]. However, we do not refer to  $\theta, \rho$  as predicates, since this would clash with the predicates of our logic. Furthermore, a snapshot corresponds to a logical structure.

**Aspects concerning the framework [SRW02].** The framework uses predicate logic (as described in Definition 2.3.1). The state space  $S \subseteq \text{Struct}[\mathcal{P}]$  of the concrete model is implicitly given by constraints. We give a formal description of constraints in Definition 5.2.1. Notably,  $S$  possibly consists of structures with different universes. We need constant-domain models, so this is a point which deserves some attention.

Models of [SRW02] are given by a predicate signature  $\mathcal{P}$ , a set of compatibility constraints  $\mathfrak{R}$  (cf. 5.2.1), a transition relation  $\rho$ , and initial states  $\theta$ . Unlike our work, the framework does not consider constant-domain models. Varying-domain models are considered because allocation and deallocation of heap-manipulating programs is modeled by universe-changing transitions, i.e., transitions between structures with a different universe (Definition 2.2.3 does not account for universe-changing models). Even if we consider models which do not change the universe, there remains the discrepancy between our constant-domain models and the varying-domain models of [SRW02]. We need to figure out whether that causes a problem. Formally, a model (a varying-domain model which is not universe-changing) in the sense of Definition 2.2.3 can be obtained by taking the structures  $S_{\mathfrak{R}} = \{s \in \text{Struct}[\mathcal{P}] \mid s \models \mathfrak{R}\}$  which satisfy the constraints. Then  $M = \langle S_{\mathfrak{R}}, \theta, \rho \rangle$  is a model with respect to 2.2.3. We need constant-domain models. Nothing detains us from considering constant-domain models. Let us choose a universe  $U$ . We set

$$S_{\mathfrak{R},U} = \{s \in \text{Struct}[\mathcal{P}, U] \mid s \models \mathfrak{R}\} = S_{\mathfrak{R}} \cap \text{Struct}[\mathcal{P}, U]$$

and obtain the constant-domain model  $M_U = \langle S_{\mathfrak{R},U}, \theta, \rho \rangle$ . It turns out that the discrepancy between  $M$  and  $M_U$  is not problematic for us, as neither of them are actually computed. We will use a simulation preorder (cf. 5.1.12) to insure property preservation. The analysis algorithm of [SRW02] can be used to compute a transition system  $K^\sharp$  which simulates  $\llbracket M \rrbracket$ , denoted by  $\llbracket M \rrbracket \preceq K^\sharp$ . Clearly,  $M$  simulates  $M_U$ ,  $\llbracket M_U \rrbracket \preceq \llbracket M \rrbracket$ . As simulation is a preorder, we have  $\llbracket M_U \rrbracket \preceq K^\sharp$ , which is sufficient for our purpose. We do not even have to consider  $M$  in the correctness proof of our analysis in Chapter 5. We can directly show that the constant-domain model  $M_U$  is simulated by  $K^\sharp$ .

There is one point in the thesis where the constraint mechanism plays a role. The quantifier elimination technique for predicate logic in Chapter 4 modifies the state space of a model. We describe how the appropriate state space can be generated using the constraint mechanism, *and* we give the appropriate constraints in Section 4.3 (for readers who are familiar with [SRW02]).

The form of the transition relation  $\rho$ , which is termed transformer in [SRW02], is also of interest. The transformer consists of actions, which model the semantic effect of statements of a programming language. Actions can be described as expressions of predicate logic. An action  $ac$  assigns to each predicate symbol its valuation in the next state, i.e., for each predicate symbol  $p$ , there is an update formula  $u_p$  which is evaluated in the current state (it contains only unprimed predicate symbols). The update formula  $u_p$  has free variables  $v_1, \dots, v_{r(p)}$  to match the number of arguments  $r(p)$  of predicate  $p$ . Furthermore, an action is guarded by a precondition  $pre$ , an expression which is evaluated in the current state (it contains only unprimed function symbols). The preconditions can be used to model a part of the control structure of a program, such as conditionals. An action  $ac$  corresponds to the following expression  $e_{ac}$  over primed and unprimed symbols:

$$e_{ac} \equiv pre \wedge \bigwedge_{p \in P} \forall v_1, \dots, v_{r(p)}. p'(v_1, \dots, v_{r(p)}) = u_p(v_1, \dots, v_{r(p)})$$

where  $\bigwedge_{p \in P}$  is a syntactic to express the conjunction over all predicate symbols. A program may consist of several actions  $ac \in Ac$ . In the shape analysis framework [SRW02] control and hence the invocation of actions depends not only on preconditions (although this would be doable) but on a control flow graph. [Yah01] adapts the methods of [SRW02] to the verification of safety properties of concurrent Java programs. Therefore the concurrency model of Java has to be modeled. The nondeterministic choice of the next thread to be executed is modeled by a system variable. We gloss over some technicalities. Let us call this variable *active*. An action is executed by a thread, namely by the thread referenced by *active*. An action has to "know" the active process and may contain *active* as a free variable. If we encode the control flow within threads using preconditions, we obtain the transitions of such a model as the disjunction over all actions:

$$\rho \equiv \bigvee_{ac \in Ac} e_{ac} .$$

The models which are considered in [YRSW03] are the ones discussed [Yah01].

**Varying-domain vs constant-domain semantics.** We give models a *constant-domain semantics*, i.e., transition systems over logical structures that share the same universe.

Creation and destruction can be modeled with one-argument predicates that mark individuals which are live (as described in [DW03]). We do not focus on heap-manipulating programs. We assume that allocation and deallocation are encoded by a unary predicate *live*. Exactly individuals which are allocated are *live*.

[YRSW03] uses a varying-domain semantics. It is assumed that all structures have disjoint universes. An individual exists in exactly one logical structure. As the specification language used there is a linear-time logic (called ETL), sets of traces rather than transition systems are used as a semantics of models. The traces are sequences of logical structures. In order to relate individuals of different states in a trace, and to handle destruction and creation, annotations are used:

Each structure  $\pi_i$  in a trace  $\pi$  is annotated with a subset of its universe  $D_i$  to be deallocated and elements  $A_i$  that have been allocated at  $i$ . Furthermore, a function maps the elements to be preserved to elements of the universe of the successor  $\pi_{i+1}$ . This varying-domain semantics is more complex than our constant-domain semantics. Later on, we will see that the analysis based on the varying-domain semantics requires additional infrastructure.

Why we have chosen a constant-domain semantics will become clear in the following two chapters.

# Chapter 3

## Properties

We introduce the syntax and semantics of a first-order branching-time logic in which we specify properties of models. The branching-time logic can quantify over paths starting in a state.

CTL\* is a branching-time logic of which LTL and CTL are sublogics. We augment CTL\* with an operator  $\forall$  which expresses first-order quantification across time. This allows us to refer to individuals in different states, e.g.,  $\forall i : Process. \mathbf{AF} at(i) = critical$ . This quantification operator has the same semantics, if applied to a first-order expression, as the universal quantification operator of first-order logic. Therefore we do not distinguish between the two. We say that a formula contains **quantification across time** if  $\forall$  is applied to a formula which is not a first-order expression. Here is a short description of the meaning of the most interesting temporal operators (we adopted the treatment from [CGP00], the expressiveness of the logics LTL and CTL is discussed in [CD89]):

the path quantifiers:

- **E** ("exists") there exists a path starting in this state that fulfills a path formula.
- **A** ("for all") every path starting in this state fulfills a path formula.

temporal operators of path formulas:

- **X** ("next time") require that a property holds in the second state of a path.
- **F** ("eventually") operator expresses that a property will hold at some state on a path.
- **G** ("globally") specifies that a property holds in every state on a path.
- The **U** ("until") operator can be used to combine properties. It holds if there is a state on a path where the second property holds, and at every preceding state on the path, the first property holds.
- The **R** ("release") is the logical dual of **U**. It requires that the second property holds along the path up to and including the first state where the first property holds. However, the first property is not required to hold eventually.

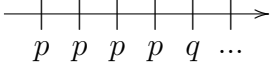
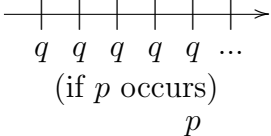
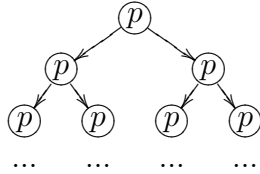
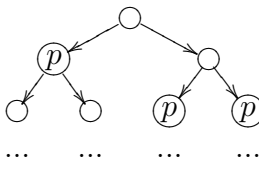
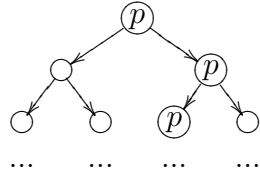
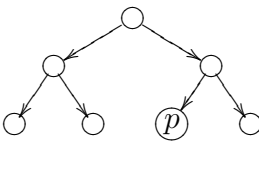
path formulas: <b>U</b> ("until") and <b>R</b> ("release")	state formulas: path quantifiers <b>E</b> ("exists") and <b>A</b> ("all")
<p style="text-align: center;"><math>p\mathbf{U}q</math></p> <p style="text-align: center;"><math>p</math> holds until <math>q</math> holds (and <math>q</math> must hold eventually)</p>  <p style="text-align: center;"><math>p\mathbf{R}q</math></p> <p style="text-align: center;"><math>q</math> holds up to and including the first state where <math>p</math> holds (<math>p</math> may never hold)</p> 	<p style="text-align: center;"><math>\mathbf{AG} p</math></p>  <p style="text-align: center;"><math>\mathbf{AF} p</math></p>  <p style="text-align: center;"><math>\mathbf{EG} p</math></p>  <p style="text-align: center;"><math>\mathbf{EF} p</math></p> 

Figure 3.1: CTL\* consists of path formulas  $\Phi$  and state formulas  $\phi$ . The path formula  $\mathbf{F}\Phi$  can be obtained from the  $\mathbf{U}$  operator as  $1\mathbf{U}\Phi$ . The path formula  $\mathbf{G}\Phi$  can be obtained as  $(\neg\Phi)\mathbf{R}\Phi$ .

Figure 3.1 illustrates the temporal operators  $\mathbf{U}$  and  $\mathbf{R}$  on the left and the path quantifiers  $\mathbf{E}$  and  $\mathbf{A}$  on the right.

The semantics of FCTL\* strongly resembles that of CTL\* as, e.g., given in [CGP00]. The only difference is that because of first-order quantification we need environments  $Z$ .

### 3.1 Syntax

The syntax of the specification language FCTL\* ("F" stands for "first-order") is given by the following definition:

**Definition 3.1.1 (FCTL\*).** *Let  $\Sigma$  be a signature. We define the logic FCTL\*. It is composed of state formulas  $\phi$ , and path formulas  $\Phi$ . It is defined recursively by:*

$$\begin{aligned} \phi & ::= e \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathbf{A}\Phi \mid \mathbf{E}\Phi \mid \forall x : T. \phi \quad (\text{state}) \\ \Phi & ::= \phi \mid \neg\Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \mathbf{X}\Phi \mid \Phi\mathbf{U}\Phi \mid \Phi\mathbf{R}\Phi \quad (\text{path}) \end{aligned}$$

where  $e \in \mathcal{FO}_\Sigma$ ,  $x \in \mathcal{V}_T$ . We denote the set of state formulas  $\phi$  over signature  $\Sigma$  as  $\text{FCTL}_\Sigma^*$ .

An FCTL\* formula  $\phi$  **quantifies across time** if it has a subexpression  $\forall x : T. \phi'$  such that  $\phi'$  is not a first-order expression, i.e.,  $\phi' \notin \mathcal{FO}_\Sigma$ .

The sublogic consisting of  $FCTL^*$  formulas  $\phi$  which do not quantify across time is called  $CTL^*$ . We denote the set of those formulas by  $CTL^*_\Sigma$ .  $QCTL^*$  formulas are  $CTL^*$  with an arbitrary number of universal quantifiers in front.

The "eventually" and the "globally" operator can be expressed with the "until" operator and the "release" operator :

$$\begin{aligned} \mathbf{F}\Phi &\equiv \mathbf{1}\mathbf{U}\Phi \\ \mathbf{G}\Phi &\equiv (\neg\Phi)\mathbf{R}\Phi \end{aligned}$$

As remarked in [Ber02]  $\neg\Phi_1\mathbf{R}\Phi_2$  can be interpreted as strong induction on time where  $\Phi_1$  is the induction hypothesis and  $\Phi_2$  the conclusion of the induction step.  $\mathbf{G}\Phi \equiv (\neg\Phi)\mathbf{R}\Phi$  is often used in circular reasoning.

We are particularly interested in a sublogic of  $FCTL^*$  we call  $QACTL^*$ . The quantifier elimination and the analysis we will present later allow us to verify  $QACTL^*$  properties. Relative to  $FCTL^*$  the sublogic  $QACTL^*$  restricts the use of first-order quantification and forbids existential path quantification. Syntactically excluding the  $\mathbf{E}$  path quantifier is not sufficient because negation produces implicit existential path quantification; the  $\mathbf{E}$  path quantifier is dual to  $\mathbf{A}$ . So we push down negation to the level of expressions. Without loss of generality we can assume that temporal formulas are in positive normal form, i.e. negation is only applied to first-order expressions. Because of the restricted use of negation we keep duals of operators, e.g.,  $f\mathbf{R}g \equiv \neg(\neg f\mathbf{U}\neg g)$ , lest we lose expressiveness. We obtain the logic  $QACTL^*$  by allowing an arbitrary number of universal quantifiers in front of an  $ACTL^*$  formula.

**Definition 3.1.2 (QACTL\*).** We obtain a logic  $ACTL^*$

$$\begin{aligned} \varphi &::= e \mid \neg e \mid \varphi \wedge \phi \mid \varphi \vee \varphi \mid \mathbf{A}\Phi && \text{(state)} \\ \Phi &::= \varphi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \mathbf{X}\Phi \mid \Phi\mathbf{U}\Phi \mid \Phi\mathbf{R}\Phi && \text{(path)} \end{aligned}$$

which consists of all  $CTL^*$  formulas in positive normal form that only contain the  $\mathbf{A}$  path quantifier. The first-order logic  $QACTL^*$  is defined recursively as:

$$\phi ::= \varphi \mid \forall x : T. \phi \quad (QACTL^*)$$

where  $\varphi$  is an  $ACTL^*$  formula and  $x \in \mathcal{V}_T$ .

## 3.2 Semantics

The following semantics of  $FCTL^*$  is a standard  $CTL^*$  semantics plus first-order quantification; universal quantification occurs in state formulas. A model fulfills a formula if all of its initial states fulfill the formula.

**Definition 3.2.1 (Paths).** Let  $K = \langle S, I, R \rangle \in \mathcal{K}_S$  be a transition system. An infinite path in  $K$  is a sequence  $\pi \in \mathbb{N} \rightarrow S$  such that  $\forall i \in \mathbb{N} : R(\pi_i, \pi_{i+1})$ . We denote the set of infinite paths in  $K$  as  $\Pi_K$ . For all paths  $\pi \in \Pi_K$  we define the notation  $\pi_k := \pi(k)$ , which means that we pick the  $k$ -th state in a path, and  $\pi^k := \pi|_{\{i \in \mathbb{N} \mid i \geq k\}}$ , which is the postfix of a path  $\pi$  starting at and including the  $k$ -th state.

**Definition 3.2.2 (Semantics of FCTL\*).** Let  $K = \langle S, I, R \rangle \in \mathcal{K}_S$  be a transition system such that  $S \subseteq \text{Struct}[\Sigma, \Delta]$ .

state formulas:

$$\begin{aligned}
K, s, Z \models e & :\Leftrightarrow ([e] s Z) = 1 \\
K, s, Z \models \neg\phi & :\Leftrightarrow \neg(K, s, Z \models \phi) \\
K, s, Z \models \phi_1 \wedge \phi_2 & :\Leftrightarrow K, s, Z \models \phi_1 \text{ and } K, s, Z \models \phi_2 \\
K, s, Z \models \phi_1 \vee \phi_2 & :\Leftrightarrow K, s, Z \models \phi_1 \text{ or } K, s, Z \models \phi_2 \\
K, s, Z \models \mathbf{A}\Phi & :\Leftrightarrow \forall \pi \in \Pi_K : \pi_0 = s \Rightarrow K, \pi, Z \models \Phi \\
K, s, Z \models \mathbf{E}\Phi & :\Leftrightarrow \exists \pi \in \Pi_K : \pi_0 = s \wedge K, \pi, Z \models \Phi \\
K, s, Z \models \forall x : T. \phi & :\Leftrightarrow \forall u \in \Delta(T) : K, s, (Z \cup \{x \rightarrow u\}) \models \phi
\end{aligned}$$

path formulas:

$$\begin{aligned}
K, \pi, Z \models \phi & :\Leftrightarrow K, \pi_0, Z \models \phi \\
K, \pi, Z \models \neg\Phi & :\Leftrightarrow \neg(K, \pi, Z \models \Phi) \\
K, \pi, Z \models \Phi_1 \wedge \Phi_2 & :\Leftrightarrow K, \pi, Z \models \Phi_1 \text{ and } K, \pi, Z \models \Phi_2 \\
K, \pi, Z \models \Phi_1 \vee \Phi_2 & :\Leftrightarrow K, \pi, Z \models \Phi_1 \text{ or } K, \pi, Z \models \Phi_2 \\
K, \pi, Z \models \mathbf{X}\Phi & :\Leftrightarrow K, \pi^1, Z \models \Phi \\
K, \pi, Z \models \Phi_1 \mathbf{U}\Phi_2 & :\Leftrightarrow \exists k \in \mathbb{N} : K, \pi^k, Z \models \Phi_2 \wedge \forall 0 \leq j \leq k : K, \pi^j, Z \models \Phi_1 \\
K, \pi, Z \models \Phi_1 \mathbf{R}\Phi_2 & :\Leftrightarrow \forall j \in \mathbb{N} : (\forall i < j : K, \pi^i, Z \not\models \Phi_1) \Rightarrow K, \pi^j, Z \models \Phi_2 .
\end{aligned}$$

The denotation of a closed state formula  $\phi$  relative to  $K$  is  $\llbracket \phi \rrbracket_K = \{s \in S \mid K, s, \emptyset \models \phi\}$ . We say that:

$$\boxed{K \text{ fulfills } \phi, \text{ denoted as } K \models \phi, \text{ iff } I \subseteq \llbracket \phi \rrbracket_K.}$$

Let  $M \in \mathcal{M}_\Sigma$  be a constant-domain model. We say that  $M$  fulfills  $\phi$ , denoted as  $M \models \phi$ , if and only if  $\llbracket M \rrbracket$  fulfills  $\phi$ .

When the transition system  $K$  is clear from context, we shall write  $s, Z \models \phi$  and  $\pi, Z \models \Phi$  instead of  $s, Z \models \phi$  and  $K, \pi, Z \models \Phi$ , respectively.

**Semantics of sublogics.** QCTL\* and QACTL\*, being sublogics of FCTL\*, have a semantics through 3.2.2. CTL\*, though being a sublogic, can also be interpreted over varying-domain transition systems (with a state space where logical structures have inhomogeneous universes). Therefore, we drop the limitation  $S \subseteq \text{Struct}[\Sigma, \Delta]$  in definition and just require  $S \subseteq \text{Struct}[\Sigma]$  in 3.2.2. Furthermore, we ignore the equation for quantification and drop the environments, since environments only make sense in presence of quantification. Thus we obtain a CTL\* semantics.

**Example 3.2.1.** We discuss the QACTL\* properties shown in Figure 3.2. The figure contains the property  $\phi_{\text{ADD}}$  which refers to the addition program  $M_{\text{ADD}}$  and two properties of the Ticket Protocol  $M_{\text{TICKET}}$ . Being a solution to the mutual exclusion problem means that the program is safe, in that no two processes are simultaneously in their critical section, as expressed by  $\phi_{\text{MUTEX}}$  in Figure 3.2.

Liveness means that every process, once it is in location "think", will eventually reach its critical section, as expressed by  $\phi_{\text{LIVE}}$  in Figure 3.2.



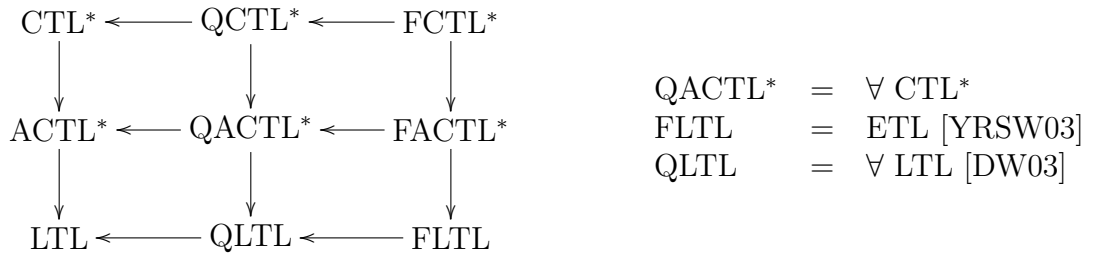
$\phi_{ADD}$	$\equiv \forall a : integer. \forall b : integer. \mathbf{AG} (x = a \wedge y = b \Rightarrow \mathbf{AX} z = a + b)$
$\phi_{MUTEX}$	$\equiv \forall i : Nat. \forall j : Nat. \mathbf{AG} ((at(i) = crit \wedge at(j) = crit) \Rightarrow i = j)$
$\phi_{LIVE}$	$\equiv \forall i : Nat. \mathbf{AG} (at(i) = think \Rightarrow \mathbf{AF}(at(i) = crit))$

Figure 3.2: Example Properties.  $\phi$  is a property of  $M_{TICKET}$ .

### 3.3 Discussion

**Characterization of Logics.** In [SRW02], safety properties are considered. The term safety property refers to properties of the form  $\mathbf{AG} e$  where  $e \in \mathcal{FO}_P$ . As such the framework supports quantification already. However, one cannot relate individuals in different states, i.e., one cannot handle quantification across time. Furthermore, safety properties cannot express liveness and reactivity properties of models. We want to verify such properties, hence we need to consider temporal logics.

The abstractions we use lead to overapproximating abstract models which have more branching behavior. We consider three temporal logics with a different capability of expressing branching behavior. LTL can be embedded into ACTL\*. ACTL\* is the universal fragment of CTL\* (see [CGP00]). We enrich these logics with first-order quantification across time. By augmenting the branching time temporal logic CTL\* with nested first-order quantification, one obtains the logic FCTL\* (one adds state formulas of the form  $\forall x : T. \phi$ ). Similarly, we obtain the logics FACTL\* and FLTL. FLTL corresponds to the logic ETL [YRSW03]. QCTL\* is the sublogic of FCTL\* which consists of formulas that are composed of a CTL\* formula with an arbitrary number of first-order quantifiers in front. The difference is that QCTL\* lacks *nesting* and existential quantification. Analogously, we obtain QACTL\* and QLTL. The quantifier elimination Skolemization removes the quantifiers of a QCTL\* formula and produces (possibly after several steps) a CTL\* formula. All these inclusions are depicted below:



Following an arrow means loss of expressiveness. The more one moves right the more restricted is our ability to quantify. The names of the logics express that. Arbitrarily nested quantification is expressed by an "F" in front. Universally quantified temporal properties are denoted with a "Q" in front. The more one moves down the less branching behavior is expressible. That ranges from CTL\*, which allows both universal and existential path quantification, to ACTL\*, which only has universal path quantifiers, to LTL, which consists of CTL\* path formulas with a universal path quantifier in front.

In [YRSW03], a varying-domain trace semantics of ETL is given. To relate individuals of different states in a trace and to express allocation and deallocation, states are anno-

tated with sets, and transitions are annotated with functions (cf. Discussion of Chapter 2). ETL cannot express branching, however, it allows for nested existential and universal quantification. [DW03] describes the verification of QLTL properties using finite instantiation and data type reduction.

In the next chapter, we will describe how QCTL\* problems can be reduced to CTL\* problems. Due to the analysis in Chapter 5, which introduces more behavior, Skolemization and the analysis allow us check QACTL\* properties.

**Varying-domain vs constant-domain semantics (continued).** Constant-domain semantics have the advantage that they make quantified temporal properties more expressive. A quantification  $\forall x : T. \phi$  in a QCTL\* property refers to the individuals within the initial states of a model. In a varying-domain semantics, it would be possible that individuals are added later on. We could not refer to such individuals using QCTL\* formulas, they would be "out of reach".

Assume that we are considering a mutual exclusion protocol where processes can be added and removed. We want to show liveness, i.e., globally every process will eventually attain its critical section

$$\mathbf{AG} \forall p : Proc. \mathbf{AF} at(p) = critical .$$

This property cannot be expressed using a quantified temporal formula if domains may vary. The formula is then in particular (unlike than in the constant-domain setting) *not equivalent* to

$$\forall p : Proc. \mathbf{AG} \mathbf{AF} at(p) = critical .$$

We do not discuss removal and addition in connection with the Ticket Protocol because we want to keep the examples short and simple. It is, however, possible to handle dynamic systems using our approach.

# Chapter 4

## Quantifier Elimination

If the semantic domain of a first-order universal quantifier is infinite, one cannot show every instance of a quantified formula. Finding a finite number of sufficient instances is one possibility. Symmetry arguments allow one to do this (cf. [DW03, McM00] and Chapters 6 and 7). However, symmetry is not a prerequisite of every quantifier elimination; in particular, the quantifier elimination we will present now does not rely on symmetry arguments. This plays a role if one wants to reason about arithmetics in a model ([McM00] makes use of uninterpreted functions to abstract away from arithmetics).

Skolemization eliminates quantifiers in front of FCTL\* formulas. As QCTL\* is a sublogic of FCTL\* (namely the sublogic that consists of universally-quantified CTL\* properties), we have a method that allows us to reduce QCTL\* problems to equivalent CTL\* problems. QCTL\* formulas have the form

$$\forall x_1 : T_1 \dots \forall x_k : T_k. \varphi(x_1, \dots, x_k) \quad (*)$$

where  $\varphi$  is a CTL\* formula. By repeated application quantification across time can be removed (for an explanation of the term "quantification across time" cf. Introduction to Chapter 3 and Definition 3.1.1). CTL\* formulas can contain nested first-order expressions, in so far CTL\* formulas need not be free of *any* quantification. (cf. Definition 3.1.1).

The quantifier elimination technique we use originates from theorem proving and is sometimes called Skolemization. We will also call it Skolemization. The idea is to instantiate the quantification variable with a fresh uninterpreted function symbol  $\zeta$  that ranges over the quantifier domain. The symbol  $\zeta$  is often called Skolem constant. Skolemization can be expressed by the tentative proof rule ( $\forall^\zeta$ ) :

$$\boxed{\frac{\zeta \text{ fresh, } M_\zeta \models \varphi(\zeta)}{M \models \forall x : T. \varphi(x)} \quad (\forall^\zeta)}$$

which will be formalized in Theorem 4.1.1. The model  $M_\zeta$  is the original model  $M$  with the Skolem constant  $\zeta$  added.  $\varphi$  has free variable  $x$ ; this is expressed by the notation  $\varphi(x)$ . We replace the free variable  $x$  with  $\zeta$  and obtain property  $\varphi(\zeta)$ . The proof rule says that the premise  $M_\zeta \models \varphi(\zeta)$  implies  $M \models \forall x : T. \varphi(x)$  (in fact, the converse also holds). The initial valuation of  $\zeta$  in  $M_\zeta$  is unconstrained and thus takes on every possible value in the quantification domain  $\Delta(T)$ . Furthermore, the valuation of  $\zeta$  does not change over time. The term uninterpreted constant expresses this notion. By proving that every

initial state of  $M_\zeta$  fulfills  $\varphi(\zeta)$ , we show that every instantiation of  $\forall x : T. \varphi(x)$  holds in  $M$ . A universally quantified formula holds if and only if all of its instances hold.

## 4.1 Skolemization

Skolemization is in principle very simple but the idea can easily get buried under notation. Maybe an example can best illustrate the technique.

**Example 4.1.1.** We consider model  $M_{ADD}$  (which corresponds to program *ADD* in Figure 1.1) and property  $\phi_{ADD}$  (cf. Figure 3.2). We need to apply Skolemization twice in order to remove the quantifiers. Instantiating quantifier variables  $a, b$  with fresh function symbols  $\zeta_1, \zeta_2$  yields the property:  $\overline{\phi}_{ADD} \equiv \mathbf{AG} (x = \zeta_1 \wedge y = \zeta_2 \Rightarrow \mathbf{AX} z = \zeta_1 + \zeta_2)$  and the model  $\overline{M}_{ADD}$  with the same syntactic description of the initial states as  $M_{ADD}$  and transitions  $\overline{\rho} \equiv z' = x + y \wedge \zeta_1' = \zeta_1 \wedge \zeta_2' = \zeta_2$  (the expression describing the transitions of  $M_{ADD}$  is  $z' = x + y$ ).

We want to specify that the Ticket Protocol  $M_{TICKET}$  is "live" ( $\phi_{LIVE}$  of Figure 3.2) - every process will eventually become critical. This can be expressed as:

$$\phi_{LIVE} \equiv \forall i : Proc. \mathbf{AG} (at(i) = think \Rightarrow \mathbf{AF} (at(i) = crit)) .$$

Let us apply quantifier elimination ( $\forall^\zeta$ ). We introduce a constant  $\zeta$  into our model. This yields a model  $\overline{M}_{TICKET} = \langle \overline{S}, \theta, \rho_\zeta \rangle$  where the transition relation is defined such that  $\zeta$  has a fixed valuation, i.e.  $\rho_\zeta \equiv \rho \wedge \zeta' = \zeta$ . The state-space of the original Ticket Protocol is augmented with  $\zeta$ , i.e.  $\overline{S} = \{ \langle \Delta, \iota \cup \{ \zeta \mapsto v \} \rangle \mid v \in \Delta(T), \langle \Delta, \iota \rangle \in S \}$  and the constraint that describes the initial states  $\theta$  is left unchanged. The modified property

$$\overline{\phi}_{LIVE} \equiv \mathbf{AG} (at(\zeta) = think \Rightarrow \mathbf{AF} (at(\zeta) = crit))$$

is checked against  $\overline{M}_{TICKET}$ . We have the following equivalence

$$M_{TICKET} \models \phi_{LIVE} \Leftrightarrow \overline{M}_{TICKET} \models \overline{\phi}_{LIVE} .$$

We have eliminated the quantifier and obtained an equivalent verification problem.

In the example, we have overlined models and properties obtained from Skolemization. However, when we argue about correctness of Skolemization, we are interested in the particular Skolem constant. We will denote models which result from Skolemization as  $M_\zeta$  where  $\zeta$  is the Skolem constant. We name  $M_\zeta$  the  $\zeta$ -augmentation of  $M$ . The  $\zeta$ -augmentation of  $M_{TICKET}$  is  $\overline{M}_{TICKET}$ . Analogously, for a property  $\phi = \forall x : T. \varphi$ , we call the property which results from Skolemization the  $\zeta$ -instance of  $\phi$ , denoted as  $\varphi(\zeta)$ . The  $\zeta$ -instance of  $\phi_{LIVE}$  is  $\overline{\phi}_{LIVE}$ . We do not annotate with an explicit Skolem constant outside this Chapter because that simplifies our notation, in particular if Skolemization has been applied more than once, and if names of models or properties already have a subscript, such as  $\phi_{LIVE}$ .

Again we consider the Skolemization proof rule ( $\forall^\zeta$ ).

$$\boxed{\frac{\zeta \text{ fresh, } M_\zeta \models \varphi(\zeta)}{M \models \forall x : T. \varphi(x)} \quad (\forall^\zeta)}$$

Now it should be more comprehensible, and we want formalize and prove it. The syntactical transformations used in Skolemization, namely the exact form of  $M_\zeta$  and  $\varphi(\zeta)$ , are given in the next definition.

**Definition 4.1.1.** Let  $\Sigma = \langle \mathcal{B}, \mathcal{F}, \mathcal{V}, r \rangle$  be a signature and  $\phi$  an  $FCTL_\Sigma^*$  formula of the form

$$\phi = \forall x : T. \varphi$$

where  $T \in \mathcal{B}$  is a base type. Let  $\zeta$  be a fresh symbol ( $\zeta \notin \mathcal{F} \cup P \cup \mathcal{V}$ ). Adding the fresh symbol  $\zeta$  to  $\Sigma$  yields the augmented signature  $\Sigma_\zeta = \langle \mathcal{B}, \mathcal{F} \cup \{\zeta\}, P, \mathcal{V}, r \cup \{\zeta : T\} \rangle$ . The  $\zeta$ -instance of  $\phi$  is the formula  $\varphi(\zeta)$  we obtain by replacing  $x$  with  $\zeta$  in  $\varphi$ , i.e.  $\varphi(\zeta) = \varphi[x/\zeta]$ .

Let  $M = \langle S, \rho, \theta \rangle \in \mathcal{M}_\Sigma$  be a model with state space  $S \subseteq \text{Struct}[\Sigma, \Delta]$ . We obtain a model  $M_\zeta = \langle S_\zeta, \rho_\zeta, \theta \rangle \in \mathcal{M}_{\Sigma_\zeta}$  with transitions  $\rho_\zeta = \rho \wedge \zeta' = \zeta$  and state space  $S_\zeta = \{ \langle \Delta, \iota \cup \{ \zeta \mapsto v \} \rangle \mid v \in \Delta(T), \langle \Delta, \iota \rangle \in S \}$ <sup>1</sup>. We call  $M_\zeta$  the  $\zeta$ -augmentation of  $M$ .

**Theorem 4.1.1 (Skolemization).**

Let  $M \in \mathcal{M}_\Sigma$  be model and  $\phi$  an  $FCTL_\Sigma^*$  formula of the form

$$\phi = \forall x : T. \varphi$$

where  $T \in \mathcal{B}$  is a base type. Then the equivalence

$$M \models \phi \quad \Leftrightarrow \quad M_\zeta \models \varphi(\zeta)$$

holds, where  $\varphi(\zeta)$  is the  $\zeta$ -instance of  $\phi$  and  $M_\zeta$  the  $\zeta$ -augmentation of  $M$  as described in Definition 4.1.1.

The idea underlying Skolemization reminds of case splitting in mathematics. One shows a universally quantified formula  $\phi = \forall x : T. \varphi$  by showing each of its instances, i.e. by inserting each possible value (values are called individuals in our setting). We do not insert values, instead we replace  $x$  with  $\zeta$  and set the instantiation value as a valuation of  $\zeta$  in a transition system. Let  $K$  be the transition system induced by model  $M$ . Furthermore, for an individual in the quantification domain we denote the transition system where the valuation of  $\zeta$  is fixed to  $u$  as  $K_u$ . A verification problem  $K_u \models \varphi(\zeta)$  is one instance of the original verification problem  $K \models \phi$ . The correctness argument of case splitting is the following equivalence:

$$K \models \phi \quad \Leftrightarrow \quad \forall u \in \Delta(T) : K_u \models \varphi(\zeta) \quad (\text{CASE}) .$$

The validity of  $M_\zeta \models \varphi(\zeta)$  is equivalent to the right side of the equivalence above. Why is that? The initial valuation of  $\zeta$  is unconstrained and hence initially every possible value is taken on. The valuation remains fixed. The transition system induced by  $M_\zeta$  is disjointly composed of the transition systems  $K_u$ . A model fulfills a property if all of its initial states do, and therefore if all  $K_u$  fulfill  $\varphi(\zeta)$ . The proof of Theorem 4.1.1 is based on the argument we have just given.

<sup>1</sup>Note that  $(\text{Struct}[\Sigma, \Delta])_\zeta = \text{Struct}[\Sigma_\zeta, \Delta]$  holds.

*Proof of Theorem 4.1.1.* Let  $S, S_\zeta$  be as in Definition 4.1.1. We abbreviate the transition systems resulting from the denotation  $K := \langle S, I, R \rangle := \llbracket M \rrbracket$  of  $M$ , and  $K_\zeta := \langle S_\zeta, I_\zeta, R_\zeta \rangle := \llbracket M_\zeta \rrbracket$  the transition system induced by  $M_\zeta$ .

For  $u \in \Delta(T)$  we denote the subset of  $S_\zeta$  in which the valuation of symbol  $\zeta$  is  $u$  as

$$S_u = \{ \langle \Delta, \iota \rangle \in S_\zeta \mid \iota(\zeta) = u \} .$$

We need this to define a transition system  $K_u$  which is identical to  $K_\zeta$  except that initially  $\zeta$  has valuation  $u$ :

$$K_u := \langle S_\zeta, R_u, I_u \rangle := \langle S_\zeta, \llbracket \rho_\zeta \rrbracket_{S_\zeta}, \llbracket I \rrbracket_{S_\zeta} \cap S_u \rangle \in \mathcal{K}_{S_\zeta} .$$

$K_u$  stands for a subproblem in a case splitting. The following equivalence holds

$$K \models \phi \quad \Leftrightarrow \quad \forall u \in \Delta(T) : K_u \models \varphi(\zeta) \quad (CASE) .$$

The equivalence (*CASE*) can be proved by a straightforward structural induction on  $\phi$ . It is an analog of case splitting as it is common in mathematical proofs. We check a first-order formula by inserting every possible value. Each insertion yields a subproblem, here  $K_u \models \varphi(\zeta)$ . The validity of the first-order formula is equivalent to the validity of all subproblems; this is what the equivalence (or proof rule) (*CASE*) expresses.

The claim follows by the following sequence of equivalences:

$$\begin{aligned} M \models \phi & \stackrel{\text{def}}{\Leftrightarrow} K \models \phi \\ & \stackrel{(CASE)}{\Leftrightarrow} \forall u \in \Delta(T) : K_u \models \varphi(\zeta) \\ & \stackrel{\text{def}}{\Leftrightarrow} \forall u \in \Delta(T) : I_u \subseteq \llbracket \varphi(\zeta) \rrbracket_{K_u} \\ & \stackrel{(*)}{\Leftrightarrow} \forall u \in \Delta(T) : I_u \subseteq \llbracket \varphi(\zeta) \rrbracket_{K_\zeta} \\ & \stackrel{(**)}{\Leftrightarrow} I_\zeta \subseteq \llbracket \varphi(\zeta) \rrbracket_{K_\zeta} \\ & \stackrel{\text{def}}{\Leftrightarrow} K_\zeta \models \varphi(\zeta) \\ & \stackrel{\text{def}}{\Leftrightarrow} M_\zeta \models \varphi(\zeta) \end{aligned}$$

thereby the following intermediate steps need to be explained:

(\*) For every formula  $\varphi \in FCTL_{\Sigma_\zeta}^*$  we have

$$\llbracket \varphi \rrbracket_{K_\zeta} = \llbracket \varphi \rrbracket_{K_u}$$

since the definition of  $\llbracket \cdot \rrbracket$  only depends on  $R_\zeta = R_u$ .

(\*\*) The initial states can be decomposed into a disjoint union:

$$I_\zeta = \bigcup_{u \in \Delta(T)} I_u .$$

□

**Rewriting formulas.** Some properties can be rewritten into QCTL\*. The benefit is that this makes them amenable to Skolemization. For example,  $\phi_{LIVE}$  is equivalent to

$$\mathbf{AG} (\forall i : Proc. at(i) = think \Rightarrow \mathbf{AF} (at(i) = crit)) .$$

$\phi_{LIVE}$  is a QCTL\* formula, the latter formula is not.

By "shifting the universal quantification outward" some properties which are not QCTL\* formulas can be transformed to equivalent QCTL\* formulas. This idea is expressed in the following Lemma:

**Lemma 4.1.2 (Rewrite Rules).** *Let  $K = \langle S, I, R \rangle$  be a transition system. Then we have:*

- $\llbracket \mathbf{AG} \forall x : T.\phi \rrbracket_K = \llbracket \forall x : T.\mathbf{AG} \phi \rrbracket_K$
- $\llbracket \mathbf{AX} \forall x : T.\phi \rrbracket_K = \llbracket \forall x : T.\mathbf{AX} \phi \rrbracket_K$
- $\llbracket \mathbf{AF} \forall x : T.\phi \rrbracket_K \subseteq \llbracket \forall x : T.\mathbf{AF} \phi \rrbracket_K$  and in general  $\supseteq$  does not hold.

Read from left to right, the equations above give rewrite rules for bringing formulas into a form that allows for Skolemization. The rewrite rules are not meant to be complete.

## 4.2 Predicate Logic Skolemization

Our predicate logic  $\mathcal{FO}_{\mathcal{P}}$  does not offer general function symbols. Skolemization introduces Skolem constants. As a result, we would obtain models which are not predicate logic models any more. So we need to find a variation of Skolemization that makes use of a predicate instead, in order to remain within predicate logic. The use of predicates complicates rewriting the property  $\phi$  compared to Skolemization for first-order logic. Having seen the more generic version of Skolemization, this transformation will, however, become clear.

First, let us look at the idea and then at the technical details. We can exploit that predicate logic is an instance of first-order logic. So a model  $M$  in predicate logic can be seen as a particular model of general first-order logic. We can apply Skolemization (cf. Theorem 4.1.1) and obtain a general first-order model  $\overline{M}$  which is not a predicate model anymore because it has been augmented with a Skolem constant  $\zeta$ . There is machinery (cf. B.2) that allows us to encode general first-order models and general first-order properties in predicate logic such that we lose no information, i.e. a one-to-one correspondence. The result of these steps is an equivalent verification problem where the quantifier is gone. This was just a plausibility argument. There is no need for such a sequence of conversions in practice.

Predicate logic Skolemization can again be described as a tentative proof rule:

$$\boxed{\frac{\eta \text{ fresh, } M[\eta] \models \varphi[\eta]}{M \models \forall x. \varphi(x)} \quad (\forall^n)}$$

where  $\varphi[\eta]$  is  $\varphi(x)$  with each occurrence of a first-order expression  $e$  with  $x \in FV(e)$  replaced with the first-order expression  $\exists u. \eta(u) \wedge e[x/u]$ . The technical difference is that  $\eta$  is now a unary predicate rather than a constant.  $M[\eta]$  is  $M$  with the transition relation  $\rho$  changed to  $\rho[\eta] = \rho \wedge \forall u. \eta'(u) = \eta(u)$ .

**Definition 4.2.1.** Let  $\mathcal{P} = \langle P, \mathcal{V}, r \rangle$  be a signature and  $\phi$  an  $FC\text{TL}^*$  formula of the form

$$\phi = \forall x. \varphi .$$

Let  $\eta$  be a fresh symbol ( $\eta \notin P \cup \mathcal{V}$ ). We call  $\eta$  a Skolem predicate. Adding the fresh predicate symbol  $\eta$  to  $\mathcal{P}$  yields the augmented predicate signature  $\mathcal{P}[\eta] = \langle P \cup \{\eta\}, \mathcal{V}, r \cup \{\eta \mapsto 1\} \rangle$ . The predicate logic  $\eta$ -instance of  $\phi$  is the formula  $\varphi[\eta]$  we obtain by replacing each occurrence of a first-order expression  $e$  with  $x \in FV(e)$  with the first-order expression  $\exists u. \eta(u) \wedge e[x/u]$ . We express this transformation with the expression-level transformation function  $\tau_{\mathcal{P}, \mathcal{P}[\eta]}$  from Figure 4.1 by setting

$$\varphi[\eta] = \tau_{\mathcal{P}, \mathcal{P}[\eta]}(\varphi, \lambda e. \text{if } (x \in FV(e)) (\exists u. \eta(u) \wedge e[x/u]) \text{ else } e)$$

where  $u \in \mathcal{V}$  is a fresh variable which does not appear in  $\phi$ .

Let  $M = \langle S, \rho, \theta \rangle \in \mathcal{M}_{\mathcal{P}}$  be a model with state space  $S \subseteq \text{Struct}[U]$ . We obtain a model  $M[\eta] = \langle S[\eta], \rho[\eta], \theta \rangle \in \mathcal{M}_{\mathcal{P}[\eta]}$  with transitions  $\rho[\eta] = \rho \wedge \forall u. \eta'(u) = \eta(u)$  and state space  $S[\eta] = \{ \langle \Delta, \iota \cup \{\eta \mapsto v\} \rangle \mid v \in U, \langle U, \iota \rangle \in S \}$ . We call  $M[\eta]$  the predicate logic  $\eta$ -augmentation of  $M$ .

**Theorem 4.2.1 (Predicate Logic Skolemization).**

Let  $M \in \mathcal{M}_{\mathcal{P}}$  be model and  $\phi$  an  $FC\text{TL}_{\mathcal{P}}^*$  formula of the form

$$\phi = \forall x. \varphi .$$

Then the equivalence

$$M \models \phi \quad \Leftrightarrow \quad M[\eta] \models \varphi[\eta]$$

holds, where  $\varphi[\eta]$  is the  $\eta$ -instance of  $\phi$  and  $M[\eta]$  the predicate logic  $\eta$ -augmentation of  $M$  as described in Definition 4.1.1.

*Proof.* The proof can be found in Appendix B.2. □

Predicate logic Skolemization introduces existential quantifiers only at the level of expressions. It does *not* introduce quantification across time. Therefore, one can reduce QCTL\* verification problems to equivalent CTL\* verification problems by using Skolemization.

In many cases, multiple quantifiers occur in front of a temporal formula. We consider a simple example formula and apply predicate logic Skolemization.

**Example 4.2.1.** We will give a more concise description of how the Ticket Protocol can be modeled in predicate logic in Section 5.3. Here we give only a short description which should be sufficient to understand the transformations of predicate logic Skolemization. Types are modeled as unary predicates: predicate symbol *process* stands for processes, and predicate symbol *number* for tickets. The local variable "a" of each process is the ticket of the particular process. This is modeled by a binary predicate *a*. The ticket of a process *p* is the number *j* such that  $a(p, j)$ . The following property expresses that two numbers have distinct tickets:



$$\phi_{UNEQ} = \forall p_1. \forall p_2. \forall j. \quad (\text{number}(j) \wedge \text{process}(p_1) \wedge \text{process}(p_2) \wedge p_1 \neq p_2) \Rightarrow \mathbf{AG}(\neg(a(p_1, j) \wedge a(p_2, j)))$$

The property  $\phi_{UNEQ}$  contains two nested first-order expressions  $e_1, e_2$  and its structure is  $\forall p_1. \forall p_2. \forall j. e_1 \Rightarrow \mathbf{AG} e_2$  where

$$\begin{aligned} e_1 &= (\text{number}(j) \wedge \text{process}(p_1) \wedge \text{process}(p_2) \wedge p_1 \neq p_2) \\ e_2 &= (\neg(a(p_1, j) \wedge a(p_2, j))) . \end{aligned}$$

We eliminate the first quantifier, the one with quantification variable  $p_1$ . We compute the  $\eta_1$ -instance of  $\phi_{UNEQ}$ . Variable  $p_1$  is free in  $e_1$  and  $e_2$ . We demonstrate the transformations on the two first-order expressions  $e_1$  and  $e_2$ . The transformations produce first-order expressions  $e'_i = \exists u_1. \eta_1(u_1) \wedge e_i[p_1/u_1]$  for  $i = 1, 2$ :

$$\begin{aligned} e'_1 &= \exists u_1. \eta_1(u_1) \wedge e_1[p_1/u_1] \\ &= \exists u_1. \eta_1(u_1) \wedge \text{number}(j) \wedge \text{process}(u_1) \wedge \text{process}(p_2) \wedge u_1 \neq p_2 \\ e'_2 &= \exists u_1. \eta_1(u_1) \wedge e_2[p_1/u_1] \\ &= \exists u_1. \eta_1(u_1) \wedge \neg(a(u_1, j) \wedge a(p_2, j)) \end{aligned}$$

The  $\eta_1$ -instance of  $\phi_{UNEQ}[\eta_1]$  is  $\forall p_2. \forall j. e'_1 \Rightarrow \mathbf{AG} e'_2$ . Now we eliminate the first quantifier in  $\phi_{UNEQ}[\eta_1]$ , the quantifier with quantification variable  $p_2$ . We compute first-order expressions  $e''_i = \exists u_2. \eta_2(u_2) \wedge e'_i[p_2/u_2]$  for  $i = 1, 2$ :

$$\begin{aligned} e''_1 &= \exists u_2. \eta_2(u_2) \wedge e'_1[p_2/u_2] \\ e''_2 &= \exists u_2. \eta_2(u_2) \wedge e'_2[p_2/u_2] \end{aligned}$$

where  $e'_1[p_2/u_2] = \exists u_1. \eta_1(u_1) \wedge \text{number}(j) \wedge \text{process}(u_1) \wedge \text{process}(u_2) \wedge u_1 \neq u_2$  and  $e'_2[p_2/u_2] = \exists u_1. \eta_1(u_1) \wedge \neg(a(u_1, j) \wedge a(u_2, j))$ . The  $\eta_2$ -instance of  $\phi_{UNEQ}[\eta_1]$  is  $\phi_{UNEQ}[\eta_1][\eta_2] = \forall j. e''_1 \Rightarrow \mathbf{AG} e''_2$ . We can remove the quantifier in front of  $\phi_{UNEQ}[\eta_1][\eta_2]$  analogously and finally we obtain the ACTL\* property

$$\phi_{UNEQ}[\eta_1][\eta_2][\eta_3] = \forall j. e'''_1 \Rightarrow \mathbf{AG} e'''_2$$

with

$$\begin{aligned} e'''_1 &= \exists u_3. \eta_3(u_3) \wedge e''_1[j/u_3] \\ &= \exists u_3. \eta_3(u_3) \wedge \\ &\quad \exists u_2. \eta_2(u_2) \wedge \\ &\quad \exists u_1. \eta_1(u_1) \wedge \text{number}(u_3) \wedge \text{process}(u_1) \wedge \text{process}(u_2) \wedge u_1 \neq u_2 \\ e'''_2 &= \exists u_3. \eta_3(u_3) \wedge e''_2[j/u_3] \\ &= \exists u_3. \eta_3(u_3) \wedge \\ &\quad \exists u_2. \eta_2(u_2) \wedge \\ &\quad \exists u_1. \eta_1(u_1) \wedge \neg(a(u_1, u_3) \wedge a(u_2, u_3)) . \end{aligned}$$

We have so far neglected the model. Let  $M_{TICKET} = \langle S, \theta, \rho \rangle \in \mathcal{M}_{\mathcal{P}}$  be the model that corresponds to the Ticket Protocol. The  $\eta_1$ -augmentation of  $M_{TICKET}$  is  $M_{TICKET}[\eta_1] = \langle S_{\eta_1}, \theta, \rho \wedge \forall v. \eta'(v) = \eta(v) \rangle$ . Applying  $\eta_2$  and  $\eta_3$ -augmentation to this model, we get

$$M_{TICKET}[\eta_1, \eta_2, \eta_3] = \left\langle \begin{array}{l} (S_{\eta_1})_{\eta_2})_{\eta_3}, \quad \theta, \\ \rho \quad \wedge \quad \forall v. \eta'_1(v) = \eta_1(v) \\ \quad \wedge \quad \forall v. \eta'_2(v) = \eta_2(v) \\ \quad \wedge \quad \forall v. \eta'_3(v) = \eta_3(v) \end{array} \right\rangle .$$

$\phi_{UNEQ}[\eta_1][\eta_2][\eta_3]$  is an ACTL\* property. Theorem 4.2.1 guarantees that

$$M_{TICKET}[\eta_1, \eta_2, \eta_3] \models \phi_{UNEQ}[\eta_1][\eta_2][\eta_3] \Leftrightarrow M_{TICKET} \models \phi_{UNEQ}$$

We have thus entirely eliminated quantification across time and reduced a QACTL\* verification problem to an ACTL\* verification problem. We have introduced existential quantification at the level of expressions only.

As we have seen in the previous example, when Skolemization is applied three times the resulting formula contains subexpressions of the form

$$e = \exists u_3. \eta_3(u_3) \wedge \exists u_2. \eta_2(u_2) \wedge \exists u_1. \eta_1(u_1) \wedge e_1$$

where  $e_1$  is the original subexpression before Skolemization. We can rewrite this to the equivalent expression

$$e' = \exists u_3. \exists u_2. \exists u_1. \eta_1(u_1) \wedge \eta_2(u_2) \wedge \eta_3(u_3) \wedge e_1$$

where the newly introduced quantifiers appear in a cascaded form. The cascaded form can be more readable. The following proof rule describes Skolemization for multiple quantifiers such that subexpressions with cascaded existential quantification, as in  $e'$ , are produced.

$$\boxed{\frac{\eta_1, \dots, \eta_n \text{ fresh, } M[\eta_1, \dots, \eta_n] \models \varphi'[\eta_1, \dots, \eta_n]}{M \models \forall x_1. \dots \forall x_n. \varphi(x_1, \dots, x_n)} \quad (\forall^{\eta_1, \dots, \eta_n})}$$

where  $\varphi'[\eta_1, \dots, \eta_n]$  is  $\varphi$  with each occurrence of a first-order expression  $e$  such that  $\{x_{i_1}, \dots, x_{i_n}\} \subseteq FV(e)$  replaced with the first-order expression

$$\exists x_{i_1}. \dots \exists x_{i_n}. \eta_{i_1}(x_{i_1}) \wedge \dots \wedge \eta_{i_n}(x_{i_n}) \wedge e[x_{i_j}/u_{i_j}].$$

### 4.3 Discussion

Skolemization<sup>2</sup> removes universal quantifiers in front of temporal formulas and has been described in, e.g., [Ber02, OS03]. Skolemization is related to finite instantiation [McM00, DW03] and the decomposition of [YR04]. Finite instantiation enumerates particular individuals of a fixed universe while Skolemization instantiates with symbols. Methods performing finite instantiation compute a finite number of sufficient instances of a QLTL property, *provided* that the system obeys certain syntactic restrictions (the syntactic restrictions are not too severe because of the character of the abstraction used in [DW03, McM00]). The instantiation decomposes one QLTL verification problem into several LTL problems, the number of which is determined by the possible equality relations among the quantification variables. We will see that models of [SRW02] also fulfill the syntactic restrictions. However, these models have a varying-domain semantics and individuals are anonymous. Skolemization instantiates with symbols. This can be useful (future work) if one wants to prove the following method correct: [YR04] instantiates with individuals of three-valued structures. As finite instantiation assumes a constant-domain semantics and enumerates particular concrete individuals, [YR04] is not a special case of finite instantiation (see Section 5.4).

<sup>2</sup>This is a name clash with the quantifier elimination for existential quantifiers which is used for the computation of the prenex form.

**Implementation.** We want to use the framework [SRW02] and methods from [YRS03] for computing abstract transition systems. We therefore need to implement the state space transformation (from  $S$  to  $S_\eta$ ) of predicate logic Skolemization by means of constraints. When we use [SRW02] the state space  $S$  of a the concrete model  $M$  is implicitly determined through constraints. We consider constant-domain models with a state space of the form  $S = \{s \in Struct[\mathcal{P}, U] \mid s \models \mathfrak{R}\}$ . The constraints  $\mathfrak{R}$  pertain to the model  $M$  and  $S$  is determined by the set of constraints  $\mathfrak{R}$ . We have to model the state space of the model  $M[\eta]$  using constraints. Fortunately, one can let  $M[\eta]$  inherit the constraints  $\mathfrak{R}$  of  $M$  plus an addition. It is sufficient to add constraints which insure that a Skolem predicate  $\eta$  always refers to exactly one individual of the universe. In the language of [SRW02], these are the constraints  $\mathfrak{r}_{\eta,1} \equiv \eta(u_1) \wedge \eta(u_2) \triangleright u_1 = u_2$  and  $\mathfrak{r}_{\eta,2} \equiv \forall u. \neg \eta(u) \triangleright 0$ . The constraints of model  $M[\eta]$  are  $\mathfrak{R}[\eta] = \mathfrak{R} \cup \{\mathfrak{r}_{\eta,1}, \mathfrak{r}_{\eta,2}\}$ . We get  $M[\eta] = \langle \{s \in Struct[\mathcal{P}[\eta], U] \mid s \models \mathfrak{R}[\eta]\}, \theta, \rho[\eta] \rangle$ , in particular the constraints exactly model Definition 4.2.1, i.e.

$$S[\eta] = \{s \in Struct[\mathcal{P}[\eta], U] \mid s \models \mathfrak{R}[\eta]\} .$$

Thus we have implemented Skolemization as an entirely syntactic transformation.

As predicate logic is one-sorted, it is sometimes useful to introduce sorts by unary predicates which have a constant valuation in the model. Let us assume that  $p$  is such a predicate. One wants to write  $\forall x : p. \varphi(x)$  which is notation for  $\forall x. p(x) \wedge \varphi(x)$ . However, Theorem 4.2.1 makes no statement about such properties. One can, however, prove such a variation. For this purpose one has to modify instantiation. Instead of replacing each occurrence of a first-order expression  $e$  with  $x \in FV(e)$  with the first-order expression  $\exists u. \eta(u) \wedge e[x/u]$ , one replaces  $e$  with  $\exists u. p(u) \wedge \eta(u) \wedge e[x/u]$ .

**Applicability of Skolemization.** Live Sequence Charts [DH01] can be translated to QLTL formulas [DW03]. Therefore, Skolemization can be used for the verification of UML models. Quantified temporal properties such as  $\phi_{LIVE}$  and quantified invariants such as  $\phi_{MUTEX}$  are amenable to Skolemization. Some properties which are not QCTL\* properties can be transformed to equivalent QCTL\* formulas with the rewrite rules of Lemma 4.1.2.

**Skolemization and abstraction** Skolemization is independent of a particular abstraction technique. This allows one to deploy different finitary abstraction techniques. It seems natural to tailor the abstraction to the function symbols introduced by Skolemization, since these function symbols occur in the property we want to prove.

$$\begin{array}{ccc}
M, \phi & \xrightarrow{\text{eliminate quantifiers}} & \overline{M}, \overline{\phi} & \xrightarrow{\text{abstraction}} & \widetilde{M}, \widetilde{\phi} \\
M \models \phi & \xleftarrow{\text{equivalent}} & \overline{M} \models \overline{\phi} & \xleftarrow{\text{implication}} & \widetilde{M} \models \widetilde{\phi}
\end{array}$$

One interesting effect of case splitting, decomposition, and Skolemization is that they shrink the scope a property refers to. [YR04] proposes to use heterogeneous abstractions to benefit from this effect. Heterogeneous abstraction means that for distinguished regions of the universe more information is kept than elsewhere. Skolemization produces regions

The expression-level transformation function:

$$\tau_{\Sigma, \Sigma'} \in FCTL_{\Sigma}^* \times (\mathcal{FO}_{\Sigma} \rightarrow \mathcal{FO}_{\Sigma'}) \rightarrow FCTL_{\Sigma'}^*$$

is defined by (for brevity we omit the signature subscripts):

$$\tau(\phi, f) = \begin{cases} f(e) & ; \phi = e \\ \neg(\tau(\phi', f)) & ; \phi = \neg\phi' \\ (\tau(\phi_1, f)) \wedge (\tau(\phi_2, f)) & ; \phi = \phi_1 \wedge \phi_2 \\ (\tau(\phi_1, f)) \vee (\tau(\phi_2, f)) & ; \phi = \phi_1 \vee \phi_2 \\ \mathbf{A}(\tau(\Phi, f)) & ; \phi = \mathbf{A}\Phi \\ \mathbf{E}(\tau(\Phi, f)) & ; \phi = \mathbf{E}\Phi \\ \forall x. (\tau(\phi, f)) & ; \phi = \forall x. \phi \end{cases}$$

$$\tau(\Phi, f) = \begin{cases} \tau(\phi, f) & ; \Phi = \phi \\ \neg(\tau(\Phi', f)) & ; \Phi = \neg\Phi' \\ (\tau(\Phi_1, f)) \wedge (\tau(\Phi_2, f)) & ; \Phi = \Phi_1 \wedge \Phi_2 \\ (\tau(\Phi_1, f)) \vee (\tau(\Phi_2, f)) & ; \Phi = \Phi_1 \vee \Phi_2 \\ \mathbf{X}(\tau(\Phi', f)) & ; \Phi = \mathbf{X}\Phi' \\ (\tau(\Phi_1, f))\mathbf{U}(\tau(\Phi_2, f)) & ; \Phi = \Phi_1\mathbf{U}\Phi_2 \\ (\tau(\Phi_1, f))\mathbf{R}(\tau(\Phi_2, f)) & ; \Phi = \Phi_1\mathbf{R}\Phi_2 \end{cases}$$

$\tau$  is given two arguments: a temporal property and a function  $f$  which transforms expressions. The function recursively descends to expressions and applies  $f$  to expressions and returns the resulting property.

Figure 4.1: Expression-level transformation function  $\tau$ .

of particular interest, namely the surroundings of an individual referenced by a Skolem constant, as this is the region a property that has evolved from Skolemization directly refers to. The surroundings of an individual are the individuals connected with it via binary predicates, e.g., a ticket  $i$  belongs to the surrounding of process  $p$  if it is connected to  $p$  by  $a(p, i)$ . Data type reduction of [McM00] is an example of heterogeneous abstraction (more on that in Chapter 7).

# Chapter 5

## Analysis

The goal of this chapter is to obtain an analysis for QACTL\* properties using Three-Valued Logical Analysis [SRW02]. Skolemization is a symbolic method, while the analysis technique we use is explicit-state. We model Skolemization by nondeterministic choice and the constraint mechanism described in [SRW02]. The transition system computed by our analysis is an approximation of the model one obtains from Skolemization. The chapter is concluded by a case study we conducted using the Three-Valued Logical Analyzer (TVLA).

How does our QACTL\* approach work? We are given a QACTL\* property  $\phi$  (cf. 3.1.2) and a model  $M \in \mathcal{M}_{\mathcal{P}}$  denoted in predicate logic (cf. Definitions 2.2.3 and 2.3.1). The quantifier elimination technique of Chapter 4 gives us an equivalent verification problem consisting of a model  $\overline{M} \in \mathcal{M}_{\overline{\mathcal{P}}}$  and an ACTL\* formula  $\overline{\phi}$ . The analysis we will present in Section 5.2 allows us to solve this verification problem. In order to make verification feasible, we use canonical abstraction. Canonical abstraction maps a logical structure to a bounded logical structure by collapsing individuals to equivalence classes. The computation of the equivalence classes is based on the valuation of individuals under one-argument predicates. The equivalence classes form the universe of the resulting bounded structure. Canonical abstraction resolves conflicting valuations of predicate symbols within an equivalence class of individuals by introducing a third truth value  $1/2$  (it stands for *unknown*) and by using a join-operator. Logical structures with the third value are called three-valued logical structures, because the valuation of a predicate symbol may yield  $1/2$ . We obtain a transition system over three-valued logical structures by executing the concrete model  $\overline{M}$  over three-valued logical structures. We want to prove that the model  $\overline{M}$  fulfills the ACTL\* property  $\overline{\phi}$ . Therefore it is necessary to talk about property preservation. We achieve property preservation by means of a simulation preorder on three-valued logical structures. The transition system resulting from evaluating  $\overline{M}$  over three-valued logical structures simulates the transition system  $\llbracket \overline{M} \rrbracket$ , the semantics of  $\overline{M}$ .

### 5.1 Three-Valued Analysis

We discuss three-valued logical structures, which are the abstract states of the transition systems on which we check temporal properties. An information order on three-valued logical structures will, among other things, allow us to relate abstract states with con-

crete states. The use of bounded three-valued logical structures ensures termination of the analysis algorithm. Canonical abstraction is a means to obtain bounded structures. All of these definitions are excerpts from [SRW02] (cf. the Kleene domain 5.1.4, three-valued logical structures 5.1.5, the Embedding Order 5.1.7, tight embeddings 5.1.9, the Embedding Theorem 5.1.1, Canonical Abstraction 5.1.11). Since we consider general temporal logic properties and not only safety properties, we relate transition systems via a simulation preorder (cf. Definition 5.1.12, and Theorem 5.1.2).

**Basics.** We need some basic notions such as partial orders, join-lattices, and the Kleene domain in order to proceed.

**Definition 5.1.1 (Partial ordering).** Let  $L$  be a set and  $\sqsubseteq \subset L \times L$  a binary relation on  $L$ .  $\sqsubseteq$  is a partial ordering of  $L$  iff it is reflexive (i.e.  $\forall l \in L : l \sqsubseteq l$ ), transitive (i.e.  $\forall l_1, l_2, l_3 \in L : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3$ ), and antisymmetric, (i.e.  $\forall l_1, l_2 \in L : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_1 \Rightarrow l_1 = l_2$ ).

For  $l_1, l_2 \in L$  the expression  $l_2 \sqsupseteq l_1$  is a syntactic equivalent for  $l_1 \sqsubseteq l_2$ .

A tuple  $\langle L, \sqsubseteq \rangle$  is a partially-ordered set iff  $\sqsubseteq$  is a partial ordering of  $L$ .

**Definition 5.1.2 (Lower and Upper Bounds).** Let  $\langle L, \sqsubseteq \rangle$  be a partially-ordered set. Let  $Y \subset L$ . An element  $l \in L$  is an upper bound of  $Y$  iff  $\forall y \in Y : y \sqsubseteq l$ . An element  $l \in L$  is a lower bound of  $Y$  iff  $\forall y \in Y : l \sqsubseteq y$ . An upper bound  $l$  of  $Y$  such that for all upper bounds  $u$  holds  $l \sqsubseteq u$  is called least upper bound of  $Y$ , written  $\sqcup Y$ . A lower bound  $l$  of  $Y$  such that for all lower bounds  $u$  holds  $l \sqsupseteq u$  is called greatest lower bound of  $Y$ , written  $\sqcap Y$ . An upper bound  $l$  of  $Y$  such that for all lower bounds  $u$  holds  $u \sqsupseteq l$  is called least upper bound of  $Y$ , written  $\sqcup Y$ .

Note that a set  $Y \subseteq L$  can have at most one least upper bound and greatest lower bound, respectively.

**Definition 5.1.3 (Lattice).** A join-lattice is partially order set  $\langle L, \sqsubseteq \rangle$  such that all subsets  $Y$  of  $L$  have a least upper bound  $\sqcup Y$ .

A complete lattice is a partially-ordered set  $\langle L, \sqsubseteq \rangle$  such that all subsets  $Y$  of  $L$  have a least upper bound  $\sqcup Y$  and a greatest lower bound  $\sqcap Y$ .

For any set  $X$  its power set  $\mathcal{P}(X)$  together with the subset order  $\subseteq$  is a complete lattice  $\langle \mathcal{P}(X), \subseteq \rangle$ .

**Definition 5.1.4 (Kleene domain).** We define the Kleene domain  $\mathbb{K} = \{0, 1, 1/2\}$ . We consider two orders on  $\mathbb{K}$ : the partial order  $0 \sqsubseteq 1/2, 1 \sqsubseteq 1/2$ , called **information order** (the greatest value is the indefinite value), and the **total order**  $\leq$  given by  $0 \leq 1/2 \leq 1$ .

We denote the least upper bound and greatest lower bound operator with respect to information order as  $\sqcup, \sqcap$ . For the least upper bound and greatest lower bound operator with respect to  $\leq$ , we use the common syntax  $\max, \min$ .

Sometimes it is convenient to interpret elements of  $\mathbb{K}$  as rational numbers,  $\mathbb{K} = \{0, 1, 1/2\} \subseteq \mathbb{Q}$ .  $1 \in \mathbb{K}$  can be interpreted as  $1 \in \mathbb{Q}$ ,  $1/2 \in \mathbb{K}$ , which stands for *unknown*, is interpreted as the fraction  $\frac{1}{2} \in \mathbb{Q}$ , and  $0 \in \mathbb{K}$  as  $0 \in \mathbb{Q}$ . The minimum and

maximum of a set of Kleene values, the functions  $\min, \max \in \mathcal{P}(\mathbb{K}) \rightarrow \mathbb{K}$ , are the greatest lower bound and least upper bound operators, respectively, with respect to the total order  $\leq$ . They correspond to the minimum and maximum induced by the inclusion of  $\mathbb{K}$  in the rational numbers. Note that for a Kleene value  $k \in \mathbb{K}$  the arithmetic expression  $(1 - k) \in \mathbb{K}$  interpreted in the rational numbers yields a Kleene value, because

$$1 - 0 = 1 \quad 1 - 1/2 = 1/2 \quad 1 - 1 = 0 .$$

Observation:  $\langle \mathbb{K}, \sqsubseteq \rangle, \langle \mathbb{K}, \leq \rangle$  are join-lattices.

**Three-valued structures.** Two-valued predicate logical structures give predicate symbols a valuation that maps into the Boolean values. Three-valued structures give predicate symbols a valuation that maps into the Kleene values. As the Boolean values 0, 1 are special Kleene values, two-valued logical structures of predicate logic are special three-valued logical structures.

**Definition 5.1.5 (Three-valued Logical Structure).** *A three-valued logical structure over predicate signature  $\mathcal{P} = \langle \{sm\} \cup P, \mathcal{V}, r \rangle$  is a tuple  $\langle U, \iota \rangle$  where  $U$  is a set ( $U$  for universe), and  $\iota$  is an interpretation function such that  $\iota(p) \in U^{r(p)} \rightarrow \mathbb{K}$ . We denote the state of three-valued logical structures over signature  $\mathcal{P}$  as  $3Struct[\mathcal{P}]$ .*

The special predicate  $sm$  expresses that an individual represents multiple concrete individuals. We define a semantics of predicate logic expressions  $e \in \mathcal{FO}_{\mathcal{P}}$  on three-valued structures.

**Semantics.** The valuation of a first-order expression on a logical structure can be defined almost as before <sup>1</sup>. The interpretation of the equality symbol is changed. The meaning of equality is defined in terms of the  $sm$  predicate and the *identically-equal* relation on individuals (denoted by the symbol  $=$ ):

- Nonidentical individuals are unequal.
- A non-summary individual must be equal to itself.
- In all other cases, we say 1/2.

**Definition 5.1.6 (Semantics).** *The valuation of  $[e] s Z$  of an expression  $e$  in a state  $s = \langle U, \iota \rangle \in 3Struct[\mathcal{P}]$  and with the complete assignment  $Z$  is an element of  $\mathbb{K}$ . We can re-use the inductive definitions of Definition 2.1.3 with the change:*

$$[x_1 = x_2] s Z = \begin{cases} 0 & ; \quad Z(x_1) \neq Z(x_2) \\ 1 & ; \quad Z(x_1) = Z(x_2) \text{ and } \iota(sm)(x_1) = 0 \\ 1/2 & ; \quad \text{otherwise} \end{cases}$$

---

<sup>1</sup>We can evaluate expressions  $1 - k$  where  $k \in \mathbb{K}$  and the minimum and maximum functions for the Kleene domain.

We term transition systems  $K \in \mathcal{K}_S$  over a state space  $S \subseteq 3Struct[\mathcal{P}]$  of three-valued logic structures *abstract transition systems*. We need an ACTL\* semantics on those transition systems. We take the ACTL\* semantics from Definition 3.2 and evaluate it on three-valued logical structures. The difference between concrete transition systems and abstract transition systems is that a first-order expression evaluated in a state may result in an indefinite result 1/2. The indefinite 1/2 value is interpreted as false:

$$K, s \models e \quad :\Leftrightarrow \quad ([e] s \emptyset) = 1 .$$

This is because we want to prove properties.

**Embedding Order.** We will now recapture the notion of information order on three-valued logical structures introduced in [SRW02]. Two-valued structures are three-valued structures, too. The inclusion  $Struct[\mathcal{P}] \subseteq 3Struct[\mathcal{P}]$  holds. Therefore embeddings also relate two-valued with three-valued structures.

**Definition 5.1.7 (Embedding Order).** *Let  $s, s' \in 3Struct[\mathcal{P}]$  where  $s = (U^s, \iota^s)$ ,  $s' = (U^{s'}, \iota^{s'})$ . Let  $f \in U^s \rightarrow U^{s'}$  be a surjective function. We say that  $f$  **embeds**  $s$  into  $s'$ , denoted as  $s \sqsubseteq_f s'$ , iff*

(i) *for all predicate symbols  $p \in P$  of arity  $n = r(p)$  and all  $u_1, \dots, u_n \in U^s$*

$$\iota^s(p)(u_1, \dots, u_n) \sqsubseteq \iota^{s'}(p)(f(u_1), \dots, f(u_n))$$

(ii)  $(|\{u \mid f(u) = u'\}| > 1) \sqsubseteq \iota^{s'}(sm)(u')$

*hold.*

*We say that  $s$  can be embedded into  $s'$ , denoted as  $s \sqsubseteq s'$ , if there exists an embedding function  $f$  such that  $s \sqsubseteq_f s'$ .*

The use of metavariable  $f$  for embedding functions collides with the use of this metavariable for function symbols. The reason why we use  $f$  nevertheless is that this is the notation used in [SRW02] and because we are currently only working with predicate logic. General function symbols do not occur in predicate logic, and predicate symbols are denoted with metavariable  $p$ .

$\sqsubseteq$  is a preorder, i.e. it is a reflexive and transitive relation. The concretization of a three-valued logical structure is the (possibly infinite) set of logical structures it represents.

**Definition 5.1.8 (Concretization).** *We define the concretization function*

$$\gamma_3 \in 3Struct[\mathcal{P}] \rightarrow Struct[\mathcal{P}], \quad \gamma_3(\tilde{s}) = \{s \in Struct[\mathcal{P}] \mid s \sqsubseteq \tilde{s}\} .$$

**Tight Embedding.** A tight embedding is a special kind of embedding  $s \sqsubseteq_f s'$  where information loss is minimized when multiple individuals are mapped to the same individual.

**Definition 5.1.9.** *Let  $s, s' \in 3Struct[\mathcal{P}]$  where  $s = (U^s, \iota^s)$ ,  $s' = (U^{s'}, \iota^{s'})$ . Let  $f \in U^s \rightarrow U^{s'}$  be a surjective function.  $s'$  is a **tight embedding** iff*



(i) for all predicate symbols  $p \in P$  of arity  $n = r(p)$  and all  $u_1, \dots, u_n \in U^s$

$$\iota^{s'}(p)(u'_1, \dots, u'_k) = \bigsqcup \{ \iota^s(p)(u_1, \dots, u_k) \mid u_1, \dots, u_k \in U, f(u_1) = u'_1, \dots, f(u_k) = u'_k \}$$

$$(ii) \iota^{s'}(sm)(u') = (|\{u \mid f(u) = u'\}| > 1) \sqcup \bigsqcup_{u \in U, f(u)=u'} \iota^s(sm)(u) .$$

$s'$  is uniquely determined by  $s$  and  $f$  and we write  $s' = f(s)$ .

It is clear from Definition 5.1.7 that the tight embedding of a structure  $s$  by a function  $f$  embeds  $s$  in  $s'$ , i.e.  $s \sqsubseteq_f f(s)$ .

**The Embedding Theorem.** The Embedding Theorem ensures that evaluating a predicate logic expression over a logical structure  $s'$  is safe with respect to the valuation of the expression over the logical structures  $s \sqsubseteq s'$  it approximates ( $\sqsubseteq$  thereby denotes the embedding order).

Informally, the Embedding Theorem says:

If  $s \sqsubseteq_f s'$ , then validity/invalidity of  $e$  in  $s'$  implies validity/invalidity of  $e$  in  $s$ .

**Theorem 5.1.1 (Embedding Theorem).**

Let  $s = \langle U^s, \iota^s \rangle \in 3Struct[\mathcal{P}]$  and  $s' = \langle U^{s'}, \iota^{s'} \rangle \in 3Struct[\mathcal{P}]$  be two three-valued logical structures and let  $f \in U^s \rightarrow U^{s'}$  be a function such that  $U^s \sqsubseteq_f U^{s'}$ . Then for every expression  $e \in \mathcal{FO}_{\mathcal{P}}$  and complete assignment  $Z$  for  $e$ , holds

$$([e] s Z) \sqsubseteq ([e] s' (f \circ Z)) .$$

*Proof.* see [SRW02] □

The  $\sqsubseteq$  is used in two different senses in the Embedding Theorem: as the embedding order on logical structures, thereby it bears a subscript  $f$ , and as the information order on the Kleene domain in the claim (the last line).

**Abstraction.** We fix a predicate signature  $\mathcal{P}$ , and a set of pairwise distinct unary predicate symbols  $A := \{a_1, \dots, a_k\} \subseteq \{p \in P \mid r(p) = 1\}$  termed **abstraction predicates**.

To guarantee that the analysis terminates, we make sure that the number of potential structures is finite. We make the following definition:

**Definition 5.1.10.** A bounded logical structure is a structure  $s = \langle U^s, \iota^s \rangle \in 3Struct[\mathcal{P}]$  such that for every pair of distinct individuals  $u_1, u_2 \in U^s$  there exists an abstraction predicate  $a \in A$  such  $\iota^s(a)(u_1) \neq \iota^s(a)(u_2)$ .

We denote the set of bounded structures as  $BStruct[\mathcal{P}]$ .

There is an upper bound on the size of the universe of a bounded structure, namely  $3^{|A|}$  (where  $|A|$  is the number of abstraction predicates). Therefore, there are only finitely many bounded structures. One way of obtaining a bounded structure is by canonical abstraction.

**Definition 5.1.11 (Canonical Abstraction).** Let  $s = \langle U^s, \iota^s \rangle \in 3Struct[\mathcal{P}]$  be a three-valued logical structure. The **canonical name** of an element  $u \in U^s$  with respect to  $A$ , denoted as  $\kappa_s^A(u)$ , is the vector

$$(\iota^s(a_1)(u), \dots, \iota^s(a_k)(u)) \in \mathbb{K}^k.$$

Let  $\widetilde{U}^s$  be the set of canonical names of all elements of  $U^s$ . Let  $\kappa_s^A \in U^s \rightarrow \widetilde{U}^s$  be the function which maps elements of  $U^s$  to their respective canonical name.

We define the **canonical abstraction** of  $s$  as  $\kappa_s^A(s)$  where  $\kappa_s^A(s)$  is a tight embedding of  $s$  induced by the embedding function  $\kappa_s^A$  (cf. Definition 5.1.9). We obtain the canonical abstraction function

$$\alpha_{can}^{\mathcal{P},A} : 3Struct[\mathcal{P}] \rightarrow 3Struct[\mathcal{P}], s \mapsto \kappa_s^A(s).$$

**Property preservation.** We introduce simulation preorder on transition systems with predicate logic structures. Simulation preorder can be seen as an approximation order on transition systems. It allows us to check properties of the concrete model in the abstract.

When  $K'$  simulates  $K$ , denoted by  $K \preceq K'$ , this means that  $K'$  approximates  $K$ .

If an ACTL\* property holds in  $K'$ , then it also holds in  $K$  (but not necessarily vice versa). We define simulation preorder.

**Definition 5.1.12 (Simulation).** Let  $\mathcal{P}, \mathcal{P}'$  be signatures such that  $\mathcal{P} \subseteq \mathcal{P}'$ . Let  $S \subseteq 3Struct[\mathcal{P}], S' \subseteq 3Struct[\mathcal{P}']$  and  $K = \langle S, I, R \rangle \in \mathcal{K}_S, K' = \langle S', I', R' \rangle \in \mathcal{K}_{S'}$  two transition systems.

A simulation relation  $H \subseteq S \times S'$  between  $K$  and  $K'$  is a relation such that for all  $(s, s') \in H$  we have

$$(i) \quad \forall t \in S : R(s, t) \Rightarrow \exists t' \in S' : H(t, t') \wedge R(s', t')$$

$$(ii) \quad \text{for all closed first-order expressions } e \in \mathcal{FO}_{\mathcal{P}} \text{ holds } s' \models e \Rightarrow s \models e.$$

We say that  $K'$  **simulates**  $K$  (denoted by  $K \preceq K'$ ) if there exists a simulation relation  $H$  between  $K$  and  $K'$  such that for every initial state  $s \in I$  there exists an initial state  $s' \in I'$  such that  $H(s, s')$ .

This is almost the standard definition. Condition (i) coincides with the classical definition. Condition (ii) does not. Classical simulation is a relation on Kripke structures, e.g., [CGP00]. Kripke structures are transition systems over a state space of propositional states. Propositional means that states are labeled with atomic propositions, as opposed to the states in this work which are logical structures. Such propositional states correspond to logical structures over a signature with 0-place predicates only, i.e. predicates that do not take an argument. Consequently, condition (ii) on states  $s, s'$  such that  $H(s, s')$  deviates from the classical condition  $L(s') \subseteq L(s)$  (where  $L$  is the labelling function for states). We could have used the embedding order on logical structures, i.e. we could have required  $s \sqsubseteq s'$  in (ii). However, by the Embedding Theorem that would be a stronger condition. We *will* use the embedding order  $\sqsubseteq$  to establish simulation relations. The current definition is sufficient, though, for our purpose:

**Theorem 5.1.2 (Simulation preserves ACTL\*).** *Let  $\phi$  be an ACTL\* state formula and  $K, K'$  two transition systems such that  $K \preceq K'$ . Then  $K' \models \phi \Rightarrow K \models \phi$ .*

*Proof.* The proof can be found in the Appendix in Section B.3. □

**Lemma 5.1.3.** *Simulation order  $\preceq$  is a preorder.*

*Proof.* Let  $K, K', K''$  be transition systems. We need to show that  $K \preceq K$ . Clearly, the identity relation is a witness for this simulation.  $K \preceq K' \preceq K''$  holds because the product  $\{(s, s'') \mid \exists s' : (s, s') \in H, (s', s'') \in H'\}$  of the simulation relations  $H$  and  $H'$ , which are witness of the simulation  $K \preceq K'$  and the simulation  $K' \preceq K''$ , respectively, is a witness of the simulation between  $K$  and  $K''$ . □

Being a preorder means reflexivity and transitivity. Antisymmetry does in general not hold, e.g., two isomorphic transition systems (in the sense of a graph isomorphism) simulate each other, however they are not necessarily identical.

## 5.2 Implementation

In this subsection, we give a meta-algorithm which, given a model computes a transition system over three-valued logical structures that simulates the semantics of the model. When the transition system has been computed, standard model checking procedures for ACTL\* can be applied to check the property, e.g., [CGP00].

Function *explore*, which is written out in pseudo-code in Figure 5.1, is an algorithm that computes a transition system  $K^\sharp = \langle S^\sharp, I^\sharp, R^\sharp \rangle$  where  $S \subseteq BStruct[\mathcal{P}] \subset 3Struct[\mathcal{P}]$ . Its interface to the framework of [SRW02] are the two functions *next* and *join*, which are passed as arguments. Furthermore, a set of initial states  $init \subseteq 3Struct[\mathcal{P}]$  is provided as an argument.

The function *next* computes the successors of an abstract state  $next \in 3Struct[\mathcal{P}] \rightarrow 2^{BStruct[\mathcal{P}]}$ . The function *join* is used to add a transition  $(s, s')$  to the transition system. The algorithm consists of a simple fixed point iteration. For this purpose, it maintains a set of processed states  $X$ , a workset  $W$  which is a set of states  $W \subset 3Struct[\mathcal{P}]$  whose successors are yet to be explored.

As expected, the arguments *init* and *next* of the algorithm depend on the model being analyzed. So let us fix such a constant-domain model  $M$  and a choice of abstraction predicates  $A$ . The state space  $S'$  of  $M = \langle S', \theta, \rho \rangle$  is given implicitly by a set of compatibility constraints  $\mathfrak{R}$  which are evaluated over  $Struct[\mathcal{P}, U]$ , so  $S' = \{s \in Struct[\mathcal{P}, U] \mid s \models \mathfrak{R}\}$ . Typically,  $A$  will contain Skolem predicates. Let  $K = \langle S', I, R \rangle = \llbracket \overline{M} \rrbracket$  be the transition system induced by model  $\overline{M}$ , the skolemized version of  $M$ . We abbreviate  $\alpha = \alpha_{can}^{\mathcal{P}, A}$ .

**Constraints.** Three-valued structures that arise during an analysis sometimes represent structures which correspond to impossible states of the concrete system. In particular, impossible structures arise after the application of the focus operation. In order to improve the accuracy of the analysis, TVLA provides mechanisms for constraint processing and generation. Thereby constraints may be used to make a structure more precise, by turning a 1/2 to a definite value. This can happen if the value of one predicate can be derived

```

1  transition system
2  explore (init: set of states,
3          next: state * set of states,
4          join: transition system * state * state ) {
5      transition system K = <init,init,emptyset>;
6      set of states W = init,
7          X = emptyset;
8      while(size(W)>0) {
9          forall(s in W) {
10             forall(s' in next(s)) {
11                 K := join(K,s,s');
12             }
13             W := W - {s};
14             X := X + {s};
15             W := W + (next(s) - X)
16         }
17     }
18     return K;
19 }

```

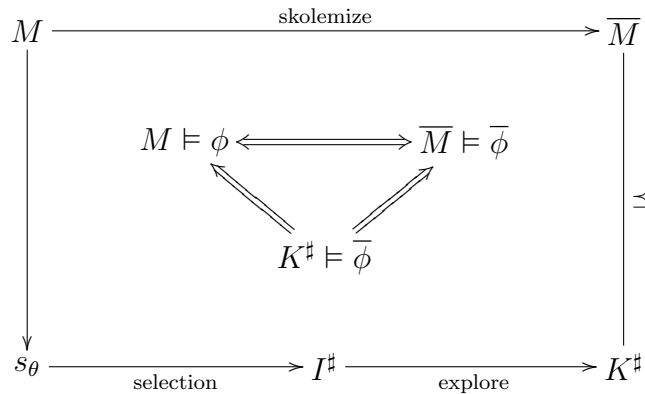
Figure 5.1: Algorithm *explore*. It is a parameterized algorithm which constructs an abstract transition system.

from another predicate. If a structure breaches a constraint, it is ruled out. Some constraints are generated automatically but one can also provide constraints by hand. These constraints often express domain-specific knowledge, e.g., a variable has exactly one value and hence the predicate symbol which models the variable cannot evaluate to 1 on two distinct individuals. Constraints can also be used to filter out undesirable results of the focus operation when one implements nondeterminism. We implemented Skolemization in TVLA this way. The Skolem predicates are set to 1/2 and the focus operation enumerates possible choices, but it also generates impossible cases where a Skolem predicate refers to a summary individual, to no individual at all, and so forth.

**Definition 5.2.1 (Compatibility Constraints).** *A compatibility constraint  $\tau$  has the form  $e_1 \triangleright e_2$  where  $e_1 \in \mathcal{FO}_{\mathcal{P}}$  is a first-order expression, and  $e_2$  is a term (an element of  $\mathcal{T}_{\mathcal{P}}$ ) or the negation of a term. We say that a three-valued structure  $s$  and an assignment  $Z$  satisfy  $e_1 \triangleright e_2$ , denoted  $s, Z \models e_1 \triangleright e_2$ , if  $([e_1] s Z) = 1 \Rightarrow ([e_2] s Z) = 1$ . We say that  $s$  satisfies  $e_1 \triangleright e_2$  if for all assignment  $Z$  holds  $s, Z \models e_1 \triangleright e_2$ . If  $\mathfrak{R}$  is a finite set of constraints, we say that  $s$  satisfies  $\mathfrak{R}$ , if  $s$  satisfies all constraints  $\tau$  in  $\mathfrak{R}$ .*

**Computing initial states: *init*.** The method corresponds to nondeterministically selecting individuals and instantiating the quantified property with them. The framework [SRW02] assumes that the initial states are given explicitly rather than syntactically. We can use [YRS03] to compute the most precise (with respect to the embedding order) bounded logical structure  $s_\theta$  that fulfills  $\theta$ . We assume that we want to quantify over individuals marked with predicate symbols  $p_1, \dots, p_n$ , i.e.  $\forall x : p_1 \dots \forall x : p_n : \phi$ . There are Skolem predicates  $\eta_1, \dots, \eta_n$ . In  $s_\theta$ , we set the valuation of  $\eta_i$  to 1/2 on the individuals marked by  $p_i$  for each  $i$  and obtain  $s'_\theta$ . This structure conservatively overapproximates  $I$ , i.e.  $I \subseteq \gamma_3(s'_\theta)$ . We use the partial concretization operation of [SRW02] called *focus* with arguments  $\{\eta_1, \dots, \eta_n\}$  and apply it to  $s'_\theta$ . Thus we obtain a set of initial states  $I^\sharp$  from  $s'_\theta$ . This makes sure that the union over the concretizations of the structures in  $I^\sharp$  contains  $I$ , i.e.  $I \subseteq \bigcup \{\gamma_3(s) \mid s \in I^\sharp\}$ .

The diagram below depicts the idea behind our implementation the step from the initial state  $s_\theta$  to the set  $I^\sharp$  consists in selecting values.



**Computing successors: *next*.** Currently, there are two ways in which one can compute a successor function, one can either use the method from [SRW02] or the decision procedure of [YRS03]. [SRW02] provides a method to compute a successor function only

for a  $\rho$  as described in the paragraph about actions in Section 2.5. This is not the case for the decision procedure [YRS03].

Both methods have in common that one provides them with  $\rho$  and an abstract state  $s \in 3Struct[\mathcal{P}]$ . The successors of  $s$  are computed using the syntactic description of the transition relation  $\rho$  of the concrete model. Compatibility constraints are used to rule out impossible structures and refine structures which satisfy the constraints. It should be noted that *next* returns sets of *bounded* logical structures. This is important for the termination of our algorithm.

More details concerning the algorithm follow in the next section.

**Join methods: *join*.** The function *join* can be chosen in several ways. The relational join operation is defined as  $join_1(\langle S, I, R \rangle, s, s') = \langle S \cup \{s'\}, I, R \cup \{s, s'\} \rangle$  and partial embedding

$$join_2(\langle S, I, R \rangle, s, s') = \begin{cases} \langle S, I, R \cup \{s, s''\} \rangle & ; \exists s'' \in S : s' \sqsubseteq s'' \\ join_1(\langle S, I, R \rangle, s, s') & ; \text{otherwise} \end{cases}$$

Using partial embedding as a join method leads to a potentially less precise transition system. In practice, particularly when safety properties are being checked, partial embedding can lead to a tremendously smaller state space.

**Correctness.** We need to show partial correctness and termination. Partial correctness means that the transition system computed by state space exploration simulates the concrete transition system, i.e.  $K \preceq K^\#$ . The Local Safety Theorem (Theorem 6.29 of [SRW02]) and the Embedding Theorem show directly that  $\sqsubseteq \cap S \times S^\#$  is a simulation relation between  $K$  and  $K^\#$ <sup>2</sup>. The fact that  $I \subseteq \bigcup \{\gamma_3(s) \mid s \in I^\#\}$  makes sure that  $K \preceq K^\#$ .

The algorithm *explore* terminates if and only if  $W$  will eventually be empty. We give a brief plausibility argument based on the boundedness of the state space (consisting of bounded structures).

$X$  and  $W$  are disjoint.  $X$  contains exactly all values the iterator of the outer *forall* loop (lines 9-16) has ever taken on. For every structure  $s'$ , the iterator  $s$  of the outer *forall* loop is equal to  $s'$  at most once during a run of *explore*. Let us assume there be a run in which a structure  $s'$  appears twice. After the first time  $s'$  appeared it must be in  $X$  (and remains in  $X$ ). When  $s'$  is picked a second time it must be in  $W$ . At this moment,  $X$  and  $W$  would not be disjoint. Contradiction. Now we know that the size of  $X$  is increasing strictly monotonously with each iteration of the out-most forall loop. However,  $X \subseteq BStruct[\mathcal{P}]$  and hence  $X$  stabilizes eventually ( $|BStruct[\mathcal{P}]| < \infty$ ). We need to show that  $W$  will eventually be empty. If this were not so, there would be infinitely many iterations of the outer *forall* loop. Contradiction. Therefore the exploration terminates.

---

<sup>2</sup>If the relational join method is used. However, when the partial embedding join method is used the resulting transition system simulates the transition system obtained by using relational join. And by transitivity of simulation preorder, we obtain correctness for this case as well.

**Optimality Considerations.** The transition system computed by *explore* may not be the most precise transition system. In this paragraph, we discuss aspects of optimality. It would seem a consideration with little value if one could only mathematically describe optimal abstraction transition systems without being able to compute them. However, the use of decision procedures, e.g., satisfiability checks, allows one to approximate optimal abstract transition systems. This differs substantially from the methods of [SRW02], where the model is literally *executed* on logical structures.

An ideal decision procedure would produce the optimal transition system. The more computational effort is invested into the computation the closer one comes to the optimal transition system. Such a form of approximation has been reported for predicate abstraction, e.g., [GHJ01, HJS01]. The work on optimal transformers for shape analysis [YRS03] is an effort with a similar goal ([YRS03] does not explicitly consider transition systems though). Because of the potential practical relevance of optimal abstract transition systems, we relate the notion of optimality given through the best transformer and the notion of optimality from [GHJ01, HJS01].

The crucial component is the computation of successors, i.e. the function *next*. We define a post operator:  $post_R(S') = \{t \in S \mid \exists s \in S' : R(s, t)\}$ . Using the post operator we can define *the best transformer*  $\alpha \circ post_R \circ \gamma_3$  and the transition relation  $R_{bt} = \{(\tilde{s}, \tilde{t}) \mid \tilde{t} \in \alpha \circ post_R \circ \gamma_3(\tilde{s})\}$ . One can extend a function  $h \in M \rightarrow M$  on a set  $M$  to the power set  $2^M$  by consider the function  $\hat{h} \in 2^M \rightarrow 2^M$  where  $\hat{h}(S) = \{h(m) \mid m \in S\}$ . This allows us to define  $S_{bt}$  as the least fixpoint of

$$F = \lambda S \subset BStruct[\mathcal{P}]. \bigcup_{s \in S} \hat{\alpha}(post_R(\gamma_3(s)))$$

(which is a monotone function on a finite domain, therefore the fixpoint exists). By choosing  $I_{bt} = \alpha(I)$  we obtain an abstract transition system  $K_{bt} = \langle S_{bt}, I_{bt}, R_{bt} \rangle$ . The transition system corresponds to choosing  $next(s) = \alpha(post_R(\gamma_3(s)))$  and the relational join operation in the exploration algorithm *explore*. The sequence of simulations  $K \preceq K_{bt} \preceq K^\sharp$  holds.

The following lemma describes another way how one can systematically construct an abstract transition system for which property preservation holds. The idea is taken from [GHJ01, HJS01]. We will see that this concept provides an even more precise abstract system.

**Lemma 5.2.1.** *Let  $K = \langle S, I, R \rangle$  be a transition system,  $\mathcal{P}' \supseteq \mathcal{P}$  another signature,  $\tilde{S} \subseteq 3Struct[\mathcal{P}']$  a state-space, and  $H_a \in S \times \tilde{S}$  such that for all pairs  $(s, \tilde{s}) \in H_a$  and closed first-order expressions  $e \in \mathcal{FO}_{\mathcal{P}}$  holds  $\tilde{s} \models e \Rightarrow s \models e$ .*

*Then the transition system  $\tilde{K} = \langle \tilde{S}, \tilde{R}, \tilde{I} \rangle \in \mathcal{K}_{\tilde{S}}$  defined as*

$$\begin{aligned} \tilde{R}(\tilde{r}, \tilde{s}) &\Leftrightarrow \exists r \in S \exists s \in S : R(r, s) \wedge H_a(r, \tilde{r}) \wedge H_a(s, \tilde{s}) \\ \tilde{I}(\tilde{r}) &\Leftrightarrow \exists r \in S : H_a(r, \tilde{r}) \end{aligned}$$

*simulates  $K$  and  $H_a$  is a simulation between  $K$  and  $\tilde{K}$ .*

Relation  $H_a = \{(s, \tilde{s}) \mid \alpha(s) = \tilde{s}\}$  yields an abstract transition system  $\tilde{K}$  ( $s \sqsubseteq \alpha(s)$  for every  $s \in 3Struct[\mathcal{P}']$  which implies by the embedding theorem that for closed first-order expressions  $e \in \mathcal{FO}_{\mathcal{P}}$  holds  $\tilde{s} \models e \Rightarrow s \models e$ ).

The relation  $\sqsubseteq \cap (\alpha(S) \times S_{bt})$  is a simulation relation between  $K_{bt}$  and  $\tilde{K}$ .

*Proof.* Clearly, because of the Embedding Theorem, we have for each closed first-order expression  $e \in \mathcal{FO}_{\mathcal{P}}$  that  $\tilde{s} \models e \Rightarrow s \models e$ . Let  $(\alpha(s), \alpha(s')) \in \sqsubseteq \cap (\alpha(S) \times S_{bt})$ . Assume that there exists a  $t \in S$  such that  $R(s, t)$ . We have  $\gamma_3(\alpha(s)) \subseteq \gamma_3(\alpha(s'))$  and hence  $t \in \text{post}_R(\gamma_3(\alpha(s))) \subseteq \text{post}_R(\gamma_3(\alpha(s')))$ . Hence  $\alpha(t) \in \alpha(\text{post}_R(\gamma_3(\alpha(s)))) \subseteq \alpha(\text{post}_R(\gamma_3(\alpha(s')))) \subseteq S_{bt}$ .  $\square$

$K_{bt}$  is in general not simulated by  $\tilde{K}$ . One reason is that the concretization of a 3-valued structure may yield structures that are not in  $S$ , the state space of the concrete system. We extend the sequence of simulations  $K \preceq \tilde{K} \preceq K_{bt} \preceq K^\sharp$ .

### 5.3 Case Study

We conducted the case study using TVLA. TVLA implements concepts of the static analysis framework [SRW02]. Models are described in predicate logic. The syntactic form of those models was discussed in Paragraph 2.5. The initial states of the abstract system are given explicitly instead of using a constraint. There are several modules in TVLA, there is a module for intraprocedural program analysis which computes the set of reachable structures for each control flow node of a program (which is used in [SRW02]), but there is also a module, called *TVMC* (for Three-Valued Model Checker) which performs a pure reachability analysis. Module *TVMC* applies the actions described in Paragraph 2.5 until a fixpoint is reached, transitions are not being recorded during exploration.

**The focus operation.** Simply evaluating the concrete model on three-valued structures produces too much imprecision (cf. [SRW02], *Strawman Semantics*). In order to increase precision, a partial concretization function, called *focus* is used. The focus operation is parameterized in a set of formulas. It is applied to three-valued logical structures and produces a set of three-valued structures. The operation produces a set of structures in which the input formulas evaluate to a definite value (0 or 1). Each action (see Section 2.5) has its own set of focus formulas. Finding appropriate formulas is nontrivial and may require experience. However, for existing analyses involving linked data structures there exist suitable formulas (we have made use of this existing knowledge when modeling the ticket domain of the Ticket Protocol). The focus algorithm enumerates structures on which the formula produces definite values and which cover the concretization of the input structure. This is done by splitting up summary individuals, i.e. abstract individuals which represent multiple concrete individuals. A simple example is a three-valued structure with a single node, a summary node on which a unary predicate  $p$  evaluates to  $1/2$ . By applying the focus operator, parameterized with predicate  $p$ , we obtain several structures. Structures with two individuals  $\{u, v\}$  where  $\iota(p)(u) = 1$ ,  $\iota(p)(v) = 0$  (there are three possibilities because of the *sm* predicate). Structures with a single individual on which  $p$  either evaluates to 0 or 1 and which is a summary individual or not. The focus operation can be used to implement nondeterminism, as it produces multiple states. The nondeterministic choice of the currently active process in *TVMC* is done with the focus operation.



**Core and Instrumentation Predicates.** *Core predicates* are used to model the semantic effect of actions of a program. In case of the Ticket Protocol, we model the global and local variables by core predicates, and we describe the effect of incrementing a variable by predicate update formulas. Sometimes the core predicates are not sufficient to prove a property. Then it is necessary to provide auxiliary predicates either manually or by automatic abstraction refinement. These predicates cache the valuation of a certain formula. Technically, the valuation of these so-called *instrumentation predicates* is described by this formula in terms of other predicates. A conservative valuation of instrumentation predicates can thus be obtained by re-evaluating this defining formula. However, maintaining the value and modeling the effect of operations by update formulas leads to drastic gains in precision compared to re-evaluation in each state. An example for an instrumentation predicate is the transitive closure of the successor function on natural numbers. The defining formula of this instrumentation predicate  $t[succ]$  is  $t[succ](m, n) = TC(v_1, v_2 : succ(v_1, v_2))(m, n)$ . Given the successor predicate  $succ$ , we can evaluate the defining formula of  $t[succ]$  but that would be less precise than caching the value (cf. Figure 5.4 where  $t[succ]$  has a definite valuation but  $succ$  has an indefinite valuation). Finding the right instrumentation predicates can be crucial for an analysis, but it often requires knowledge about the program and about the analysis engine. [RSL03] describes the maintenance of instrumentation predicates, i.e. how update formulas can be computed from the defining formula and the predicate update formulas of the other predicates.

**Computing next.** In the framework [SRW02], the successor function is a composition of the canonical abstraction function  $\alpha$ , the focus operation  $focus$ , the application of the update formulas  $upd$ , and constraint checks  $constr$ . We used the following sequence of operations  $focus - upd - coerce - \alpha$ . First, the focus operation is applied. This operation materializes, e.g., the process which is to execute an action and possibly some data fields of it. That yields a collection of three-valued structures. The precondition of the action is checked. Structures which do not fulfill the precondition are ignored. The update formulas are applied to the remaining structures. The update formulas implement the semantic effect of an action. Then the constraint checks, denoted by  $coerce$ <sup>3</sup>, are used to re-gain precision or rule out impossible structures. Finally, canonical abstraction is applied to ensure termination. It is also possible to perform a constraint check after focusing, but since constraint checks are expensive we omitted this step. Our experience was that this second constraint check did not lead to better results in our examples.

[YRS03] promises to be an approach with a higher degree of automation where  $\theta$  and  $\rho$  suffice to compute the transition system. Whether that is a practical solution remains to be verified.

**Running Example.** The Ticket Protocol (cf. Figure 1.1) is a solution to the problem of mutual exclusion, i.e. it is safe and live. We were only able to prove safety using TVLA, since, at the time of this writing, TVLA did not produce transition systems. This restrained us from showing  $\phi_{LIVE}$ . As mentioned in [YR04], quantifier elimination can

---

<sup>3</sup> meaning and etymology according to [FB95], *coerce*: (formal) *constrain by superior force*, *coercere* (Latin): restrain, ward off

predicate	intended meaning
$\{x(l) \mid x \in \{process, number\}\}$	type of node $l$
$succ(i, i')$	number $i'$ is the successor of number $i$
$\{at[loc](t) \mid loc \in \{think, wait, crit\}\}$	thread $t$ is at location $loc$
$a(p, i)$	$a$ -field of process $p$ has value $i$
$s(i)$	variable $s$ has value $i$
$t(i)$	variable $t$ has value $i$

Figure 5.2: Ticket Protocol: intended meaning of the core predicates.

predicate	intended meaning
$t[succ]$	reflexive, transitive closure of $succ$
$r[s, succ](i)$	number $i$ is transitively reachable from variable $s$
$r[t, succ](i)$	number $i$ is transitively reachable from variable $t$

Figure 5.3: Ticket Protocol: intended meaning of the instrumentation predicates. In words  $r[s, succ](i)$  means that  $s$  has a value that is less or equal  $i$ .

be a useful means to produce more compact and possibly more precise abstractions. We observed such effects while verifying the Ticket Protocol.

We have already formulated safety and liveness of the Ticket Protocol as QACTL\* formulas in Figure 3.2. However, there we referred to a model in general first-order logic and not in predicate logic. Before we come to re-formulating properties, we need to talk about how we model the Ticket Protocol in predicate logic. We have seen some predicates already in Example 4.2.1. We go through the predicates and repeat some things that have been mentioned before. We give individuals a type using unary predicates *process* and *number*. We model the global variables  $s$  and  $t$  as unary predicates  $s, t$  and the  $a$ -field as a binary predicate  $a$ . The core and instrumentation predicates we used are listed in Figure 5.2 and Figure 5.3, respectively.

Safety, i.e. mutual exclusion, is expressed by the property:

$$\forall p_1. \forall p_2. \mathbf{AG}(at[*crit*](p_1) \wedge at[*crit*](p_2) \Rightarrow p_1 = p_2)$$

The three-valued structures at the start of the analysis are of particular interest since they illustrate the Skolemization step. The implementation in TVLA is that we nondeterministically pick the value of the Skolem predicate. TVLA output showing a three-valued structure as it occurred during the verification of property *MUTEX* is depicted in Figure 5.4. The figure shows how Skolemization was implemented in TVLA by the focus operation.

*MUTEX* is a safety property and TVLA comes with means to check safety properties directly without Skolemization. However without using additional instrumentation predicates or Skolemization we were not successful. Without Skolemization, we had to use the following instrumentation predicate to verify *MUTEX*:

$$\begin{aligned} ins(n) = & \text{number}(n) \wedge \forall p_1. \forall p_2. process(p_1) \wedge process(p_2) \\ & \wedge a(p_1, n) \wedge a(p_2, n) \wedge \neg at[think](p_1) \wedge \neg at[think](p_2) \Rightarrow p_1 = p_2 \end{aligned}$$

We successfully verified *MUTEX* using Skolemization and heterogeneous abstraction. More aggressive heterogeneous abstraction which loses all information about summary individuals, as in [DW03], leads to spurious changes of counter variable  $s$ , and thus to a spurious counterexample. We will discuss this in more detail in Chapter 7.

**Update Formulas.** Finding update formulas for the core predicates is quite straightforward in this example. The command  $\langle a := t; t := t + 1 \rangle$  can be translated as:

$$\begin{aligned} a(p, i) &= p = tr \wedge t(i) \vee \neg(p = tr) \wedge a(p, i) & (a:=t) \\ t(i_2) &= \exists i_1. succ(i_1, i_2) \wedge t(i_1) & (t:=t+1) \end{aligned}$$

where  $tr$  denotes the process that is currently active. The condition  $a = s$  translates to  $\exists i : a(tr, i) \wedge s(i)$ . The command  $s := s + 1$  analogously can be modeled as:

$$s(i_2) = \exists i_1 : succ(i_1, i_2) \wedge s(i_1) .$$

The instrumentation predicates (except those used to implement heterogeneous abstraction) are listed in Figure 5.3. The TVLA files are made available together with this document.

**Heterogeneous Abstraction.** We want to distinguish the tickets of the processes referenced by Skolem predicates from other tickets. The Skolem predicates which refer to processes are in our case  $p_1, p_2$ . The following instrumentation predicates  $a_{p_1}, a_{p_2}$  are used to implement that:

$$\begin{aligned} a_{p_1}(i) &= \exists p. p_1(p) \wedge a(p, i) \\ a_{p_2}(i) &= \exists p. p_2(p) \wedge a(p, i) . \end{aligned}$$

The right-hand sides of the equations above are the defining formulas of the instrumentation predicates. This kind of instrumentation is systematic in that we want to distinguish all individuals in connection with the individuals referenced by Skolem predicates. However, it is only an implementation of heterogeneous abstraction. A more generic implementation would be more efficient because then less constraint checks and predicates would be necessary.

## 5.4 Related 3-valued analyses

Aspects of temporal verification in the context of three-valued analysis have been discussed in earlier work, e.g. [YRSW03, YR04, YRS01]. We are concerned with three-valued program analysis and first-order (temporal) verification which goes beyond the invariant checking of [SRW02]. We do not discuss the work concerning LTL model checking using TVLA; it does not allow verifying *quantified temporal properties* (just temporal properties). We have claimed that our method is an improvement compared to *previous* 3-valued analysis based on the framework [SRW02] because our method allows checking temporal properties using decomposition. Our method is related to [YR04]; a difference is that they consider safety properties and we consider general temporal properties.

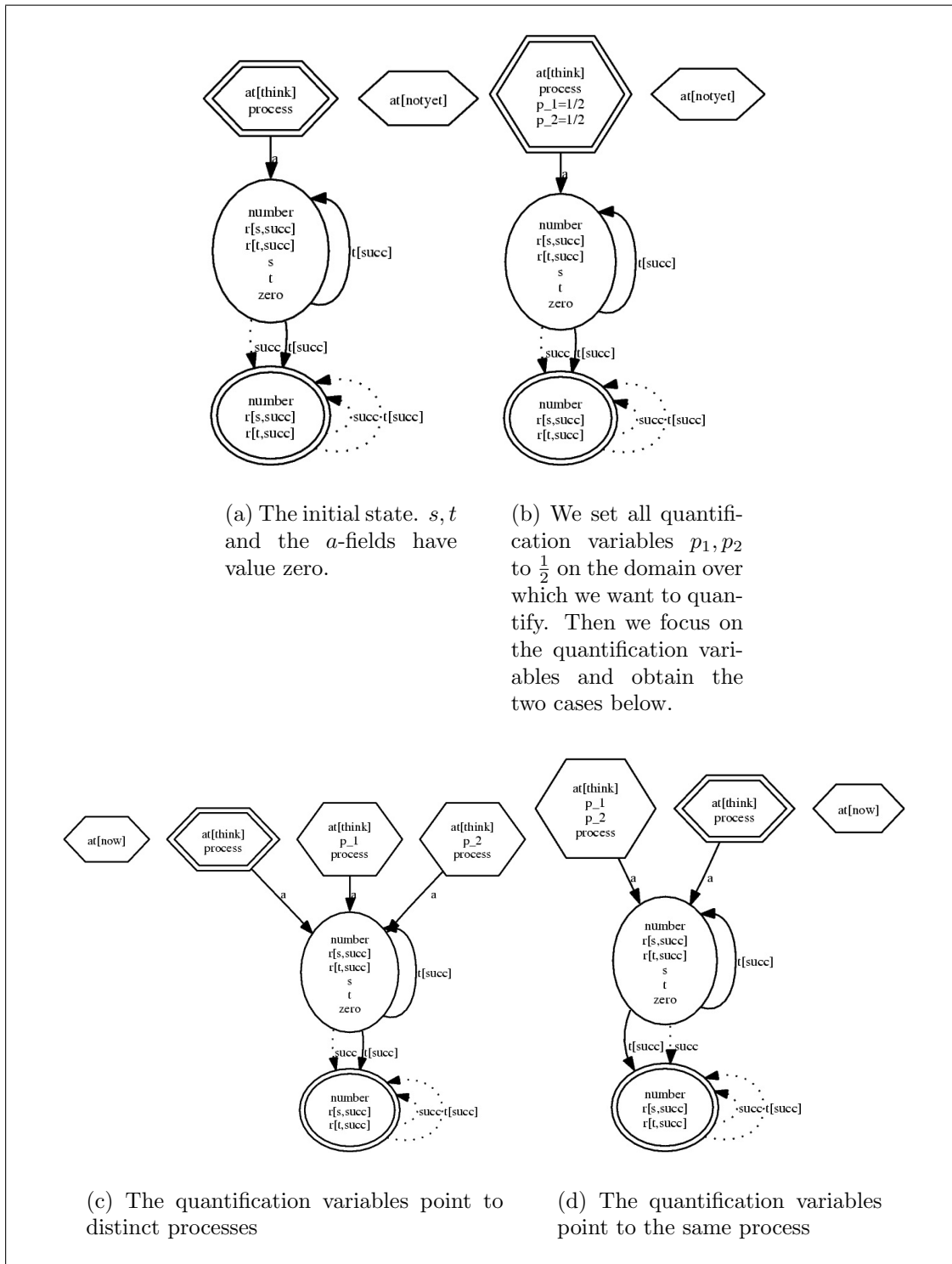


Figure 5.4: **MUTEX**. The nodes labeled with  $at[now]$ ,  $at[notyet]$  are an artifact of the way we implemented Skolemization in TVLA. Namely, the nodes represent the TVMC thread that produces the cases by focusing on the quantification predicates  $p_1, p_2$ . After the Skolemization we need to show  $\mathbf{AG}(\exists u. \exists v. \eta_1(u) \wedge \eta_2(v) \wedge at[crit](u) \wedge at[crit](v) \Rightarrow p_1 = p_2)$  for each of the two cases 5.4(c) and 5.4(d). Which yields two verification problems. (Note that in the picture  $\eta_1$  and  $\eta_2$  are called  $p_1$  and  $p_2$ .)

**Heterogeneous Abstraction and Decomposition Strategies.** [YR04] is about decomposition strategies for the verification of first-order *safety properties* and heterogeneous abstraction. The analysis is aimed verifying so-called *first-order safety properties* which resemble quantified safety properties. The ideas are illustrated with a Java example (JDBC library and Easl specifications). We have adopted the term *heterogeneous abstraction* from this article. The treatment of heterogeneous abstraction is orthogonal to our work. Decomposition is done by instantiation in the abstract. Similar syntactic transformations for quantifier instantiation are employed as we use. Computing the abstract transition system of a Skolemized model, can also be implemented by decomposition and yields subgoals (in our method, the subgoals appear in one first-order transition system  $K^\sharp$ ). This can be seen in Figures 5.4.

Summarizing the relation between our work and [YR04] one can say that:

- Skolemization is a general-purpose method which does not rely on a particular abstract representation of states (or the heap).
- We use a form of selection (within one transition system) to approximate the concrete model  $\bar{M}$  created by Skolemization. Our *implementation* is related to the method sketched in [YR04].
- First-order safety properties expressible by safety properties  $\mathbf{AG} e$  ( $e \in \mathcal{FO}_P$ ) with universal quantifiers in front. [YR04] is a verification method for first-order safety properties. General quantified temporal properties are not considered.
- The semantics of models is a varying-domain semantics, i.e. allocation is modeled as adding an individual. We implement allocation by making a dead individual live, i.e. by changing a unary predicate.

Our work focuses more on the formal side, i.e. semantics and models, while [YR04] is concerned with practical aspects of decomposition. Our current practical experience does not go beyond quantified safety properties because TVLA did not offer transitions systems at the time of this writing.

**3-valued ETL verification.** [YRSW03] gives a language, called ETL, in which one can specify *heap-evolution properties*. ETL is given a varying-domain trace semantics (we have discussed aspects of this method in Chapter 2). We want to focus on the analysis which was proposed for ETL verification. The analysis has conceptual shortcomings in terms of precision. The fact that *quantification is a source of imprecision* is inevitable. We illustrate this in Figure 5.5. The imprecision is inevitable because, at many points, indefiniteness has to hold otherwise the analysis would not be conservative. In order to understand this, it is necessary to descend into the details of the implementation of the analysis. Another less severe issue will be discussed afterwards.

The ETL checking problem is reduced to valuating an expression of first-order logic with transitive closure in a three-valued structure. Traces are encoded in a single logical structure. States (there they are called *worlds*) are individuals of the logical structure and transitions are modeled as a successor predicate *succ* (not to be confused with the successor on natural numbers we used in our example).

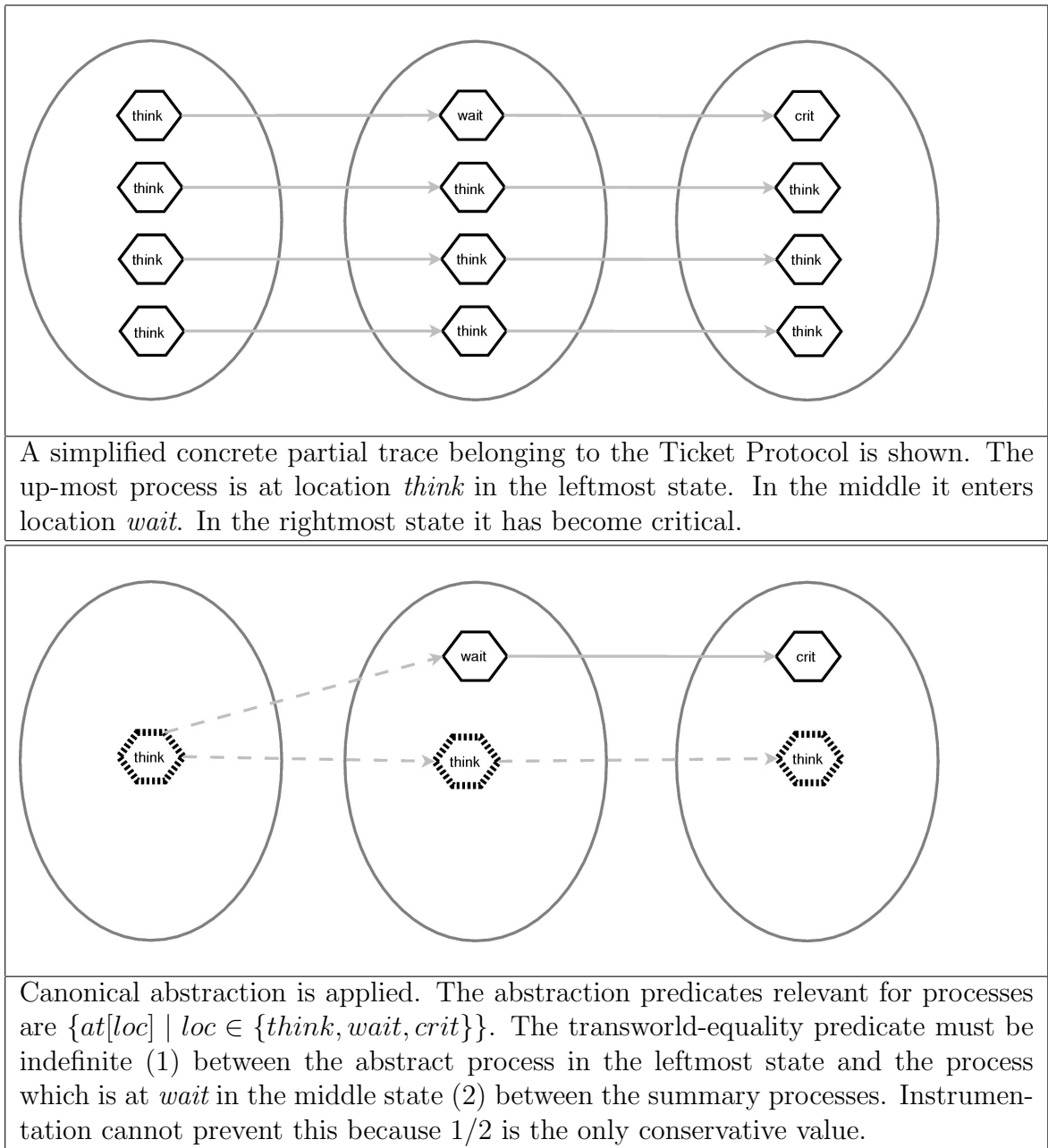


Figure 5.5: One inherent imprecision of [YRSW03] is illustrated above. The oval shapes stand for states (worlds). The gray arrows depict transworld equality; solid lines denote definite information, dashed lines represent indefiniteness. The hexagons stand for processes. Transworld equality becomes indefinite very quickly: (1) between summary individuals it must be indefinite to be conservative (cf. dashed lines in the lower picture) (2) as soon as one process *leaves* a cluster of abstract processes (in our example this is when interesting things happen). As quantification is implemented via transitive closure over transworld equality indefinite transworld equality propagates and produces indefinite answers. *Instrumentation cannot remedy this weakness*. The imprecision springs from the need of transworld equality to be *conservative*. It is not specific to this particular example, it is inherent.

The individuals within states are also individuals in the encoding. A predicate expresses that an individual belongs to a world. In order to relate individuals in different worlds, a binary evolution predicate, called transworld-equality, is used. Every two states which are conceptually equal evaluate to 1 under the predicate. Existential and universal quantification are implemented using this predicate. The problem is that the predicate becomes indefinite quickly. Such a situation is described in Figure 5.5. The pictures illustrate the situation nicely. Whenever individuals leave a summary individual, transworld equality must become indefinite between the original summary individual and those individuals who have *left it*. This is necessary to be conservative, since not all individuals represented by the summary node leave the partition. The authors propose to use instrumentation, as previously discussed in this Chapter, to make the analysis more precise. Among other things, they propose to use instrumentation to record the transitive closure of transworld-equality. However, even if instrumentation is used, the analysis still has to be conservative. Transworld-equality has to be indefinite here in order to be conservative. This problem leads to the effect that quantification produces indefiniteness. If we have universal quantification, transworld equality (or more precisely its transitive closure) will produce the value 1/2 when a cluster of individuals is split up.

Our method does not have such a problem. Quantifier instantiation in combination with the abstraction which distinguishes individuals *selected* by the instantiation *cannot* contribute indefiniteness.

Transitions produce indefiniteness. However, this is a problem which could be fixed. We explain what the problem is. In abstract traces, the predicate which encodes transitions may have an indefinite valuation. *succ* and its transitive closure are used to model temporal operators in the encoding. If *succ* has an indefinite value, indefiniteness propagates to temporal properties. It should be noted that the analysis in [YRSW03] is biased, i.e. 0 does not necessarily mean that the concrete model does not fulfill the specification. Other approaches, e.g., [CGL94, GHJ01, HJS01, SS99], which use modality on transitions are not biased. It seems as if the use of the *succ* predicates does not bring a benefit but rather produces unnecessary imprecision. Furthermore, the conservative interpretation of predicates by canonical abstraction differs from the conservative interpretation of abstract modal transitions as described in [GHJ01, HJS01]. This is best illustrated with an example. Transitions on states are represented as a binary predicate *succ*. There are two pairs  $s1, s2$  and  $s3, s4$  of concrete states  $\{s1, s2, s3, s4\}$ . There are transitions  $succ(s1, s3), succ(s1, s4)$ . Now, let us assume that each pair falls together to an abstract state  $u1 = \{s1, s2\}, u2 = \{s3, s4\}$ . Canonical abstraction yields  $succ(u1, u2) = 1/2$ . However, in the sense of [CGL94, GHJ01, HJS01]  $succ(u1, u2) = 1$ , i.e. in the lingo of modal transitions: it is a must-transition. This situation is illustrated in 5.6. For the existence of a must transition from  $u1$  to  $u2$  it is sufficient that all concrete states in  $u1$ , namely  $s1, s2$ , can move to *any* of the concrete states in  $u2$ , here either  $s3$  or  $s4$ . Canonical abstraction requires that all states in  $u1$  can move to *all* states in  $u2$  ( $succ(u1, u2)$  requires  $succ(s1, s3), succ(s1, s4), succ(s2, s3), succ(s2, s4)$ ).

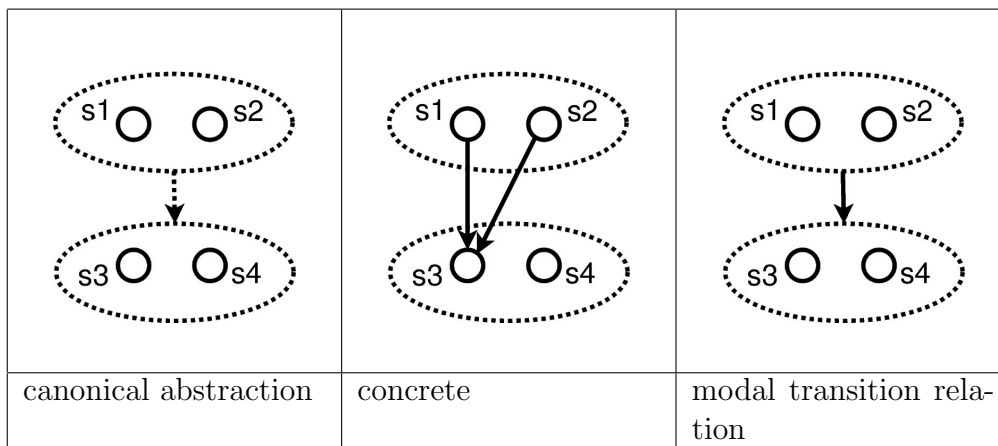


Figure 5.6: Canonical abstraction and modal transitions.

## 5.5 Discussion

**Preservation of QCTL\*.** We consider abstract transition systems which *overapproximate* the behavior of the concrete system. Therefore we can show QACTL\* formulas and not full QCTL\*. By means of modal transitions systems [GHJ01, HJS01, SS99], it is also possible to preserve CTL\* (using Skolemization we could show QCTL\* properties of models). Thereby an over- and an underapproximation of the concrete transition relation is computed. The underapproximation can often be computed as the dual of the overapproximation [SS99]. We will not go into detail here. It is not difficult to mathematically define best over- and underapproximating abstract transition systems. Algorithms for computing such underapproximating relations for canonical abstraction were not known to us at the time of this writing. This might be more difficult than in case of predicate abstraction [SS99] since the computation of duals is more complicated<sup>4</sup> (complementation of a set of 3-valued structures cannot be implemented as efficiently). Addressing this problem goes beyond the scope of this thesis.

**Strong and Weak Preservation.** Simulation only guarantees *weak preservation*, i.e. only validity can be shown on the abstract model, not invalidity. Strong preservation for first-order expressions between a logical structure and the structures it represents is guaranteed by the Embedding Theorem. In [YRSW03, YRS01], there are theorems ([YRSW03]: Theorem 2, ETL, [YRS01]: Theorem 52, LTL) which claim that simulation guarantees strong preservation. That, however, cannot be, since a simulation  $K \preceq K'$  only guarantees that transition system  $K'$  overapproximates the behavior of  $K$ . Consequently, there can be paths in the overapproximating transition system for which there is no counterpart (cf. Definition B.3.1) in  $K$ . If there is a path starting with an initial state in  $K'$  which does not fulfill a path formula  $\Phi$ , it can happen that  $K \models \mathbf{A}\Phi$  holds nonetheless. At other points in [YRSW03, YRS01], it is said that for the analysis only weak preservation holds. This ambiguity is mentioned because we only show weak preservation in Theorem 5.1.2. This is *not* a deficiency of our work compared to [YRSW03, YRS01]. The methods

<sup>4</sup>We conjecture that this may be done using decision procedures and without using complementation.



discussed in the previous paragraph would make it possible to obtain strong preservation.

# Chapter 6

## Symmetry

This chapter has several purposes. It conveys an intuition of symmetry. We explain why, in case of our predicate logic, symmetry coincides with isomorphism. Thereby symbols whose valuation is determined by the logic play an important role. Last but not least, we motivate, formalize and prove the notion that canonical abstraction collapses symmetries (Theorem 6.2.2).

### 6.1 Intuition.

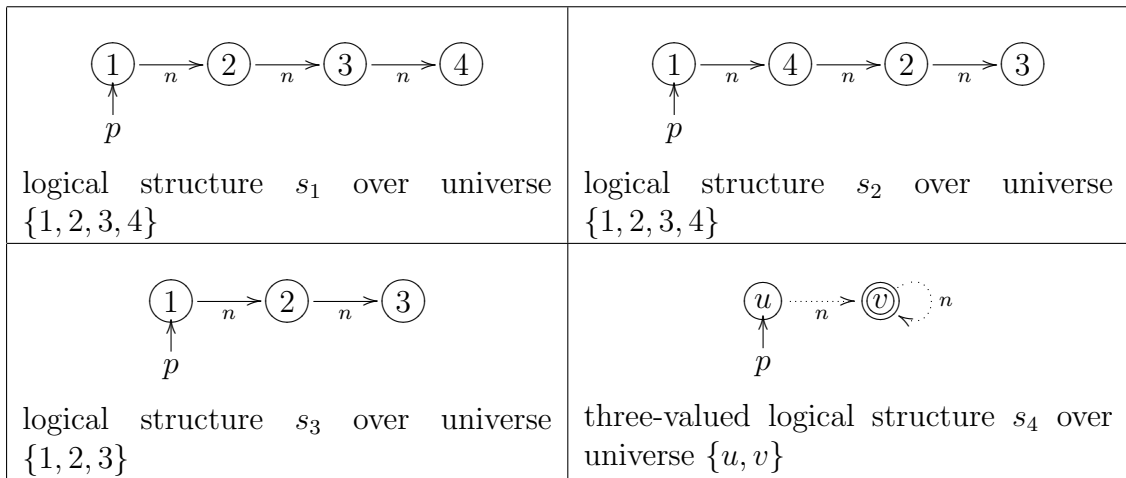


Figure 6.1: Four logical structures.

Figure 6.1 shows two labeled graphs  $s_1$  and  $s_2$ . They are isomorphic, which means that  $s_1$  can be obtained from  $s_2$  by swapping the labels  $U = \{1, 2, 3, 4\}$ . The graphs symbolizes logical structures  $s_1$  and  $s_2$ . We interpret the labels as individuals of the universe  $U$  of those two structures. The labeled edges give the valuation of two predicate symbols  $n, p$ . We can imagine that  $s_1$  and  $s_2$  are lists on the heap with root pointer  $p$  and next field  $n$ . The individuals are heap nodes. Suppose we want to obtain the tail of the two lists. We simply advance the root pointer. If we do this with each of the two structures  $s_1$  and  $s_2$ , we obtain two structures  $s'_1$  and  $s'_2$ , respectively.  $s'_1$  and  $s'_2$  are isomorphic again ( $s'_1$  is  $s_1$

with  $p$  pointing to 2,  $s'_2$  is  $s_2$  with  $p$  pointing to 4). One can observe the same effect with other list operations which can be expressed in terms of the predicates  $p, n$ . They do not care about labels.

From the perspective of the predicates  $n$  and  $p$ , we have changed their valuations on the universe  $U$  by re-labeling the nodes. One can say that we permuted the predicate valuations. This is not always desirable. It would be irritating if  $1 \leq 2 \leq 3 \leq 4$  would hold in  $s_1$  and  $1 \leq 4 \leq 2 \leq 3$  in  $s_2$ , but this is how we treated  $n$  and  $p$ . Such symbols are *not* to be permuted. Usually, their valuation is a part of the logic, and is not determined by the interpretation of the structure. The valuation of the equality symbol in our first-order logic is such an example. Equality is a *bad example* in terms of understanding symmetry; it is invariant under permutation. If we augment the predicate logic with the fixed  $\leq$ -symbol with the conventional valuation, the formula  $\exists u. \exists v. \exists w. p(u) \wedge n(u, v) \wedge n(v, w) \wedge v \leq w$  evaluates to true in  $s_1$  ( $2 \leq 3$ ) and to false in  $s_2$  (not  $4 \leq 2$ ). As the predicate logic of [SRW02] is traditionally used for models which do not look at node labels (names of heap nodes), the only fixed symbol is equality. Therefore, this predicate logic 2.3.1 cannot distinguish between isomorphic structures (formally in Lemma 6.2.4).

**Observation 6.1.1.** *The question whether a logic can distinguish between two logical structures becomes intricate as soon as symbols come into play which have a fixed valuation different from identity.*

We call structures which cannot be distinguished by our logic symmetric. Symmetry is an equivalence relation. The idea of symmetry reduction [ID96, GS99, ES96] is to look at as few states of the same symmetry class as possible, thus reducing the size of the searched state space. As indicated in Observation 6.1.1, fixed symbols pose a problem, because then it can happen that not all permutations preserve the logic. Checking symmetry on-the-fly can eat up the savings from the reduced state space.

**Syntactical Criteria.** The authors of [ID96] (and others) discovered that there are syntactical criteria (they called them "scalarset criteria") which ensure that permutations yield symmetric structures, structures which cannot be distinguished. The idea is to forbid all fixed-valuation symbols but equality. Arrays and variables are allowed. They correspond to predicates in our predicate logic. Constant symbols of fixed valuation, arithmetic operations and comparison operators (other than equality) are forbidden. The predicate logic of Definition 2.3.1 fulfills these syntactic restrictions. Predicate symbols correspond to arrays and variables. There are no symbols of fixed valuation except equality. In case of the first-order logic of Definition 2.1.2, we consider two structures isomorphic if individuals of the base types are interchanged. The first-order logic cannot distinguish such isomorphic structures. As the consideration for the general first-order logic is analogous to the one-sorted case of predicate logic, we omit a treatment of general first-order logic. Though, often, many-sorted logics are considered in verification. For example, [ID96] considers subranges of the integers. Operations on this domain look at data. So the restriction to equality is not acceptable, and so is mixing up numbers with other individuals through permutations. As a consequence, permutations are restricted to so-called symmetric types on which permutation is allowed. The user has to supply type annotations and the according criteria are checked for annotated types.

**Observation 6.1.2.** *First-order logic (2.1.2) with equality abides to the scalarset criteria [ID96].*

There is an application of the scalarset other than symmetry reduction. We discussed instantiations of universally quantified formulas in Chapter 4. In [McM00], it was discovered that, if the scalarset criteria hold for model and universally quantified property, it suffices to show finitely many instantiations. The number of these instantiations is determined by the equality relations among the quantification variables. The use of symmetry arguments for decomposition as in [McM00] is discussed in Chapter 7.

## 6.2 Canonical Abstraction and Symmetry

Our idea of symmetry (Definition 6.2.1) is simple:

*Two states are symmetric if predicate logic cannot distinguish between them.*

We have motivated this choice in the previous section. Symmetry reduction *does not* abstract infinite to finite structures, while canonical abstraction does. Symmetry reduction in our setting, is the map  $[\cdot]_{\sim}$ . Often, the equivalent of  $[\cdot]_{\sim}$  is hard to compute and finer reductions with less compression are used.  $[\cdot]_{\sim}$  yields the best compression one can obtain with a symmetry reduction. If states which are distinguishable by predicate logic would be considered symmetric, relevant states would be overlooked. Canonical Abstraction compresses symmetric structures automatically without annotations, syntax checks, or any extra cost.

The following definition formalizes our intuition.

**Definition 6.2.1 (Symmetry).** *Let  $s, s' \in 3Struct[\mathcal{P}]$ .  $s$  and  $s'$  are **symmetric** iff there exists a bijection  $f \in U^s \rightarrow U^{s'}$  such that  $\forall e \in \mathcal{FO}_{\mathcal{P}} \forall Z : \llbracket e \rrbracket(s)(Z) = \llbracket e \rrbracket(s')(f \circ Z)$ .*

*Let  $\sim := \sim_{\mathcal{P}} := \{(s, s') \mid s, s' \in 3Struct[\mathcal{P}], s \text{ and } s' \text{ are symmetric}\}$*

Structures are symmetric if they are isomorphic and our logic cannot distinguish the structures. Two-valued structures which can be uniquely described up to isomorphism by a predicate logic expression (this does not hold for three-valued structures), e.g.  $s_1$  of Figure 6.1 has the encoding:

$$\begin{aligned} \exists u, v, w, x. \quad & \bigwedge_{a,b \in \{u,v,w,x\}, a \neq b} \neg(a = b) \\ & \wedge p(u) \wedge \bigwedge_{a \in \{v,w,x\}} \neg p(a) \wedge \\ & n(u, v) \wedge n(v, w) \wedge n(w, x) \wedge \neg(n(v, u) \vee \dots) . \end{aligned}$$

Indistinguishable structures fulfill the same formulas, therefore they also fulfill formulas which uniquely determine a structure up to isomorphism. For two-valued structures, the intuition coincides with the definition, because indistinguishable structures are isomorphic.

The symmetry relation  $\sim_{\mathcal{P}}$  is parameterized in the predicate signature  $\mathcal{P}$ . For brevity we shall write  $\sim$  within this section. It is common that symmetry is relative to properties of interest.

**Lemma 6.2.1.**  *$\sim$  is an equivalence relation.*

**Definition 6.2.2.** We write  $[s]_{\sim}$  for the equivalence class of  $s \in 3Struct[\mathcal{P}]$  with respect to  $\sim$ .

The following theorem states the relation between Canonical Abstraction and  $[\cdot]_{\sim}$ :

**Theorem 6.2.2 (Canonical Abstraction and Symmetry).**

Let  $\mathcal{P}$  be a predicate signature with abstraction predicates  $\mathcal{A}$ . Then the canonical abstraction function  $\alpha_{can}^{\mathcal{P},\mathcal{A}}$  fulfills the following properties:

- (i)  $\forall s, s' \in 3Struct[\mathcal{P}] : s \sim s' \Rightarrow \alpha_{can}^{\mathcal{P},\mathcal{A}}(s) = \alpha_{can}^{\mathcal{P},\mathcal{A}}(s')$
- (ii)  $\forall s \in Struct[\mathcal{P}] : [s]_{\sim} \subseteq \gamma_3(\alpha_{can}^{\mathcal{P},\mathcal{A}}(s))$

*Proof.* The proof can be found in Lemma 6.2.4. □

The first statement (i) says that canonical abstraction maps symmetric structures to the same abstract structure. The implication arrow indicates that canonical abstraction is coarser than  $[\cdot]_{\sim}$  (which would have an equivalence arrow in the middle). This is not surprising since *finite* symmetric structures have the same size (however, our theorem and Canonical Abstraction do not impose finiteness). An abstract structure may represent multiple symmetry equivalence classes. Of course, (i) also holds for 2-valued structures since  $Struct[\mathcal{P}] \subsetneq 3Struct[\mathcal{P}]$ .

Figure 6.1 shows logical structures over the predicate signature  $\mathcal{P} = \langle \{p, n\}, \mathbb{N}, \{p \mapsto 1, n \mapsto 2\} \rangle$ . The logical structures  $s_1, s_2, s_3$  could, e.g., stand for lists with a pointer variable  $p$  and a next pointer  $n$ . Structures  $s_1$  and  $s_2$  are symmetric, denoted as  $s_1 \sim s_2$ , since  $s_2$  can be obtained from  $s_1$  by the permutation  $\{1 \mapsto 1, 2 \mapsto 4, 3 \mapsto 2, 4 \mapsto 4\}$ .  $s_1$  and  $s_3$  are not symmetric because they have a different universe. Now, we consider canonical abstraction with the abstraction predicates  $\mathcal{A} = \{p\}$ . We have  $\alpha_{can}^{\mathcal{P},\mathcal{A}}(s_1) = \alpha_{can}^{\mathcal{P},\mathcal{A}}(s_2) = \alpha_{can}^{\mathcal{P},\mathcal{A}}(s_3) = s_4$ . Among other things, that shows that in (i) the direction  $\Leftarrow$  does in general not hold, e.g.  $\alpha_{can}^{\mathcal{P},\mathcal{A}}(s_2) = \alpha_{can}^{\mathcal{P},\mathcal{A}}(s_3)$  but  $s_2 \not\sim s_3$ .

The second statement (ii) says that the canonical abstraction of a 2-valued structure  $s$  automatically represents its entire symmetry class  $[s]_{\sim}$ , e.g.  $s_2 \in [s_1]_{\sim} \subsetneq \gamma_3(s_4)$ .

**Corollary 6.2.3.** The concretization of a 3-valued structure  $\tilde{s}$  is the disjoint union

$$\gamma_3(\tilde{s}) = \bigcup_i [s_i]_{\sim}$$

of suitable symmetry classes  $[s_i]_{\sim}$ .

*Proof.* This is because  $\sim$  is an equivalence relation and because of 6.2.2 (ii). □

Let us turn to the proof of Theorem 6.2.2. The following lemma characterizes symmetric structures in terms of canonical abstraction and the concretization function  $\gamma_3$ .

**Lemma 6.2.4 (Symmetry Lemma).** Let  $\mathcal{P}, \mathcal{A}$  be defined as in 6.2.2. Let  $s, s' \in 3Struct[\mathcal{P}]$  be two symmetric structures,  $s \sim s'$  such that  $f \in U^s \rightarrow U^{s'}$  is a bijection with  $\forall e \in \mathcal{FO}_{\mathcal{P}} \forall Z : \llbracket e \rrbracket(s)(Z) = \llbracket e \rrbracket(s')(f \circ Z)$ .

- (a)  $\forall e \in \mathcal{FO}_{\mathcal{P}} \forall Z' : \llbracket e \rrbracket(s')(Z') = \llbracket e \rrbracket(s)(f^{-1} \circ Z')$

(b)  $\alpha_{can}^{\mathcal{P},\mathcal{A}}(s) = \alpha_{can}^{\mathcal{P},\mathcal{A}}(s')$

(c)  $s \sqsubseteq_f s'$  is a tight embedding.

(d) if  $s \in Struct[\mathcal{P}]$  we have  $[s]_{\sim} \subseteq \gamma_3(\alpha_{can}^{\mathcal{P},\mathcal{A}}(s))$

*Proof.* see B.5

□

# Chapter 7

## Finite Instantiation and Data Type Reduction

Finite instantiation and data type reduction are a part of the compositional methodology [McM00]. The methodology is implemented in the verification tool SMV, SMV is mainly used for hardware verification. The verification of a model in SMV consists of several techniques which are used in combination. Therefore we do not consider data type reduction in isolation. Canonical abstraction (5.1.11) is part of the three-valued analysis framework [SRW02]. We compare data type reduction with canonical abstraction.

**Logics and Models.** Data type reduction is defined in a first-order logic with function symbols (the first-order logic of Definition 2.1.2 without transitive closure). A constant-domain semantics is considered. The models of [DW03] correspond to the models we consider (Definition 2.2.3).

Canonical abstraction (5.1.11) is defined in a first-order logic without function symbols but with transitive closure (Definition 2.3.1). A varying-domain semantics is used to model allocation and deallocation. Models used in three-valued logical analysis are action-based and can modify the universe.

The syntactic restrictions imposed in [DW03, McM00] (*scalarset* criteria) are implicitly given in the predicate logic of [SRW02]. This is explained in Chapter 6.

We tried to bridge the gap in terms of logics by finding a counterpart of canonical abstraction for first-order logic with function symbols. We did not find a method for general first-order logic which subsumes the concept of canonical abstraction. One can give an embedding order, however it does not subsume its predicate logic counterpart as a special case. We discuss this in Appendix A.

**Mechanism.** Data type reduction is parameterized by sets of individuals that are provided as inputs. For each base type  $T$ , a subset  $D_T$  of the semantic domain  $\Delta(T)$  is given. This *partition* remains fixed during the entire analysis. We will refer to this choice of individuals as *the partition*. The complement of  $D_T$  in  $\Delta(T)$  is abstracted to one abstract summary individual  $sum_T$ . The abstract semantic domain consists of singletons from  $D_T$  and the summary individual plus a top element, i.e.  $\tilde{\Delta}(T) = \{\{u\} \mid u \in D_T\} \cup \{sum_T, \top_T\}$  is the abstract semantic domain of  $T$ . The values  $\top_T$  are needed to account for uncertainty.

Abstract models are constant-domain models. The valuations of the function symbols are adjusted to be conservative. Equality and function symbols have to be modified. The comparison  $sum_T = sum_T$  cannot be conservatively answered with true or false, so  $\top_{Bool}$  is returned. The interpretation of equality is

$$\llbracket t_1 = t_2 \rrbracket = \begin{cases} \top_{Bool} & ; \quad sum \in \{\llbracket t_1 \rrbracket, \llbracket t_2 \rrbracket\} \\ 1 & ; \quad \llbracket t_1 \rrbracket = \llbracket t_2 \rrbracket = \{v\} \\ 0 & ; \quad \text{otherwise} \end{cases}$$

*Data type reduction loses all information about summary individuals.* A function symbol  $f$  of rank  $r(f) = (T_1, T_2)$  has to return a conservative result so  $f(sum_{T_1}) = \top_{T_2}$ . An abstract model is extracted from the concrete model which simulates the concrete model. The abstract model is constant-domain with the semantic domains described above.

Canonical abstraction is parameterized by a set of unary abstraction predicates. It computes a normal form on three-valued logical structures. Individuals are mapped to equivalence classes according to their valuation under abstraction predicates. The equivalence classes are interpreted as individuals of a new abstract universe. An interpretation which is conservative with respect to the input state is computed with minimal information loss. The result is the canonical abstraction of the input structure. Abstraction predicates can change their valuation. *Canonical abstraction is applied during state space exploration* to ensure termination. Thus the universes are dynamically computed during the analysis. Unlike in case of data type reduction there can be multiple summary individuals and there is no fixed partition. Canonical abstraction preserves information about summary individuals. In so far, canonical abstraction preserves information globally and data type reduction preserves information only locally. Furthermore, regions with more information can move dynamically when canonical abstraction is used. For example, when we verified the Ticket Protocol active processes and their ticket were materialized (using the focus operation).

**Sources of the parameters.** Data type reduction is used in combination with finite instantiation. Finite instantiation provides instantiation values, but one can also give individuals which are to be kept material. So the user provides the partition directly or indirectly through the property and finite instantiation.

Abstraction predicates are provided by hand or by abstraction refinement heuristics. For particular applications, there exists a suitable vocabulary and actions which model entire program statements.

**Complexity and Running Times.** The fact that data type reduction uses a fixed partition and that syntactic restrictions hold (*scalarset* criteria) facilitates the computation of abstract models. The abstract model can be transformed into a Boolean program. For each data variable, one needs logarithmically<sup>1</sup> many Boolean variables. Arrays are considered as a collection of variables. The state space of the Kripke structure has size  $2^n$  where  $n$  is the number of Boolean variables. Symbolic model checking can be used.

---

<sup>1</sup>logarithmic in the size of the corresponding semantic domain



The state space of three-valued logical analysis consists of logical structures (this has an effect on the worst-case running time). The abstraction function and the constraint mechanism are used during state space exploration. During our case study the constraint mechanism was often the most expensive operation. Practical experience indicates that the predicate choice is the most important factor. Sometimes more predicates lead to shorter analysis times. How forecasts about the running time of three-valued logical analysis can be made, is being investigated. As both SMV and TVLA use BDDs, variable ordering plays a role. Thus the problem of determining average-case running times is hard.

**Trading precision for running time.** As SMV uses a symbolic model checker, releasing constraints and introducing more nondeterminism leads to smaller BDDs and thus shorter analysis times. In TVLA, more nondeterminism would mean longer analysis times. Releasing constraints (consistency constraints) can mean longer analysis times because more structures are visited or a shorter analysis time because less constraints have to be checked. Obviously, the tendency is that the less predicates are used the shorter the analysis time.

**Strategy.** Verification following the methodology [McM00] is a compositional process. A correctness proof is broken down into small parts. The idea is that the smaller parts are more manageable because a smaller portion of the system is involved. There are several methods in SMV which support the user in this process.

- Refinement means that a concrete processor model is checked against a simpler abstract model. For example, one can verify that a router corresponds to a simpler model of end-to-end data transfer. Refinement maps relate signals of the concrete with the signals of the abstract model. Each signal provides a separate subgoal.
- Case splitting allows one to split cases on the value of a variable. One shows that a property holds for each value  $i$  variable  $v$  can take on. The idea is that

$$\forall i. \mathbf{AG}(v = i \Rightarrow \phi) \text{ implies } \mathbf{AG} \phi.$$

Finite instantiation applies to these properties, i.e. they are decomposed into subgoals. Case splitting ensures that data type reduction does not collapse the value  $v$  has.

- Spurious counterexamples are removed by providing lemmas and proving them.
- SMV comes with a proof assistant which allows for circular proofs, induction on time and compositional reasoning.

[McM00] considers data type reduction as one building block. Data type reduction is more predictable and simpler than canonical abstraction. This makes it less flexible while computations are much cheaper. Our experience is that most work goes into finding the right lemmas. Data type reduction leads to counterexamples which, in general, cannot be removed by changed parameters to data type reduction.

The strategy in most work about shape analysis with three-valued logical analysis is to verify a property in one verification run and to refine the abstraction until the analysis *runs through*. Verification problems are in general not decomposed (an exception is [YR04]). Spurious counterexamples can be removed by

- providing instrumentation predicates.
- changing the parameters to the focus operation.
- constraints.

Abstraction is at the center of the concept of three-valued analysis. Finding good predicates is the challenge.

**Non-interference.** When using data type reduction the number of components about which information is maintained is decreased to but a few distinguished ones. All information about the other components is lost. However, the abstracted components corrupt the data of the components kept material. The case studies [JM01, McMa, McMb], where processors and cache protocols were verified with SMV, have in common that there are mostly exclusive pair relationships among components, e.g. a processor has *its own* cache [McMa], "while a reservation station is expecting a result from a given execution unit not other execution unit returns a result for that particular reservation station" [McMb, JM01]. For a period of time, there is an exclusive relation between components, i.e. other components do not interfere. These system properties need to be provided as so-called non-interference lemmas (of course, they need to be proved as well).

However, if there are no exclusive relations among a few components, there *can* be interference. The Ticket Protocol is an example. All processes manipulate counter variables and the behavior of all processes depends on those counter variables. Therefore, it is hard to separate two processes and show mutual exclusion for them. It is not sufficient to prove that two processes have distinct tickets. Assume that one of the two processes is critical. The abstract processes can increase the winner ticket, allowing the second process to enter its critical section. We discuss this example in detail (Section 7.1). It is not necessary to know SMV well to understand our explanations. The crucial difference compared to verification with canonical abstraction is that canonical abstraction preserves invariants about all processes including summary individuals.

## 7.1 Case Study: SMV

What follows is a description of our case study with SMV. The complete code can be found in Appendix C.2, and will be made available with this document.

We cannot use [DW03] alone. As there is a successor function on the tickets and thus the tickets are not fully symmetric (this breaches the *scalarset* criteria). Fortunately, we can use the data type *ordset* of SMV which gives us an unbounded counter type with a successor function.

Modeling the Ticket Protocol in SMV is easy. SMV models are very similar to our formal models (cf. Definition 2.2.3). We start by giving a signature. The type definitions consist of a type PROC for processes, a type TICKET and a type LOC for locations.

```

scalarset PROC undefined;
ordset TICKET 0..;
typedef LOC {think,wait,crit};

```

The process indices PROC are a scalarset type, i.e., a type with full symmetry. Next we define symbols.

```

s : TICKET;
t : TICKET;
a : array PROC of TICKET;
at: array PROC of LOC;
act:PROC;

```

We come to the initial states,  $\theta$ , speaking in terms of formal models of Definition 2.2.3. The keyword *init* means that the initial valuation is specified. Initially, we set all counters to zero and the locations to *think*:

```

init(s):=0;
init(t):=0;

forall (p in PROC){
  init(at[p]):=think;
  init(a[p]):=0;
}

```

We define the transition relation  $\rho$ . The keyword *next* corresponds to primed symbols and thus to the valuation of a symbol in the next state, e.g.,  $next(a[act]) := t$ ; means  $a'(act) = t$ . Below the code of a process is shown:

```

switch(at[act]){
  think: { next(a[act]):=t;
           next(t):=t+1;
           next(at[act]):=wait;
         }
  wait:  { if(a[act]=s) {
           next(at[act]):=crit;
         }
         }
  crit:  { next(s):=s+1;
           next(at[act]):=think;
         }
}

```

which also agrees with the pseudo-code and our formalization of the Ticket Protocol. Note that *act* is the process which is currently active. *act* is a free unconstrained symbol and thus nondeterministically picks a process which is to be executed.

Our goal is to show mutual exclusion. In the SMV language, we can write mutual exclusion as:

```
forall(p in PROC) forall(q in PROC)
mutex[p][q]: assert G (at[p]=crit & at[q]=crit)->p=q)
```

Since there was no bound on the number of processes, the above specification consists of infinitely many subgoals. However, symmetry arguments reduce the above problem to a finite number of instances. The idea is the following:

*"... exchanging the roles of any two values of the type has no effect on the semantics of the program. In order to ensure that this symmetry exists, there are a number of rules placed on the use of variables of a scalarset type. "*

Ken McMillan, SMV tutorial, [McMb]

One cannot use symbols of fixed valuation on a scalarset type, the only operation allowed on scalarset quantities is equality. In addition, one cannot mix scalarset values with values of any other type. One can, however, declare a function or predicate symbol with scalarset arguments. This makes it legal to make PROC a scalarset. When SMV encounters an array of properties whose index is of scalarset type, it chooses instances to prove, since if it can prove these cases, then by symmetry it can prove all of them, i.e., here it suffices to prove  $mutex[0][0]$  and  $mutex[0][1]$ . There is more than one quantifier. The two quantification variables could have the same value. As equality is allowed, it is necessary to distinguish between the cases where the quantifier variables are equal and unequal, respectively. Showing either of the instances  $mutex[0][1]$  and  $mutex[0][0]$  alone would not account for all cases. SMV produces two verification problems  $mutex[0][1]$  and  $mutex[0][0]$  which account for all equality relations between  $p$  and  $q$ . The individuals of the base type  $PROC$  are collapsed to 0, 1 and one summary individual (SMV calls it  $NaN$ ). The tickets are collapsed to one summary individual. As expected, we get a spurious counterexample as the abstraction is too coarse. Two processes can possibly have the same ticket and become critical simultaneously.

We perform temporal case splitting on the value of ticket  $a[p]$  of process  $p$ , which means that we add a quantifier  $i$  to the mutual exclusion property and a premise to the mutual exclusion temporal formula:

```
a[p]=i ->((at[p]=crit & at[q]=crit)->p=q).
```

The idea is that, if we can show the property for each value the variable can take on, the property holds. This is done in order keep the tickets of  $p$  and  $q$  apart:

```
forall (p in PROC) forall(q in PROC) forall (i in TICKET)
{ mutex[p][q][i]:
  assert G(at[p]=crit & at[q]=crit)->p=q) for a[p]=i;
  using TICKET -> {i} prove mutex[p][q][i];
}
```

The *using-prove* construct usually specifies assumptions but also the abstraction. We abstract the tickets  $TICKET \rightarrow \{i\}$ , namely  $i$  is kept material. The other tickets are abstracted to a summary individual.

Unfortunately, there is still a spurious counterexample. The first material process 0 enters its critical section. Now the chaos process, the summary process which represents all

processes except  $p$  and  $q$  interferes. It increments the global variable  $s$ , which contains the winner ticket. Thus the other material process 1 can also enter its critical section. When canonical abstraction was used this counterexample could not occur because canonical abstraction maintains information about summary individuals.

We added another lemma. It claims that, while a process is critical, the winner ticket remains the same:

```
forall (p in PROC) forall(i in TICKET){
  remain[p][i]: assert G( (at[p]=crit&a[p]=i) -> (s=i)U(at[p]=think));
}
```

Using this lemma, one can prove *mutex*. However, SMV was not able to show *remain*. We were not able to successfully use compositional techniques to show mutual exclusion.

**Spurious Counterexamples.** Data type reduction allows only one static summary individual, and discards all information about this summary individual. If process indices are abstracted, the effect is that this results in a "chaos" process (pertaining to the summary process index). It manipulates the winner ticket, global variable  $s$ . Thus one more process can enter its critical section. In case of spurious counterexamples, [DW03] proposes to use non-interference lemmas. It was not clear to us how to apply the concept of non-interference to the Ticket Protocol.

# Chapter 8

## Conclusion

A method for checking quantified temporal properties of systems with infinite data domains and an unbounded number of components was given. It is based on a symbolic quantifier elimination technique, called Skolemization. We have approximated models produced by Skolemization with three-valued logical analysis. Furthermore, we have discussed advantages and disadvantages of using finite instantiation and data type reduction compared to our three-valued logical analysis.

### 8.1 Future Work

Skolemization does not require a particular abstraction. Combining our methods with Symbolic Shape Analysis (cf. [Wie04]) is a promising idea because Skolemization is a symbolic method. In three-valued shape analysis, often varying-domain semantics are used. Extending Skolemization to a varying-domain semantics is another challenge. A more generic implementation of Skolemization and heterogeneous abstraction in TVLA (due to costly constraint checks the analysis is currently slowed down), would improve the performance of the three-valued analysis we have presented. Decision procedures [YRS03] promise more automation and more precise transition relations. In order to take full advantage of our method, one needs transition systems. TVLA did not compute transition systems at the time of this writing.

The construction of abstractions which make use of the embedding order for first-order logic with function symbols is still to be done (see Appendix A).

# Bibliography

- [BCDR04] Thomas Ball, Byron Cook, Satyaki Das, and Sriram Rajamani. Refining approximations in software predicate abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 388–403. Springer-Verlag, March 2004.
- [BCG95] Bhat, Cleaveland, and Grumberg. Efficient on-the-fly model checking for CTL. In *LICS: IEEE Symposium on Logic in Computer Science*, 1995.
- [BCR] Thomas Ball, Byron Cook, and Sriram Rajamani. SLAM project.
- [Ber02] Sergey Berezin. *Model Checking and Theorem Proving: A Unified Framework*. PhD thesis, Carnegie Mellon University, January 2002.
- [BMMR01] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.
- [BPR01] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. *Lecture Notes in Computer Science*, 2031:268+, 2001.
- [BPR02] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Relative completeness of abstraction refinement for software model checking. In Joost-Pieter Kaoen and Perdita Stevens, editors, *Proceedings of TACAS02: Tools and Algorithms for the Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 158–172. Springer-Verlag, 2002.
- [BR00] Thomas Ball and Sriram K. Rajamani. Bebop: A symbolic model checker for boolean programs. In *SPIN*, pages 113–130, 2000.
- [CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fix-points. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.
- [CC00] Patrick Cousot and Radhia Cousot. Temporal abstract interpretation. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–25. ACM Press, 2000.

- [CCG] Sagar Chakar, Edmund Clarke, and Alex Groce. MAGIC project.
- [CD89] Edmund M. Clarke and I.A. Draghicescu. Expressibility results for linear-time and branching-time logic. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *REX Workshop*, volume 354 of *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, School/Workshop, Noordwijkerhout, The Netherlands, May 30 - June 3, 1988, Proceedings, LNCS*, pages 428–437. Springer, 1989.
- [CDEG03] M. Chechik, B. Devereux, S. Easterbrook, and A. Gurfinkel. Multi-valued symbolic model-checking, 2003.
- [CG03] Edmund M. Clarke and Orna Grumberg. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
- [CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
- [CGP00] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.
- [Cou96] Patrick Cousot. Abstract interpretation. *ACM Comput. Surv.*, 28(2):324–328, 1996.
- [DGG97] Dennis Dams, Rob Gerth, and Orna Grumberg. Abstract interpretation of reactive systems. *ACM Trans. Program. Lang. Syst.*, 19(2):253–291, 1997.
- [DH01] Werner Damm and David Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.
- [DK01] Werner Damm and Jochen Klose. Verification of a radio-based signaling system using the STATEMATE verification environment. *Formal Methods in System Design*, 19(2):121–141, 2001.
- [DL97] Ekaterina Dolginova and Nancy A. Lynch. Safety verification for automated platoon maneuvers: A case study. In *HART*, pages 154–170, 1997.
- [DPJ03] Werner Damm, Amir Pnueli, and Bernhard Josko. Understanding uml: A formal semantics of. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *Formal Methods for Components and Objects, First International Symposium, FMCO 2002, Leiden, The Netherlands, November 5-8, 2002, Revised Lectures*, volume 2852 of *Lecture Notes in Computer Science*, 2003.
- [DW03] Werner Damm and Bernd Westphal. Live and Let Die: LSC-based verification of UML-models. In *Formal Methods for Components and Objects, FMCO 2002*, volume 2852 of *Lecture Notes in Computer Science*, pages 99–135. Springer, 2003.



- [ES96] E. Allen Emerson and A. Prasad Sistla. Symmetry and model checking. *Form. Methods Syst. Des.*, 9(1-2):105–131, 1996.
- [FB95] G.W.S. Friedrichsen and R.W. Burchfield. *The Oxford Dictionary of English Etymology*. Number 0-19-861112-9. Oxford at the Clarendon Press, 1995.
- [GHJ01] Patrice Godefroid, Michael Huth, and Radha Jagadeesan. Abstraction-based model checking using modal transition systems. *Lecture Notes in Computer Science*, 2154:426+, 2001.
- [GS99] Viktor Gyuris and A. Prasad Sistla. On-the-fly model checking under fairness that exploits symmetry. *Formal Methods in System Design: An International Journal*, 15(3):217–238, November 1999.
- [HJS01] Michael Huth, Radha Jagadeesan, and David Schmidt. Modal transition systems: A foundation for three-valued program analysis. *Lecture Notes in Computer Science*, 2028:155+, 2001.
- [HM] Thomas Henzinger and Rupak Majumdar. BLAST.
- [HP95] Gerard J. Holzmann and Doron Peled. An improvement in formal verification. In *Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques VII*, pages 197–211. Chapman & Hall, Ltd., 1995.
- [ID96] C. N. Ip and D. L. Dill. Better verification through symmetry. *Form. Methods Syst. Des.*, 9(1-2):41–75, 1996.
- [JM01] Ranjit Jhala and Kenneth L. McMillan. Microarchitecture verification by compositional model checking. In G. Berry, H. Comon, and A. Finkel, editors, *Computer-Aided Verification (CAV'01)*, number 2102 in Lecture Notes in Computer Science, pages 396–410, Paris, France, jul 2001. Springer-Verlag.
- [Lah04] Shuvendu Lahiri. *Unbounded System Verification using Decision Procedure and Predicate Abstraction*. PhD thesis, Carnegie Mellon University, September 2004.
- [Lon93] David E. Long. *Model Checking, Abstraction, and Compositional Verification*. PhD thesis, Carnegie Mellon, 1993.
- [LSW] Kim G. Larsen, Bernhard Steffen, and Carsten Weise. A constraint oriented proof methodology based on modal transition systems. pages 13–28.
- [McMa] Kenneth L. McMillan. Parameterized verification of the flash cache coherence protocol by compositional model checking. In *CHARME 2001*.
- [McMb] Kenneth L. McMillan. SMV.
- [McM92] Kenneth L. McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, Carnegie Mellon University, 1992.

- [McM00] Kenneth L. McMillan. A methodology for hardware verification using compositional model checking. *Sci. Comput. Program.*, 37(1-3):279–309, 2000.
- [MP95] Zohar Manna and Amir Pnueli. *Temporal verification of reactive systems: safety*. Springer-Verlag New York, Inc., 1995.
- [MPC<sup>+</sup>] Madanlal Musuvathi, David Park, Andy Chou, Dawson Engler, and David Dill. CMC model checker.
- [MQS00] Kenneth L. McMillan, Shaz Qadeer, and James B. Saxe. Induction in compositional model checking. In *Computer Aided Verification*, pages 312–327, 2000.
- [MYRS05] Roman Manevich, Eran Yahav, G. Ramalingam, and Mooly Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In Radhia Cousot, editor, *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI 2005*, Lecture Notes in Computer Science. Springer, jan 2005.
- [NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
- [OS03] Sam Owre and N. Shankar. Writing pvs proof strategies. In Myla Archer, Ben Di Vito, and César Muñoz, editors, *STRATA 2003*, number CP-2003-212448 in NASA Conference Publication, pages 1–15, Hampton, VA, September 2003. NASA Langley Research Center.
- [RSL03] T. Reps, M. Sagiv, and A. Loginov. Finite differencing of logical formulas for static analysis, 2003.
- [SRW<sup>+</sup>] Mooly Sagiv, Thomas Reps, Reinhard Wilhelm, E. Yahav, and Roman Manevich. TVLA.
- [SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 2002.
- [SS99] Hassen Saïdi and Natarajan Shankar. Abstract and model check while you prove. In Nicolas Halbwachs and Doron Peled, editors, *Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 443–454, Trento, Italy, jul 1999. Springer-Verlag.
- [Wie04] Thomas Wies. Symbolic Shape Analysis. Master's thesis, Universität des Saarlandes, 2004.
- [Yah01] E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. *ACM SIGPLAN Notices*, 36(3):27–40, March 2001.

- [YR04] E. Yahav and G. Ramalingam. Verifying safety properties using separation and heterogeneous abstractions. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 25–34. ACM Press, 2004.
- [YRS01] E. Yahav, T. Reps, and M. Sagiv. LTL model checking for systems with unbounded number of dynamically created threads and objects. Technical Report TR-1424, Computer Sciences Department, University of Wisconsin, Madison, WI, March 2001.
- [YRS03] Greta Yorsh, Thomas Reps, and Mooly Sagiv. Symbolically computing most-precise abstract operations for shape analysis. Technical report, School of Computer Sciences, Tel Aviv University, Sept. 2003.
- [YRSW03] E. Yahav, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Verifying Temporal Heap Properties Specified via Evolution Logic. In *European Symposium on Programming*, volume 2618 of *Lecture Notes in Computer Science*, pages 204 – 222. Springer-Verlag, 2003.
- [Zha04] Lijun Zhang. Logic and model checking for hidden markov models. Master’s thesis, Universität des Saarlandes, Dependable Systems and Software Group, Prof. Hermanns, March 2004.

# Appendix A

## Extended Embedding Order

Canonical abstraction is based on the concept of embeddings and tight embeddings (see Definition 5.1.9). It merges together values to clusters, which make up the abstract universe, and computing a best conservative interpretation of the predicate symbols on the abstract universe. We want to give an Extended Embedding Order (cf. Definition 5.1.7) for first-order logic with general function symbols. This means that function symbols may return individuals. Hence, if we want to resolve inconsistencies by a join-operator, we need a semi-lattice structure on the universe, too.

As an example, we consider an array. The table below shows the valuation  $\iota(a)$  of a function symbol  $a$  which models an array  $A$  with index and data domain  $\{1, \dots, 4\}$ .

$i$	1	2	3	4
$A[i] = \iota(a)(i)$	1	3	2	4

We abstract 1 to an abstract value  $u$  and 2 is abstracted to  $v$ , 3 and 4 we merge together to an abstract value  $w$ . So we have an abstract index and data domain  $\{u, v, w\}$  where  $u$  stands for  $\gamma(u) = \{1\}$ ,  $v$  for  $\gamma(v) = \{2\}$  and  $w$  for  $\gamma(w) = \{3, 4\}$ . What valuation  $\tilde{\iota}$  should be given to  $a$  in the abstract? For  $u$  it must be  $\tilde{\iota}(a)(u) = u$  and for  $v$  it must be  $\tilde{\iota}(a)(v) = w$ . But what about  $\tilde{\iota}(a)(w)$ ? Clearly, none of the candidates  $v, w$  is conservative. 2 abstracts to  $v$  and 4 to  $w$ . Somehow we would like to take both  $v$  and  $w$ . However, what value should that be?

$i$	$u$	$v$	$w$
$\gamma(i)$	$\{1\}$	$\{2\}$	$\{3, 4\}$
$\tilde{\iota}(a)(i)$	$u$	$w$	?

Being conservative means that information extracted from an abstract state also holds for all the concrete states represented by the abstract state. The abstract state  $\tilde{s}_0$  we obtain by choosing an interpretation of  $p$  which is always 0 is not conservative. The formula  $\exists u, v : p(u, v)$  evaluates to 1 in the concrete state  $s$  above and to 0 in  $\tilde{s}_0$ . Obtaining a conservative abstraction of  $s$  given just the truth values 0, 1 is problematic. There is "contradictory" information as to  $p(w, w)$  because  $\iota(p)(3, 3) = 0, \iota(p)(4, 4) = 1$ .

Note that we cannot find a conservative interpretation of  $a$  on the three abstract values  $u, v, w$  since we can only assign *one* value to  $\tilde{\iota}(p)(w)$ .

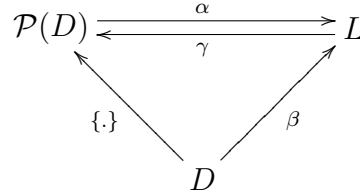
**Finding optimal conservative interpretations.** Conservativeness and optimality are obtained by abstract interpretation. A representation function  $\beta \in D \rightarrow L$  maps "concrete" values to "abstract" values. The abstract values are elements of a lattice  $L$  with partial order  $\sqsubseteq$  and join operator  $\bigsqcup$ .  $\sqsubseteq$  expresses information order, i.e.,  $l_1 \sqsubseteq l_2$  if  $l_2$  conveys less information than  $l_1$ . Given a subset  $L' \subseteq L$  the join operator  $\bigsqcup \in \mathcal{P}(L) \rightarrow L$  returns the least upper bound of  $L'$  with respect to  $\sqsubseteq$ . This gives rise to a Galois connection (cf. Definition B.1.1)

$$\langle \mathcal{P}(D), \alpha, \gamma, L \rangle$$

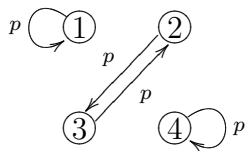
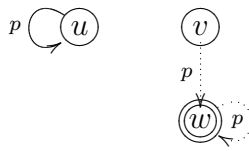
between  $\mathcal{P}(D)$  and  $L$  defined by

$$\begin{aligned} \alpha(D') &= \bigsqcup\{\beta(v) \mid v \in D'\} \\ \gamma(l) &= \{v \in D \mid \beta(v) \sqsubseteq l\} \end{aligned}$$

for  $D' \subseteq D$  and  $l \in L$  (proof see B.1, the approach is taken from [NNH99]).  $\alpha(D')$  is the least lattice element which represents  $D'$  and  $\gamma(l)$  is the greatest set which is represented by  $l$ . Furthermore,  $\alpha(\{v\}) = \beta(v)$ . We obtain the following diagram:



**Canonical Abstraction.** In pure-predicate calculus the array  $A$  is modeled as a binary predicate  $p$ . The concrete valuation of  $a$  corresponds to a predicate valuation of  $p$ . This is displayed in the following figure:

concrete valuation of the predicate $p$	abstract valuation given the embedding function $1 \mapsto u, 2 \mapsto v, 3, 4 \mapsto w$																																									
<table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th><math>\iota'(p)</math></th> <th>1</th> <th>2</th> <th>3</th> <th>4</th> </tr> </thead> <tbody> <tr> <td>1</td> <td>1</td> <td></td> <td></td> <td></td> </tr> <tr> <td>2</td> <td></td> <td></td> <td>1</td> <td></td> </tr> <tr> <td>3</td> <td></td> <td>1</td> <td></td> <td></td> </tr> <tr> <td>4</td> <td></td> <td></td> <td></td> <td>1</td> </tr> </tbody> </table>  <p style="text-align: center;">the edges model the valuation of <math>p</math></p>	$\iota'(p)$	1	2	3	4	1	1				2			1		3		1			4				1	<table border="1" style="margin-left: auto; margin-right: auto;"> <thead> <tr> <th><math>\tilde{\iota}'(p)</math></th> <th>u</th> <th>v</th> <th>w</th> </tr> </thead> <tbody> <tr> <td>u</td> <td>1</td> <td></td> <td></td> </tr> <tr> <td>v</td> <td></td> <td></td> <td>1/2</td> </tr> <tr> <td>w</td> <td></td> <td>1/2</td> <td>1/2</td> </tr> </tbody> </table>  <p style="text-align: center;">dashed edges stand for indefinite valuation</p>	$\tilde{\iota}'(p)$	u	v	w	u	1			v			1/2	w		1/2	1/2
$\iota'(p)$	1	2	3	4																																						
1	1																																									
2			1																																							
3		1																																								
4				1																																						
$\tilde{\iota}'(p)$	u	v	w																																							
u	1																																									
v			1/2																																							
w		1/2	1/2																																							

After merging together concrete individuals of a universe  $U$ , we want to compute conservative interpretations for predicate symbols over the abstract universe  $W$ . Merging individuals is described as a surjection  $h \in U \rightarrow W$ , here  $h \in \{0, 1, 2, 3, 4\} \rightarrow \{u, v, w\}$

where  $h(1) = u, h(2) = v, h(3) = h(4) = w$ . For some predicate symbol  $p$  its interpretation in the concrete is a function  $f \in U^n \rightarrow \mathbb{K}$ . The predicate that models array  $A$  is a two-place predicate, i.e.,  $n = 2$ . The most precise conservative interpretation of predicate symbol  $p$  is a function function  $F \in W^n \rightarrow \mathbb{K}$ . We want to obtain this interpretation using the idea of a representation function.

$\mathbb{K} = \{0, 1, 1/2\}$  lacks a least element<sup>1</sup>. A least element would stand for the empty set of Boolean values.  $\perp$  never occurs, hence it is ignored<sup>2</sup>. So we set  $D = \mathbb{K} = L$  and choose  $\beta$  as the identity on  $\mathbb{K}$ .

Conservative means

$$\forall u_1, \dots, u_n \in U : f(u_1, \dots, u_n) \sqsubseteq F(h(u_1), \dots, h(u_n)) .$$

The best possible interpretation is

$$\begin{aligned} F(w_1, \dots, w_n) &= \bigsqcup \{f(u_1, \dots, u_n) \mid \forall i \in \{1, \dots, n\} : u_i \in U \wedge h(u_i) = w_i\} \\ &= \alpha(\{f(u_1, \dots, u_n) \mid \forall i \in \{1, \dots, n\} : u_i \in U \wedge h(u_i) = w_i\}) . \end{aligned}$$

for all  $w_1, \dots, w_n \in W$ . In [SRW02], this best interpretation is termed *tight embedding*.

We compute the abstract valuation of  $p$  on  $(w, w)$  as

$$\begin{aligned} \iota(p)(w, w) &= \bigsqcup \{\iota'(p)(3, 3), \iota'(p)(3, 4), \iota'(p)(4, 3), \iota'(p)(4, 4)\} \\ &= \bigsqcup \{0, 1\} = 1/2 \end{aligned}$$

and analogously  $\iota(p)(w, v) = 1/2$ .

**General function symbols.** Before coming to the general case we consider the simple case where there is just one base type. We assume that the concrete universe is  $D$  and the abstract universe is a lattice  $L$ . For an interpretation of an  $n$ -place function symbol, a function  $f \in D^n \rightarrow D$ , we want to find a precise *and* conservative interpretation  $f_L \in L^n \rightarrow L$  of  $f$ . Conservative, means  $\forall l_1, \dots, l_n \in L \forall v_1 \in \gamma(l_1) \dots \forall v_n \in \gamma(l_n) : f(v_1, \dots, v_n) \in \gamma(f_L(l_1, \dots, l_n))$ . This is equivalent to  $\forall l_1, \dots, l_n \in L \forall v_1 \in \gamma(l_1) \dots \forall v_n \in \gamma(l_n) : \alpha(\{f(v_1, \dots, v_n)\}) \sqsubseteq f_L(l_1, \dots, l_n)$ . Now it is clear that the most precise ( $\sqsubseteq$ -least) conservative interpretation of  $f$  is

$$f_L(l_1, \dots, l_n) = \alpha(\{f(v_1, \dots, v_n) \mid \forall 1 \leq i \leq n : v_i \in \gamma(l_i)\})$$

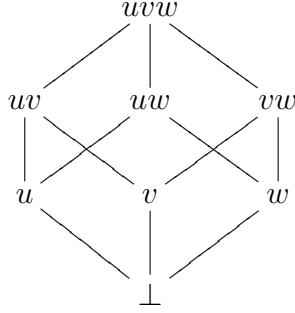
for all  $l_1, \dots, l_n \in L$ .

Let us apply our newly gained knowledge. In our example, the set of concrete values is  $D = \{1, 2, 3, 4\}$ . It seems easy: we just have to find a complete lattice and a representation function  $\beta$ , the rest follows automatically. We play the idea through with two different lattices  $L_1 = \{\perp, u, v, w, uv, uw, vw, uvw\}$  and  $L_2 = \{\perp, u, v, w, uvw\}$ .  $L_1$  is  $\mathcal{P}(\{u, v, w\})$  up to isomorphism and  $L_2$  is  $\{u, v, w\}$  plus a bottom and a top element. We begin with

<sup>1</sup> Leaving out the  $\perp$  element is allowed because an embedding function  $h \in U \rightarrow W$  is surjective and  $\emptyset \neq \{f(u_1, \dots, u_n) \mid \forall i \in \{1, \dots, n\} : u_i \in U \wedge h(u_i) = w_i\} \subseteq \{0, 1, 1/2\}$ .

<sup>2</sup> Similar to relational abstraction in [BPR01].

$L_1$ . The ordering  $\sqsubseteq_1$  is given graphically as



The representation function  $\beta_1$  is given as  $\beta_1(1) = u, \beta_1(2) = v, \beta_1(3) = w, \beta_1(4) = w$ . The concretization function is given by

$l$	$\perp$	$u$	$v$	$w$	$uv$	$vw$	$uw$	$uvw$
$\gamma_1(l)$	$\emptyset$	$\{1\}$	$\{2\}$	$\{3,4\}$	$\{1,2\}$	$\{2,3,4\}$	$\{1,3,4\}$	$D$

and the abstraction function is

$\emptyset$	$\perp$
$\{1\}$	$u$
$\{2\}$	$v$
$\{3\}$	$w$
$\{4\}$	$w$
$\{1,2\}$	$uv$
$\{1,3\}$	$uw$
$\{1,4\}$	$uw$
$\{2,3\}$	$vw$
$\{2,4\}$	$vw$
$\{3,4\}$	$w$
$\{1,2,3\}$	$uvw$
$\{1,2,4\}$	$uvw$
$\{1,3,4\}$	$uw$
$\{2,3,4\}$	$vw$
$D$	$uvw$

With  $\iota(a)$  we have a function  $D \rightarrow D$  and we want to find an interpretation  $\tilde{\iota}(a)$  on  $L_1$  that is conservative to  $\iota(a)$  and most precise. The formula we obtain from our earlier consideration is:

$$\tilde{\iota}(a)(l) = \alpha_1(\{\iota(a)(v) \mid \forall v \in \gamma_1(l)\})$$

for all  $l_1, \dots, l_n \in L$ . We obtain for the value  $w$  the image

$$\begin{aligned} \iota(a)(w) &= \alpha_1(\{\iota(a)(v) \mid \forall v \in \gamma_1(w)\}) \\ &= \alpha_1(\{\iota(a)(v) \mid \forall v \in \{3,4\}\}) \\ &= \alpha_1(\{3,4\}) = \bigsqcup \{v, w\} = vw \end{aligned}$$

and as a whole we get:

$l$	$\perp$	$u$	$v$	$w$	$uv$	$vw$	$uw$	$uvw$
$\tilde{\iota}(a)(l)$	$\perp$	$u$	$w$	$vw$	$uw$	$vw$	$uvw$	$uvw$

**Obtaining a Lattice from a Partition.** A partition is essentially a surjective function  $h \in D \rightarrow \tilde{D}$  where  $\tilde{D}$  is some set. The *canonical naming schema* from canonical abstraction computes a partition based on abstraction predicates. Given such a partition of  $D$ , we can choose the power set  $L = \mathcal{P}(\tilde{D})$  or augment  $\tilde{D}$  with a top and a bottom element,  $L = \tilde{D} \cup \{\perp, \top\}$ . In the previous example, the first choice corresponded to  $L_1$  and the second choice to  $L_2$ .

**Formalization.** We have the intuition and want to formalize it. First, we need a notion of abstract logical structures.

**Definition A.0.1 (Extended modal logical structure).**

Let  $\Sigma = \langle \mathcal{B}, \mathcal{F}, P, \mathcal{V}, r \rangle$  be a signature.

A modal logical structure is a tuple  $\langle U, \Delta, \iota \rangle$  such that

- universe.*  $U$  is a universe of values. The elements of the universe  $U$  are called individuals.
- semantic domains.*  $\Delta$  is a function which maps each base type  $T$  to a join-lattice (cf. 5.1.3)  $\Delta(T) \subseteq U$ . We write  $\sqsubseteq_T$  for the partial order of  $\Delta(T)$  and  $\bigsqcup_T$  for its join operator. The universe  $U = \bigcup_{T \in \mathcal{B}} \Delta(T)$  is the disjoint union of semantic domains.
- interpretation.* The interpretation  $\iota$  maps each function symbol  $f \in \mathcal{F}$  of rank  $r(f) = (T_1 \dots T_n, T)$  to a function

$$\iota(f) \in \Delta(T_1) \times \dots \times \Delta(T_n) \rightarrow \Delta(T) .$$

The interpretation  $\iota$  maps each predicate symbol  $p \in P$  of rank  $r(p) = (T_1 \dots T_n, \text{Bool})$  to a function

$$\iota(p) \in \Delta(T_1) \times \dots \times \Delta(T_n) \rightarrow \mathbb{K} .$$

The interpretations are monotone.

We denote the set of modal logical structures over  $\Sigma$  as  $MStruct[\Sigma]$ . Sometimes we omit the universe of a logical structure and write  $\langle \Delta, \iota \rangle$  since the universe is uniquely determined by  $U = \bigcup_{T \in \mathcal{B} \setminus \{\text{Bool}\}} \Delta(T)$ . Using this notation we define the set of logical structures over  $\Sigma$  with fixed  $\Delta$

$$MStruct[\Sigma, \Delta] = \{s \mid s = \langle \Delta, \iota \rangle \in MStruct[\Sigma]\} .$$

**Definition A.0.2 (Extended embedding).**

Let  $s = \langle U, \Delta, \iota \rangle \in Struct[\Sigma] \cup MStruct[\Sigma]$  and  $\tilde{s} = \langle W, \tilde{\Delta}, \tilde{\iota} \rangle \in MStruct[\Sigma]$ . We say that  $\beta \in U \rightarrow W$  embeds  $s$  into  $\tilde{s}$ , denoted as  $s \sqsubseteq_\beta \tilde{s}$ , iff

- $\beta$  decomposes into maps  $\beta_T = \beta|_{\Delta(T)} \in \Delta(T) \rightarrow \tilde{\Delta}(T)$ . We define concretization functions  $\gamma_T(l) = \{v \in D \mid \beta_T(v) \sqsubseteq l\}$  for each base type  $T$ .
- For every  $T$  there exists a subset  $K_T \subseteq \tilde{\Delta}(T)$  with pre-image  $\beta^{-1}(K_T) = \Delta(T)$  and  $\forall l \in \tilde{\Delta}(T) \exists k \in K_T : k \sqsubseteq l$ . We write  $\sqsubseteq_T$  for the partial order of  $\tilde{\Delta}(T)$  and  $\bigsqcup_T$  for its join operator.



- For each  $f \in \mathcal{F}$  with rank  $r(f) = (T_1 \dots T_n, T)$  holds:

$$\forall l \in \prod_{i=1}^n \tilde{\Delta}(T_i) \quad \forall u \in \prod_{i=1}^n \gamma_{T_i}(l_i) : \beta_T(\iota(f)(u)) \sqsubseteq_T \tilde{\iota}(f)(l)$$

- For each predicate symbol  $p \in P$  with rank  $r(p) = (T_1 \dots T_n, \text{Bool})$  holds:

$$\forall l \in \prod_{i=1}^n \tilde{\Delta}(T_i) \quad \forall u \in \prod_{i=1}^n \gamma_{T_i}(l_i) : \iota(p)(u) \sqsubseteq_{\mathbb{K}} \tilde{\iota}(p)(l)$$

where  $\sqsubseteq$  is the least upper bound operator of the Kleene domain  $\sqsubseteq_{\mathbb{K}}$ . We say that  $s$  can be embedded in  $\tilde{s}$ , denoted by  $s \sqsubseteq \tilde{s}$ , if there exists a function  $\beta$  such that  $s \sqsubseteq_{\beta} \tilde{s}$ .

**Theorem A.0.1 (Extended Embedding Theorem).**

If  $s \sqsubseteq_{\beta} \tilde{s}$  and  $e \in \mathcal{FO}_{\Sigma}$  then we have for every complete assignment  $Z$  that  $[e] s Z \sqsubseteq [e] \tilde{s} (\beta \circ Z)$ . An alternative formulation is that validity of  $e$  in  $\tilde{s}$  implies validity of  $e$  in  $s$ , i.e.,  $([e] \tilde{s} (\beta \circ Z)) = 1$  implies  $([e] s Z) = 1$ , and invalidity of  $e$  in  $\tilde{s}$  implies invalidity of  $e$  in  $s$ , i.e.,  $([e] \tilde{s} (\beta \circ Z)) = 0$  implies  $([e] s Z) = 0$ .

*Proof.* see B.4 □

**Definition A.0.3 (Extended Tight Embedding).** We say that  $\tilde{s} = \langle W, \tilde{\Delta}, \tilde{\iota} \rangle \in MStruct[\Sigma]$  is a tight embedding of  $s = \langle U, \Delta, \iota \rangle \in Struct[\Sigma] \cup MStruct[\Sigma]$  if there exists a function  $\beta \in U \rightarrow W$  such that

- $\beta$  decomposes into maps  $\beta_T = \beta|_{\Delta(T)} \in \Delta(T) \rightarrow \tilde{\Delta}(T)$ . We define concretization functions  $\gamma_T(l) = \{v \in D \mid \beta_T(v) \sqsubseteq l\}$  for each base type  $T$ .
- For every  $T$  there exists a subset  $K_T \subseteq \tilde{\Delta}(T)$  with pre-image  $\beta^{-1}(K_T) = \Delta(T)$  and  $\forall l \in \tilde{\Delta}(T) \exists k \in K_T : k \sqsubseteq l$ . We write  $\sqsubseteq_T$  for the partial order of  $\tilde{\Delta}(T)$  and  $\bigsqcup_T$  for its join operator.
- For each  $f \in \mathcal{F}$  with rank  $r(f) = (T_1 \dots T_n, T)$  holds:

$$\forall l \in \prod_{i=1}^n \tilde{\Delta}(T_i) : \tilde{\iota}(f)(l) = \bigsqcup_T \{\beta_T(\iota(f)(u)) \mid u \in \prod_{i=1}^n \gamma_{T_i}(l_i)\}$$

- For each  $p \in P$  with rank  $r(p) = (T_1 \dots T_n, \text{Bool})$  holds:

$$\forall l \in \prod_{i=1}^n \tilde{\Delta}(T_i) : \tilde{\iota}(p)(l) = \bigsqcup_T \{\iota(p)(u) \mid u \in \prod_{i=1}^n \gamma_{T_i}(l_i)\}$$

**Remark A.0.1 (Embedding and Tight Embedding).** If a function  $\beta$  tightly embeds a structure  $s$  into a structure  $\tilde{s}$  then  $s \sqsubseteq_{\beta} \tilde{s}$ .  $\tilde{s}$  is uniquely determined by  $s$ , a function  $\tilde{\Delta} \in \mathcal{B} \rightarrow W$  and the embedding function  $\beta$  if  $\tilde{s}$  tightly embeds  $s$ . We write  $\tilde{s} = \text{embed}_{\tilde{\Delta}, \beta}(s)$ .

# Appendix B

## Proofs

### B.1 Galois connection induced by representation function

**Definition B.1.1 (Galois connection (cf. [Cou96, CC77])).** Let  $\langle L, \sqsubseteq_L \rangle, \langle M, \sqsubseteq_M \rangle$  be complete lattices, and  $\alpha \in L \rightarrow M, \gamma \in M \rightarrow L$  total functions.  $\langle \alpha, \gamma \rangle$  is a Galois connection iff

$$\forall l \in L \forall m \in M : \alpha(l) \sqsubseteq_M m \Leftrightarrow l \sqsubseteq_L \gamma(m)$$

holds.

The following equivalences hold:

$$\begin{aligned} \alpha(D') \sqsubseteq l &\Leftrightarrow \bigsqcup \{ \beta(v) \mid v \in D' \} \sqsubseteq l \\ &\Leftrightarrow \forall v \in D' : \beta(v) \sqsubseteq l \\ &\Leftrightarrow D' \subseteq \gamma(l) . \end{aligned}$$

### B.2 Skolemization for predicate logic

The translation helps understand and prove Skolemization for the pure-predicate setting. Models denoted in predicate logic are special models of first-order logic. Skolemization as in Theorem 4.1.1 is hence applicable to them. However, the introduction of Skolem constants produces a model which is not formulated in predicate logic. We just translate this model back to predicate logic. The encoding which is mentioned (but not formalized) in the Observation below provides a more general encoding which can be used for this purpose.

**Observation B.2.1 (Encoding).** For every many-sorted model  $M$  there is a corresponding predicate model  $M^\sharp$  and for every formula  $\phi \in FCTL_\Sigma^*$  one can compute a formula  $\phi^\sharp$  such that

$$M \models \phi \Leftrightarrow M^\sharp \models \phi^\sharp.$$

*Proof.* The claim follows immediately from Theorem B.2.4 (Preservation Theorem for Encodings) and Lemma B.2.5 (Existence of Encodings).  $\square$

**Lemma B.2.1.** *The result of successively applying skolemization and then the Encoding within this proof produces predicate logic skolemization.*

In this section we will reduce many-sorted problems to one-sorted pure predicate problems. The rewrite method for expressions is quite generic.

Many-sorted logical structures are encoded as one-sorted pure-predicate logical structures. This encoding is then 'lifted' to a models over a state-space by means of a translation of many-sorted first-order expressions. But let us first deal with the encoding of structures. Pure predicate first-order logic lacks sorts and function symbols; these we will encode. We encode sorts as unary predicates and non-predicate  $n$ -place function symbols as  $n + 1$ -place predicates. That yields pure-predicate signatures. We give an injective encoding function from many-sorted to pure-predicate logical structures. Furthermore, we give a syntactic transformation function that maps many-sorted first-order expressions to equivalent first-order expression.

**Definition B.2.1.** *Let  $\Sigma = \langle \mathcal{B}, \mathcal{F}, P, \mathcal{V}, r \rangle$  be a many-sorted signature. We define the predicate signature  $\mathcal{P}_\Sigma = \langle P', \mathcal{V}, r \rangle$  consisting of the predicate symbols.*

$$\begin{aligned} P' &= P \\ &\dot{\cup} \{ p[f] \mid f \in \mathcal{F} \} \\ &\dot{\cup} \{ t[T] \mid T \in \mathcal{B} \} \end{aligned}$$

and the arity function  $r \in P' \rightarrow \mathbb{N}$  defined as

$$r(x) = \begin{cases} n + 1 & ; \quad x = p[f], \quad r^\Sigma(f) = (T_1 \dots T_n, T) \\ n & ; \quad x = p \in P \\ 1 & ; \quad x = t[T] \end{cases} .$$

**Definition B.2.2 (Encoding of a Structure).** *Let  $\Sigma$  be a many-sorted signature. We define the encoding function  $enc_\Sigma \in Struct[\Sigma] \mapsto Struct[\mathcal{P}_\Sigma]$  as follows*

$$\langle U, \Delta, \iota_{many} \rangle \rightarrow \langle U, \iota_{one} \rangle$$

where for each predicate  $f \in \mathcal{F}$  with  $r(f) = n$  we set for all  $u_1, \dots, u_n \in U$

$$\iota_{one}(f)(u_1, \dots, u_n) = \begin{cases} \iota_{many}(f)(u_1, \dots, u_n) & ; \quad (u_1, \dots, u_n) \in dom(\iota_{many}(f)) \\ 0 & ; \quad otherwise \end{cases}$$

and for each non-predicate function symbol  $f \in \mathcal{F}$  of arity  $r(p[f]) = n + 1$  and all  $u_1, \dots, u_n, u \in U$

$$\iota_{one}(p[f])(u_1, \dots, u_n, u) = \begin{cases} \iota_{many}(f)(u_1, \dots, u_n) = u & ; \quad (u_1, \dots, u_n) \in dom(\iota_{many}(f)) \\ 0 & ; \quad otherwise \end{cases} .$$

For each sort  $T$  we set  $\forall u \in U : \iota_{one}(t[T])(u) \Leftrightarrow u \in \Delta(T)$ .

We show a technical lemma:  $enc_\Sigma$  is one-to-one.

**Lemma B.2.2.**  $enc_\Sigma$  is injective.

**Example B.2.1.** Let us assume that our signature contains the usual symbols and types for reasoning about natural numbers: addition  $add : Nat \times Nat \rightarrow Nat$ , incrementation  $succ : Nat \rightarrow Nat$  and equality  $eq : Nat \times Nat \rightarrow Nat$ . The first-order formula that expresses that every natural number  $n$  has a successor  $m$ ,  $\forall n : Nat. \exists m : Nat. eq(m, succ(n))$  is translated to  $\forall n. \exists m. t[Nat](n) \wedge t[Nat](m) \wedge \exists v_{succ}. p[succ](n, v_{succ}) \wedge eq(m, v_{succ})$ .

The algorithm below<sup>1</sup> computes such a translation function.

**Definition B.2.3 (Translation Algorithm).** Let  $\Sigma$  be a signature. The translation procedure  $Trans \in \mathcal{FO}_\Sigma \rightarrow \mathcal{FO}_{\mathcal{P}_\Sigma}$  for first-order expressions uses an auxiliary procedure for translating terms

$$Trans(e) = \begin{cases} TransTerm(t) & ; e = t \\ c & ; c \in \mathbb{B} \\ Trans(e_1) \wedge Trans(e_2) & ; e = e_1 \wedge e_2 \\ \neg Trans(e') & ; e = \neg e' \\ \exists x. t[T](x) \wedge Trans(e') & ; \exists x : T. e' \\ (TCv_1, v_2. Trans(e'))(v_3, v_4) & ; e = (TCv_1, v_2 : T.e')(v_3, v_4) \end{cases}$$

The translation procedure  $TransTerm \in Term_\Sigma \rightarrow \mathcal{FO}_{\mathcal{P}_\Sigma}$  for terms is given below:

```

TransTerm(t) = e := t
  QV := ∅
  while ( there is a subterm  $t = f(v_1, \dots, v_n)$  of  $e$ 
    where  $f \in \mathcal{F}$  is a non-predicate function symbol )
    pick a fresh variable  $v$ 
    bind := bind  $\wedge p[f](v_1, \dots, v_n, v)$ 
    QV := QV  $\cup \{v\}$ 
     $e := e[v/t]$  , i.e., substitute  $v$  for  $t$  in  $e$ 
  return  $\exists x_1. \dots \exists x_n. bind \wedge e$ 
  where  $QV = \{x_1, \dots, x_n\}$ 

```

where the set  $QV$  are the newly created variables.

**Lemma B.2.3.** The algorithm in B.2.3 defines a function  $tr_\Sigma = \lambda e. Trans(e)$ . The translation function  $tr_\Sigma$  is correct, i.e.,

$$\forall e \in \mathcal{FO}_\Sigma \forall s \in Struct[\Sigma] \forall Z : s, Z \models e \Leftrightarrow enc_\Sigma(s), Z \models tr_\Sigma(e) .$$

*Proof.* We give a plausibility argument here. The function  $Trans$  itself is straightforward.  $TransTerm$  is more interesting. The interpretation of predicates  $p[f]$  are functions, that is if  $u_1, \dots, u_n$  are individuals  $p[f](u_1, \dots, u_n, u)$  uniquely determines  $u$ . In  $TransTerm$  we insert the variable that refers to the value of  $f(u_1, \dots, u_n)$  in place of  $f(u_1, \dots, u_n)$ .  $\square$

<sup>1</sup>An equivalent deterministic recursive linear-time algorithm can be implemented.

**Definition B.2.4 (Encoding of a Model).** Let  $M = \langle S, \theta, \rho \rangle$  many-sorted model. A pure-predicate model  $M^\# = \langle S^\#, \theta^\#, \rho^\# \rangle$  is an encoding of  $M$  iff:

$$\forall s^\#, t^\# \in S^\# : \quad \begin{array}{l} \llbracket \rho^\# \rrbracket_{S^\#}(s^\#, t^\#) \Leftrightarrow \text{enc}_\Sigma(s) = s^\# \wedge \text{enc}_\Sigma(t) = t^\# \wedge \llbracket \rho \rrbracket_S(s, t) \\ \llbracket \theta^\# \rrbracket_{S^\#}(s^\#) \Leftrightarrow \text{enc}_\Sigma(s) = s^\# \wedge \llbracket \theta \rrbracket_S(s) \end{array}$$

and  $\text{enc}_\Sigma(S) \subseteq S^\#$ .

We extend  $\text{tr}_\Sigma$  to temporal formulas.  $\text{tr}_\Sigma(\phi)$  for a temporal formula  $\phi$  is  $\phi$  with each first-order expression  $e$  replaced with  $\text{tr}_\Sigma(e)$ .

**Theorem B.2.4 (Equivalence Theorem).** Let  $M = \langle S, \theta, \rho \rangle$  be many-sorted model and  $M^\# = \langle S^\#, \theta^\#, \rho^\# \rangle$  an encoding of  $M$ . Then for every first-order temporal formula  $\phi$  the equivalences  $\forall s \in S : s \models \phi \Leftrightarrow \text{enc}_\Sigma(s) \models \text{tr}_\Sigma(\phi)$  and  $M \models \phi \Leftrightarrow M^\# \models \text{tr}_\Sigma(\phi)$  hold.

**Lemma B.2.5 (Existence of Encodings).** Let  $M = \langle S, \theta, \rho \rangle$  be a many-sorted model. The pure-predicate model  $M^\# = \langle \text{enc}_\Sigma(S), \theta^\#, \rho^\# \rangle$  with

$$\rho^\# = \text{tr}_{\Sigma \cup \Sigma'}(\rho)$$

and

$$\theta^\# = \text{tr}_\Sigma(\theta)$$

is an encoding of  $M$ .

**Example B.2.2.** This is taken from the Ticket Protocol Example that follows later:

$$\forall i_2. p[t'](i_2) \equiv \exists i_1. p[t](i_1) \wedge p[\text{succ}](i_1, i_2)$$

and corresponds to  $t'() := \text{succ}(t())$ . The right-hand side of the  $\equiv$  is exactly  $\text{tr}_\Sigma(\text{succ}(t()))$ . So, even in restricted cases  $\text{tr}_\Sigma$  can be used for the right-hand sides of assignments  $:=$ .

## B.3 Preservation by Simulation

**Claim.** Let  $\phi$  be an ACTL\* state formula and  $K, K'$  two transition systems such that  $K \preceq K'$ . Then  $K' \models \Phi \Rightarrow K \models \Phi$ .

*Proof.* Let  $K = \langle S, I, R \rangle$ ,  $K' = \langle S', I', R' \rangle$  be transition systems such that  $K \preceq K'$  and  $H$  is a simulation between  $K$  and  $K'$ .

**Definition B.3.1.** Two paths  $\pi \in \Pi_K$ ,  $\pi' \in \Pi_{K'}$  correspond, denoted as  $\pi \sim_H \pi'$ , iff  $\forall i \in \mathbb{N} : H(\pi_i, \pi'_i)$ .

**Lemma B.3.1.** For every pair of states  $(s, s') \in H$ , holds that for every path in  $K$  that starts with  $s$  there exists a corresponding path in  $K'$  that starts with  $s'$ , i.e.,

$$\forall (s, s') \in H \forall \pi \in \Pi_K : \pi_0 = s \Rightarrow \exists \pi' \in \Pi_{K'} : \pi'_0 \wedge \pi \sim_H \pi' .$$

*Proof.* Let  $(s, s') \in H$  and  $\pi \in \Pi_K$  such that  $\pi_0 = s$ . We exploit the fact that  $H$  is a simulation relation and inductively construct a path that corresponds to  $\pi$ . We show that there is a family of maps  $(m^i)_{i \in \mathbb{N}}$  such that

- (i)  $m^i$  is a finite path of length  $i$  in  $K'$ , i.e.,  $m^i \in \{0, \dots, i\} \rightarrow S'$  and  $\forall k \in \{0, \dots, i-1\} : R'(m^i(k), m^i(k+1))$
- (ii)  $m^{i+1}$  is prolongation of  $m^i$ , i.e.,  $m^{i+1}|_{\{0, \dots, i\}} = m^i$  for all  $i$ .
- (iii)  $\forall i \in \mathbb{N} \forall k \in \mathbb{N} : H(\pi_k, m^i(k))$ .

We can then define  $\pi' = \lambda k \in \mathbb{N} : m^i(k)$  and  $\pi' \in \Pi_{K'}$ . Let us turn to the induction. The base case is simple. Since  $R(\pi_0, \pi_1)$  there exists a  $t'$  such that  $R'(s', t')$  and  $H(\pi_1, t')$ . We set  $m^0 = t'$ . Let us assume that  $m^0, \dots, m^j$  have been constructed. Since  $H(\pi_j, m^j(j))$  there exists exists a  $t''$  such that  $R'(m^j(j), t'')$  and  $H(\pi_1, t'')$ . We set

$$m^{j+1} = \lambda k \in \{0, \dots, j+1\} : \begin{cases} t'' & ; \quad k = j+1 \\ m^j(k) & ; \quad \text{otherwise} \end{cases} .$$

Apparently, criteria (i)-(iii) hold. This concludes our proof.  $\square$

Statement (I) of the following lemma is exactly the claim in Theorem 5.1.2. We use Lemma B.3.1 to prove it.

**Lemma B.3.2 (Preservation Lemma).** *The following two claims hold:*

- Let  $\phi$  be an ACTL\* state formula and  $(s, s') \in H$  then  $s' \models \phi \Rightarrow s \models \phi$ . (I)
- Let  $\Phi$  be an ACTL\* state formula and  $\pi \sim_H \pi'$  then  $\pi' \models \Phi \Rightarrow \pi \models \Phi$ . (II)

*Proof.* We prove (I) and (II) by simultaneous structural induction on state and path formulas.

$$\begin{aligned} \phi & ::= e \mid \neg e \mid \phi \wedge \phi \mid \phi \vee \phi \mid \mathbf{A}\Phi && \text{(state)} \\ \Phi & ::= \phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \mathbf{X}\Phi \mid \Phi \mathbf{U}\Phi \mid \Phi \mathbf{R}\Phi && \text{(path)} \end{aligned}$$

We assume (I) to prove (II) and vice versa. This is well-founded since we always decrease the size of the formulas. One could give a correctness argument directly using the Knaster-Tarski Fixpoint Theorem. We omit that.

**state formulas  $\phi$ :** Let  $(s, s') \in H$ . Assume that  $s' \models \phi$ .

$\phi = e$ : The claim holds by definition of simulation.

$\phi = \neg e$ : The claim holds by definition of simulation, since  $\neg e$  is an expression, too (or, at least, has the same semantics).

$\phi = \phi_1 \wedge \phi_2$ : We assume that  $s' \models \phi$ , i.e.,  $s'$  fulfills  $\phi_1$  and  $\phi_2$ , then by induction hypothesis  $s$  also fulfills both, and therefore  $s \models \phi$ .

$\phi = \phi_1 \vee \phi_2$ : We assume that  $s' \models \phi$ , i.e.,  $s'$  fulfills either of  $\phi_1$  and  $\phi_2$ . If  $s'$  fulfills  $\phi_1$ , then by induction hypothesis  $s$  also fulfills  $\phi_1$ . If  $s'$  fulfills either  $\phi_2$  by induction hypothesis  $s$  also fulfills  $\phi_2$ . Hence we have  $s \models \phi$ .

$\phi = \mathbf{A}\Phi$ : Lemma B.3.1 comes into play.  $s' \models \phi$ , i.e., every path starting in  $s'$  fulfills  $\Phi$ . By Lemma B.3.1 for every path starting in  $s$  there exists a corresponding path starting in  $s'$ . Let  $\pi$  be a path starting in  $s$ . There exists a corresponding path  $\pi'$  starting in  $s'$ . By assumption  $\pi' \models \Phi$  and by induction hypothesis  $\pi \models \Phi$ . Every path starting in  $s$  fulfills  $\Phi$ . Therefore,  $s \models \phi$ .

**path formulas**  $\phi$ : Let  $\pi \sim_H \pi'$ . Assume that  $\pi' \models \phi$ .

$\Phi = \phi$ : Since  $\pi' \models \Phi$  holds, we get  $\pi'_0 \models \phi$  and by induction hypothesis  $\pi_0 \models \phi$ . So  $\pi \models \Phi$  holds also.

$\Phi = \Phi_1 \wedge \Phi_2$ : We assume that  $\pi' \models \Phi$ , i.e.,  $\pi'$  fulfills  $\Phi_1$  and  $\Phi_2$ , then by induction hypothesis  $\pi$  also fulfills both, and therefore  $\pi \models \Phi$ .

$\Phi = \Phi_1 \vee \Phi_2$ : We assume that  $\pi' \models \Phi$ , i.e.,  $\pi'$  fulfills either of  $\Phi_1$  and  $\Phi_2$ . If  $\pi'$  fulfills  $\Phi_1$ , then by induction hypothesis  $\pi$  also fulfills  $\Phi_1$ . If  $\pi'$  fulfills either  $\Phi_2$  by induction hypothesis  $\pi$  also fulfills  $\Phi_2$ . Hence we have  $\pi \models \Phi$ .

$\Phi = \mathbf{X}\Phi'$ : The postfix of  $\pi'$  fulfills  $\Phi'$ , i.e.,  $\pi'^1 \models \Phi'$ . The postfix  $\pi^1$  of  $\pi$  corresponds to  $\pi'^1$ , i.e.,  $\pi^1 \sim_H \pi'^1$ . Therefore, by induction hypothesis,  $\pi^1 \models \Phi'$  and  $\pi \models \Phi$ .

$\Phi = \Phi_1 \mathbf{U} \Phi_2$ : We have that  $\pi' \models \Phi_1 \mathbf{U} \Phi_2$ , which means that there exists a  $k \in \mathbb{N}$  such that

$$\pi'^k \models \Phi_2 \wedge \forall 0 \leq j \leq k : \pi'^j \models \Phi_1 .$$

We have that  $\pi^k \sim_H \pi'^k$  and  $\forall 0 \leq j \leq k : \pi^j \sim_H \pi'^j$ . By induction hypothesis

$$\pi^k \models \Phi_2 \wedge \forall 0 \leq j \leq k : \pi^j \models \Phi_1$$

holds, which proves the claim.

$\Phi = \Phi_1 \mathbf{R} \Phi_2$ :  $\pi' \models \Phi_1 \mathbf{R} \Phi_2$  which means that

$$\forall j \in \mathbb{N} : (\forall i < j : \pi'^i \not\models \Phi_1) \Rightarrow \pi'^j \models \Phi_2$$

holds. Again we get the correspondences  $\forall j \in \mathbb{N} : (\forall i < j : \pi^i \sim_H \pi'^i \wedge \pi^j \sim_H \pi'^j)$  and apply the induction hypothesis:

$$\forall j \in \mathbb{N} : (\forall i < j : \pi^i \not\models \Phi_1) \Rightarrow \pi^j \models \Phi_2$$

which proves the claim. □

□

## B.4 Extended Embedding

*Proof.* It is easy to see that since the interpretation of function symbols in a modal logical structure is monotone, the interpretation of expressions  $e \in \mathcal{FO}_\Sigma$  is monotone as well, i.e.,

$$\forall \tilde{s} \in MStruct[\Sigma] : (\forall x \in V : Z(x) \sqsubseteq Z'(x)) \Rightarrow [e] \tilde{s} Z \sqsubseteq [e] \tilde{s} Z'$$

where  $Z, Z'$  are complete assignments for  $e$ .

$s, \tilde{s}$  and  $\beta$  be defined as in the Theorem. For every term  $t$  we prove

$$\beta([t] s Z) \sqsubseteq [t] \tilde{s} (\beta \circ Z) \quad (*)$$

for every complete assignment  $Z$  and use this as a lemma to prove

$$[e] s Z \sqsubseteq [e] \tilde{s} (\beta \circ Z) \quad (**)$$

for every complete assignment  $Z$ .

We give proofs by structural induction

$$\begin{aligned} t & ::= x \mid f(t_1, \dots, t_n) && \text{(terms)} \\ e & ::= t \mid e \wedge e \mid \neg e \mid \forall x : T. e \mid TC(v_1, v_2 : T.e)(v_3, v_4) && \text{(expressions)} \end{aligned}$$

We begin with (\*):

The base case is the case where the term is a variable  $t = x$ . By definition we have :  $\beta([x] s Z) = \beta(Z(x)) = \beta([x] \tilde{s} (\beta \circ Z))$  and the claim follows because  $\sqsubseteq$  is reflexive.

The inductive case is  $t = f(t_1, \dots, t_n)$ . We make use of the induction hypothesis and our assumption:

$$\begin{aligned} \beta([e] s Z) & = \beta(\iota^s(f)([t_1] s Z, \dots, [t_n] s Z)) \\ & \sqsubseteq \iota^{\tilde{s}}(f)([t_1] \tilde{s} (\beta \circ Z), \dots, [t_n] \tilde{s} (\beta \circ Z)) \quad . \end{aligned}$$

Proving (\*\*) is a bit more involved because of quantification and transitive closure.

**Term:**  $e = t$ . The claim follows from the fact that  $\beta(0) = 0, \beta(1) = 1$  and (\*).

**Conjunction:**  $e = e_1 \wedge e_2$ . We apply the induction hypothesis:

$$\begin{aligned} [e] s Z & = \min([e_1] s Z, [e_2] s Z) \\ & \sqsubseteq \min([e_1] \tilde{s} Z, [e_2] \tilde{s} Z) \quad . \end{aligned}$$

**Negation:**  $e = \neg e'$ . We apply the induction hypothesis:

$$\begin{aligned} [e] s Z & = 1 - [e'] s Z \\ & \sqsubseteq 1 - [e'] \tilde{s} Z \quad . \end{aligned}$$

**Universal quantification:**  $e = \forall x : T.e'$ . There are two interesting cases  $[e] \tilde{s} (\beta \circ Z) = 1$  and  $[e] \tilde{s} (\beta \circ Z) = 0$ . If  $[e] \tilde{s} (\beta \circ Z) = 1/2$  nothing is to be shown.

**case:**  $[e] \tilde{s} (\beta \circ Z) = 1$ . This means that for all  $\tilde{v} \in \tilde{\Delta}^{\tilde{s}}(T)$  holds  $([e'] \tilde{s} ((\beta \circ Z) \cup \{x \mapsto \tilde{v}\})) = 1$ . Since by assumption there is a subset of  $\tilde{\Delta}^{\tilde{s}}(T)$  with pre-image  $\Delta^s(T)$  and because of the inductive assumption the claim follows.



**case:**  $[e] \tilde{s} (\beta \circ Z) = 0$ . This means that there exists a  $\tilde{v} \in \Delta^{\tilde{s}}(T)$  such that  $([e'] \tilde{s} ((\beta \circ Z) \cup \{x \mapsto \tilde{v}\})) = 0$ . Without loss of generality we can assume that  $\tilde{v} \in K_T$  (otherwise we pick a  $K_T \ni \tilde{w} \ni \tilde{v}$  and by monotonicity we have  $([e'] \tilde{s} ((\beta \circ Z) \cup \{x \mapsto \tilde{w}\})) \sqsubseteq ([e'] \tilde{s} ((\beta \circ Z) \cup \{x \mapsto \tilde{v}\})) = 0$ ). There is a value  $v \in \Delta^s(T)$  such that  $\beta(v) = \tilde{v}$  and by induction hypothesis we have:

$$\begin{aligned} [e] \tilde{s} (Z \cup \{x \mapsto v\}) &\sqsubseteq ([e'] \tilde{s} ((\beta \circ (Z \cup \{x \mapsto v\}))) \\ &= ([e'] \tilde{s} ((\beta \circ Z) \cup \{x \mapsto \tilde{v}\})) = 0 \end{aligned}$$

which proves our claim.

**Transitive Closure:**  $e = TC(v_1, v_2 : T.e')(v_3, v_4)$ .

**case:**  $[e] \tilde{s} (\beta \circ Z) = 1$ . There exist  $\tilde{u}_1, \dots, \tilde{u}_{n+1} \in \Delta(T)$  such that for all  $1 \leq i \leq n$   $[e']$  we have  $\tilde{s} ((\beta \circ Z) \cup \{v_1 \mapsto \tilde{u}_i, v_2 \mapsto \tilde{u}_{i+1}\})$ ,  $\beta \circ Z(v_3) = \tilde{u}_1$ , and  $\beta \circ Z(v_4) = \tilde{u}_{n+1}$ . Without loss of generality  $\tilde{u}_1, \dots, \tilde{u}_{n+1} \in K_T$  (otherwise we can choose values from  $K_T$  which fulfill the aforesaid three conditions by monotonicity). Because the pre-image of  $K_T$  under  $\beta$  is  $\Delta(T)$  there exist  $u_1, \dots, u_{n+1} \in \Delta(T)$  such that  $\beta(u_i) = \tilde{u}_i$  for all  $1 \leq i \leq n+1$ . Therefore,  $Z(v_3) = u_1, Z(v_4) = u_{n+1}$  and by induction hypothesis, for all  $1 \leq i \leq n$ ,

$$\begin{aligned} [e'] s (Z \cup \{v_1 \mapsto u_i, v_2 \mapsto u_{i+1}\}) &\sqsubseteq [e'] s (\beta \circ (Z \cup \{v_1 \mapsto u_i, v_2 \mapsto u_{i+1}\})) \\ &= [e'] s ((\beta \circ Z) \cup \{v_1 \mapsto \tilde{u}_i, v_2 \mapsto \tilde{u}_{i+1}\}) = 1 . \end{aligned}$$

**case:**  $[e] \tilde{s} (\beta \circ Z) = 0$ . Assume that  $[e] s Z \neq 0$ . Then there exist  $u_1, \dots, u_{n+1} \in \Delta(T)$  such that  $Z(v_3) = u_1, Z(v_4) = u_{n+1}$ , and for all  $1 \leq i \leq n$ ,

$$[e'] s (Z \cup \{v_1 \mapsto u_i, v_2 \mapsto u_{i+1}\}) \neq 0 .$$

Hence, by the induction hypothesis, there exist  $\tilde{u}_1, \dots, \tilde{u}_{n+1} \in \tilde{\Delta}(T)$  such that  $\beta \circ Z(v_3) = \tilde{u}_1, \beta \circ Z(v_4) = \tilde{u}_{n+1}$  and for all  $1 \leq i \leq n$ ,  $[e'] \tilde{s} (\beta \circ Z) \neq 0$ . Therefore,  $[TC(v_1, v_2 : T.e')(v_3, v_4)] \tilde{s} (\beta \circ Z) \neq 0$  holds which is a contradiction.  $\square$

## B.5 Symmetry Lemma

*Proof.*

**Claim (a):**

Apparently, the inverse  $f^{-1}$  of  $f$  is a bijection too. We choose  $e \in FO_{\mathcal{P}}$  and  $Z'$ . Let  $Z := f^{-1} \circ Z'$ . Hence

$$\begin{aligned} \llbracket e \rrbracket(s)(f^{-1} \circ Z') &= \llbracket e \rrbracket(s)(Z) \\ &= \llbracket e \rrbracket(s')(f \circ f^{-1} \circ Z) \\ &= \llbracket e \rrbracket(s')(Z') . \end{aligned}$$

**Claim (b):**

Let  $\tilde{s} := \alpha_{can}^{\mathcal{P}, \mathcal{A}}(s), \tilde{s}' := \alpha_{can}^{\mathcal{P}, \mathcal{A}}(s')$ . First, we show that  $U^{\tilde{s}} = U^{\tilde{s}'}$ . It suffices to prove the inclusion  $U^{\tilde{s}} \subseteq U^{\tilde{s}'}$  since the other direction is analogous due to (a).

The function that computes canonical names with respect to  $\mathcal{A} := \{a_1, \dots, a_n\}$ .

$$\kappa_{\mathcal{A},s} \in U \rightarrow \mathbf{K}^{|\mathcal{A}|}, u \mapsto \langle \iota^s(a_1)(u), \dots, \iota^s(a_k)(u) \rangle .$$

Let  $\vec{x} \in U^{\tilde{s}} = \kappa_{\mathcal{A},s}(U^s)$ . We need to prove  $\vec{x} \in U^{\tilde{s}'}$ . There exists  $u \in U^s$  such that  $\kappa_{\mathcal{A},s}(u) = \vec{x}$  has canonical name  $\vec{x}$ .

Now, we exploit the symmetry:

$$\forall k \in \mathbf{N} \quad \forall u_1, \dots, u_k \in U^s \quad \forall p \in \mathcal{P}_k \quad \forall \nu_1, \dots, \nu_k \in Var :$$

$$\begin{aligned} \iota^s(p)(u_1, \dots, u_k) &= \llbracket p \rrbracket(\nu_1, \dots, \nu_k) \text{ s } \{\nu_i \mapsto u_i\} \\ &= \llbracket p \rrbracket(\nu_1, \dots, \nu_k) \text{ s}' \{\nu_i \mapsto f(u_i)\} \\ &= \iota^{s'}(p)(f(u_1), \dots, f(u_k)) \end{aligned} \quad (I)$$

which yields

$$\begin{aligned} \kappa_{\mathcal{A},s}(u) = \vec{x} &= \iota^s(a_1)(u), \dots, \iota^s(a_k)(u) \\ &= \iota^{s'}(a_1)(f(u), \dots, \iota^{s'}(a_k)(f(u))) = \kappa_{\mathcal{A},s'}(f(u)) \in U^{\tilde{s}'} \end{aligned} \quad (II) .$$

It remains to prove that  $\tilde{s} = \tilde{s}'$ . Again we use (I):

$$\begin{aligned} \forall p \in \mathcal{P}_k : \quad \tilde{s}(p)(\tilde{u}_1, \dots, \tilde{u}_k) &= \bigsqcup_{u_i \in U^s, \kappa_{\mathcal{A},s}(u_i) = \tilde{u}_i} \iota^s(p)(u_1, \dots, u_k) && \text{s. Def. 5.1.7} \\ &= \bigsqcup_{u_i \in U^s, \kappa_{\mathcal{A},s}(u_i) = \tilde{u}_i} \iota^s(p)(f(u_1), \dots, f(u_k)) && (I) \\ &= \bigsqcup_{u_i \in U^s, \kappa_{\mathcal{A},s'}(f(u_i)) = \tilde{u}_i} \iota^s(p)(f(u_1), \dots, f(u_k)) && (II) \\ &= \bigsqcup_{u'_i \in U^{s'}, \kappa_{\mathcal{A},s'}(u'_i) = \tilde{u}_i} \iota^s(p)(u'_1, \dots, u'_k) && f \text{ surjective} \\ &= \tilde{s}'(p)(\tilde{u}_1, \dots, \tilde{u}_k) . && \text{s. Def. 5.1.7} \end{aligned}$$

The equality for the summary predicate  $sm$  can be shown analogously :

$$\begin{aligned} \iota^{s'}(sm)(\tilde{u}) &= (|\{u \mid \kappa_{\mathcal{A},s}(u) = \tilde{u}\}| > 1) \sqcup \bigsqcup_{u \in U, \kappa_{\mathcal{A},s}(u) = \tilde{u}} \iota^s(sm)(u) \\ &\stackrel{(I),(II)}{=} (|\{u \mid \kappa_{\mathcal{A},s'}(f(u)) = \tilde{u}\}| > 1) \sqcup \bigsqcup_{u \in U, \kappa_{\mathcal{A},s'}(f(u)) = \tilde{u}} \iota^s(sm)(f(u)) \\ &\stackrel{f \text{ surj.}}{=} (|\{u' \mid \kappa_{\mathcal{A},s'}(u') = \tilde{u}\}| > 1) \sqcup \bigsqcup_{u' \in U^{s'}, \kappa_{\mathcal{A},s'}(u') = \tilde{u}} \iota^s(sm)(u') \\ &= \tilde{s}'(sm)(\tilde{u}) \end{aligned}$$

This concludes the proof of Claim (b).

**Claim (c):**

$f$  is bijective, hence the joins  $\bigsqcup$  collapse and we can use (I).

**Claim (d):**

Let us assume that  $s \in Struct[\mathcal{P}]$  is a 2-valued structure.

$[s]_{\sim} \subseteq Struct[\mathcal{P}]$  holds because of the definition of symmetry, elements of  $s$ 's symmetry class cannot be 3-valued.

Let  $t \in [s]_{\sim}$ . We have the tight embeddings  $t \sqsubseteq s$ , because of (a) and (c). So there is a chain of embeddings  $t \sqsubseteq s \sqsubseteq \alpha_{can}^{\mathcal{P},\mathcal{A}}(s)$  and hence  $t \in \gamma_3(\alpha_{can}^{\mathcal{P},\mathcal{A}}(s))$  by definition of  $\gamma_3$ .  $\square$

# Appendix C

## Sources

This chapter contains some selected source files from the TVLA and the SMV case studies.

### C.1 Case Study with TVLA

The file which contains the "transition relation".

```
/* typing predicate */
%p number (integer) abs

/* successor relation */
%p succ(integer,integer) function acyclic

/* the value zero is needed because it is used for initialization */
%p zero(integer) abs

/* constraints on successor relation */
/* 1. (unique successors)
   "function" means there is at most one successor */

/* 2. (existence of successors)
   each number has a successor (and the successor is not zero) */
%r !(A(i) number(i) -> E (j) number(j) & !zero(j) & succ(i,j)) ==> 0

/* global counter variables */
%p s(integer) abs unique
%p t(integer) abs unique

/* transitive closure of succ */
%i t[succ] (n1,n2)=number(n1)&number(n2)&succ*(n1,n2)
/* reflexive*/
%r number(n1)&number(n2)&n1==n2 ==> t[succ] (n1,n2)
/* transitive */
%r number(n1)&number(n2)&number(n3)&t[succ] (n1,n2)&t[succ] (n2,n3)
==> t[succ] (n1,n3)
```

```

/* s \leq n2 */
%i r[s,succ](n2)=E(n1) s(n1) & t[succ](n1,n2)
/* t \leq n2 */
%i r[t,succ](n2)=E(n1) t(n1) & t[succ](n1,n2)

%p process(p) abs

/* local counter variables */
%p a(process,integer) function

/* these are variables not pointers, so there is always
   something s,t,a point to */
/* %r process(p)& !E(i) a(p,i) ==> 0 */

/* %r !(E(i) s(i)) ==> 0
   %r !(E(i) t(i)) ==> 0 */

%p p_1 (process) unique abs
%p p_2 (process) unique abs
%p initialized ()

/* instrument the things connected to p_1 and p_2 */
/*
%i a_p_1(integer)=E(p) p_1(p) & a(p,integer) abs
%i a_p_2(integer)=E(p) p_2(p) & a(p,integer) abs
*/

/* Skolemization constraints */

%r ( (A (p) process(p)-> !p_1(p)) |
      (A (p) process(p)-> !p_2(p)) |
      (A (p) process(p)-> !sm(p))) ==> 0

%r (p_1(p) & p_2(q)) ==> !p==q

/* We may assume our result from the lemma
   (s. ticket_lemma)

   No two processes have the same ticket !
*/

%r (E(p1,p2,i) p1!=p2 & !at[think](p1) & !at[think](p2)
      & a(p1,i) & a(p2,i) ) ==> 0

%%

%action init () {
  %f { p_1(p), p_2(p)}

```

```

%p !initialized()
{
    initialized() = 1
}
}

%action draw_ticket() {
    %t "a:=t; t:=t+1"
    %f {runnable(tr)&a(tr,u), E(n1, n2) t(n1) & succ(n1, n2)
        & t[succ](n2, n)}

    %p initialized()
    {
        /* core predicates */
        a(process,integer) = (process==tr ? t(integer): a(process,integer))
/*
        a_p_1(i) = (p_1(tr) ? t(i) : a_p_1(i))
        a_p_2(i) = (p_2(tr) ? t(i) : a_p_2(i))
*/
        t(i2) = E (i1) succ(i1,i2) & t(i1)

        /* instrumentation predicates */
        r[t,succ](n)= !t(n) & r[t,succ](n)
    }
}

%action myturn() {
    %t "when a=s do"
    %f {runnable(tr)&a(tr,u), s(x)}
    %p initialized() & E(i) a(tr,i) & s(i)
    {
    }
}

%action increase() {
    %t "s:=s+1"
    %f { runnable(tr), E(n1, n2) s(n1) & succ(n1, n2) & t[succ](n2, n)}
    %p initialized()
    {
        s(i2) = E (i1) succ(i1,i2) & s(i1)

        /* instrumentation predicates */
        r[s,succ](n)= !s(n) & r[s,succ](n)
    }
}

%action verifyProperty() {
    %message (E (u,v) !u==v & p_1(u) & p_2(v) & at[crit](u) & at[crit](v))
    -> "violation!!!!"
}

```

```

}

%%

%thread bigbrother {
  notyet init() now
}

%thread pr {
  think  draw_ticket() wait
  wait   myturn()        crit
  crit   increase()      think
}

%%

verifyProperty()

```

The file which describes the initial states.

```

%t = { p(pr), b(bigbrother) }
%n = { intz, intgz }
%p = {
  ready= {b,p}
  at[think] = {p}
  sm      = {p:1/2,intgz:1/2}
  number = {intz, intgz}
  process = {p}
  zero   = {intz}
  s      = {intz}
  t      = {intz}
  succ   = {(intz,intgz): 1/2, (intgz,intgz) : 1/2}
  t[succ] = {intz->intz,intz->intgz,intgz->intgz:1/2}
  r[s,succ] = {intz,intgz}
  r[t,succ] = {intz,intgz}
  a       = {(p,intz)}

  p_1 = { p:1/2}
  p_2 = { p:1/2}
  /*  a_p_1 = {intz}
     a_p_2 = {intz} */

  initialized = 0
}

```

## C.2 Case Study with SMV

The following file is the SMV code of the Ticket Protocol.

```

scalarset PROC undefined;
ordset TICKET 0..;
typedef LOC {think,wait,crit};

module main() {
  s : TICKET;
  t : TICKET;
  a : array PROC of TICKET;
  at: array PROC of LOC;
  act:PROC;

  init(s):=0;
  init(t):=0;

  forall (p in PROC){
    init(at[p]):=think;
    init(a[p]):=0;
  }
  /* the code of a process */
  switch(at[act]){
    think: { next(a[act]):=t;
             next(t):=t+1;
             next(at[act]):=wait;
           }
    wait: {  if(a[act]=s) {
              next(at[act]):=crit;
            }
           }
    crit: {  next(s):=s+1;
             next(at[act]):=think;
           }
  }

  /* MUTEX */
  forall(q in PROC)
  forall(i in TICKET)
  forall (p in PROC)
  {
    uneq [p][q][i]: assert
    G(!(p=q)& !at[p]=think & !at[q]=think -> !( a[p]=i & a[q]=i));
  }

  forall (p in PROC) forall(i in TICKET){
    remain[p][i]: assert G( (at[p]=crit&a[p]=i) -> (s=i)U(at[p]=think));
  }
}

```



```

using
  TICKET -> {i-1..i,i+1},
  uneq,
  remain[p][i-1]
prove
  remain[p][i];
}

forall (p in PROC) forall(q in PROC)
forall (i in TICKET) forall(j in TICKET)
{
  mutex[p][q][i][j]:
  assert G( a[p]=i & a[q]=j ->((at[p]=crit & at[q]=crit)->p=q));

  using
    TICKET -> {i,j},
    uneq,
    remain
  prove
    mutex[p][q][i][j];
  assume remain;
}
}

```