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

Vesa Ojala

Counterexample Analysis for Automated Refinement of Data Abstracted State Machine Models

Master's thesis submitted in partial fulfilment of the requirements for the degree of Master of Science in Technology

Espoo, 1st December 2008

Supervisor: Prof. Ilkka Niemelä
Instructor: Heikki Tauriainen, D.Sc.(Tech.)

HELSINKI UNIVERSITY OF TECHNOLOGY
Faculty of Information and Natural Sciences
Degree Programme of Computer Science and Engineering

ABSTRACT OF MASTER'S THESIS

Author	Vesa Ojala	Date	1st December 2008
		Pages	viii + 68
Title of thesis	Counterexample Analysis for Automated Refinement of Data Abstracted State Machine Models		
Professorship	Theoretical Computer Science	Code	T-119
Supervisor	Prof. Ilkka Niemelä		
Instructor	Heikki Tauriainen, D.Sc.(Tech.)		

State space explosion has been one of the main problems in model checking when dealing with anything but the smallest systems. Different abstraction techniques have been developed to tackle this problem. We use data abstraction for model checking state machine models of objects communicating using asynchronous message passing against assertion failures and implicit message consumption.

When a model checker reports a counterexample from the abstract model, the counterexample does not necessarily correspond to an execution in the original model. Such spurious counterexamples need to be identified and the abstraction needs to be refined so that the spurious counterexample is no longer possible in the abstract model.

In this thesis we describe a technique for identifying spurious counterexamples. We also introduce a method for applying data flow analysis to calculate which of the variables and objects are relevant to the occurrence of the spurious counterexample at different points of the execution. These *relevant locations* help us to focus on the variables needed to be refined in order to remove the spurious counterexample.

We have also developed a method for the automatic refinement of integer interval abstractions, a certain type of data abstraction. This method uses the notion of relevant locations in the search for a suitable refinement.

The methods introduced in this thesis have been implemented in the SMUML (Symbolic Methods for UML Behavioural Diagrams) toolkit.

Keywords

model checking, data abstraction, refinement, UML, state machines

TEKNILLINEN KORKEAKOULU Informaatio- ja luonnontieteiden tiedekunta Tietotekniikan tutkinto-ohjelma	DIPLOMITYÖN TIIVISTELMÄ

Tekijä	Vesa Ojala	Päiväys	1. joulukuuta 2008
		Sivumäärä	viii + 68
Työn nimi	Vastaesimerkkianalyysi abstrahoituja tietotyyppejä käyttävien tilakonemallien automaattiseen hienonnukseen		
Professuuri	Tietojenkäsittelyteoria	Koodi	T-119
Työn valvoja	Prof. Ilkka Niemelä		
Työn ohjaaja	Tekn.tri. Heikki Tauriainen		

Tarkastettavan järjestelmän tila-avaruuden valtava koko on ollut yksi suurimmista ongelmista mallintarkastuksessa, pienimpiä järjestelmiä lukuun ottamatta. Ongelman ratkaisemiseksi on kehitetty erilaisia abstraktiotekniikoita. Tässä työssä käytetään abstrahoituja tietotyyppejä tarkastettaessa assert-lauseiden pitävyyttä ja implisiittisten viestinkulutusten olemassaoloa asynkronisesti viestivistä tilakonemalleista.

Abstrahoitua mallia tarkastettaessa saadut vastaesimerkit eivät välttämättä vastaa ainuttakaan suoritusta alkuperäisessä mallissa. Tällaiset valheelliset vastaesimerkit täytyy tunnistaa ja abstraktiota on hienonnettava valheellisen vastaesimerkin mahdollisuuden poistamiseksi abstrahoidusta mallista.

Tässä diplomityössä kuvataan, miten valheelliset vastaesimerkit voidaan tunnistaa. Työssä esitellään myös menetelmä oleellisten muuttujien ja olioiden löytämiseen suorituksen kussakin pisteessä tietovuoanalyysiä käyttäen. Nämä *oleelliset paikat* auttavat keskittymään muuttujiin, joiden tietotyyppejä hienontamalla saadaan valheellisen vastaesimerkin mahdollisuus abstraktiosta poistettua.

Työssä esitellään myös menetelmä kokonaislukuvälejä abstrakteina tietotyyppeinään käyttävien mallien automaattiseen hienonnukseen. Menetelmä etsii sopivaa hienonnusta käyttämällä hyväkseen oleellisia paikkoja.

Työssä esitellyt menetelmät on toteutettu osana SMUML-projektia (Symbolic Methods for UML Behavioural Diagrams) osaksi SMUML toolkit -ohjelmistoa.

Avainsanat	mallintarkistus, abstrahoidut tietotyypit, abstraktion hienonnus, UML, tilakoneet

Acknowledgements

I want to express my gratitude to my instructor Dr. Heikki Tauriainen for all the discussions and the guidance in the course of this study. I would also like to thank my supervisor Prof. Ilkka Niemelä. I want to thank Dr. Tommi Junttila for the discussions and the support during this study. Finally, I would like to thank my family and friends for the invaluable support during my studies.

This work has been funded by Tekes (Finnish Funding Agency for Technology and Innovation), Nokia Oyj, Conformiq Software Oy, and Mipro Oy. Their support is gratefully acknowledged.

Espoo, 1st December 2008

Vesa Ojala

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With each passing year software systems are becoming more and more complicated. The more complicated they become, the harder it is to achieve bug-free systems. Formal verification methods [11] like model checking [12] have been developed for achieving better quality products. Although model checking techniques have been applied directly to the source code of software system [14, 37, 8, 27], it is common to construct a model representing the logic of the system while hiding some of the details in the real system and then apply model checking programs [9, 29] to the constructed model.

One widely used modelling language used for modelling of software systems is Unified Modelling Language (UML) [2, 1]. The research resulting in this thesis has been done in the Symbolic Methods for UML Behavioural Diagrams (SMUML) project developing model checking techniques for UML models in the Laboratory for Theoretical Computer Science at Helsinki University of Technology. Thus all the techniques described in this thesis have been developed to be part of a model checking framework for UML models. In this thesis we focus on UML features for specifying the behavior of systems: systems are supposed to be described with objects of classes whose behavior have been described with state machines. The UML does not fix an action language for state machines. A Java-like action language Jumbala [18] is used as an action language of state machines.

The huge size of the state space all but the smallest systems tend to have is a major problem for model checkers [36]. Different abstraction techniques have been developed to tackle this state space explosion problem. The idea in abstraction techniques is to simplify the model in order to reduce the state space of the model. The simplification is constructed in such a way that some of the characteristics of the original model are preserved and thus it is possible to prove some properties from the original model by using the abstracted model. In the SMUML project we considered the use of two commonly used abstraction techniques. These techniques are predicate and data abstraction. In SMUML project data abstractions were chosen (for the reasons, see Chapter 3).

Intuitively in data abstractions the domains of variables are replaced with abstract domains having smaller size than the original ones. For example all negative integer values can be represented by a single value in the abstract domain. It is clear that abstract operators, for example addition, need to be defined for abstract domains because the abstract values store only partial information on the values which appear in actual computations. Two logical ways of defining abstract operators are to define the operators as over-approximations or under-approximations. Over-approximatively defined operators produce all the possible outcomes of an operation, for example with an addition of a negative and a positive integer the outcome might be negative, zero or positive. Over-approximative abstractions do not lose behavior in the abstraction though they might add it. On the other hand under-approximative abstractions do not add any extra behavior, but they might lose behavior in the abstraction. For example the addition between a negative and a positive integer can not result negative, zero, nor positive value because none of the results is guaranteed to correspond to the result of an unabstracted operation. Data abstraction can be implemented using value lattices [15] or with non-deterministic operators [26, 24], the latter being our choice.

Our data abstractions are over-approximative abstractions meaning that no behavior in the original model is lost in the abstraction but instead the abstract model may contain extra behavior with no corresponding behavior in the original model. If a data abstracted model cannot violate any assertions or perform implicit message consumptions, the original model cannot do so either [13].

The drawback of the over-approximative data abstraction is that a counterexample demonstrating a property violation in the abstract model does not allow us to automatically conclude that there is an execution violating the property in the original model. The counterexample might demonstrate a property violating execution utilizing the extra behavior introduced by the abstraction. Counterexamples demonstrating execution not possible in the original model are called spurious or infeasible and counterexamples demonstrating a real property violation in the original model are called true or feasible. Feasibility analysis [33] is used for recognizing true counterexamples from the spurious ones.

When the model checker gives us a spurious counterexample it cannot be deduced whether the properties hold in the original model or not. The extra behavior introduced by the abstraction makes the spurious counterexample possible. The existence of the spurious counterexample in the abstract model prevents us from getting more conclusive results. To get more conclusive results a new abstraction with no possibility for the spurious counterexample can be created. Because the occurrence of the spurious counterexample was made possible by the extra behavior introduced by the abstraction, a logical step is to make a new abstraction such that it does not include the extra behavior which made the spurious counterexample possible in the first place.

The old abstraction can be used as a starting point for the new abstraction. Abstraction refinement [10] is a process of modifying the old abstraction so that it more accurately captures the behavior of the original model. The refinement can be made by hand or, with a suitable algorithm, automatically. To achieve good results with the manual approach the person refining the abstraction has to have an understanding of the abstraction technique used, the old abstraction, the spurious counterexample and the original model. In other words refining the abstraction manually requires much detailed knowledge. With automatic abstraction refinement a fully automatic model checking procedure utilizing abstractions can be constructed. It can be left to calculate its results unsupervised and if it encounters a spurious counterexample it refines the abstraction automatically. After the calculation is done we have either a true counterexample demonstrating a real property violation in the original model or we have verified that the properties hold in the original model. Though in the real world, after the refinement the abstract model can have too large a state space and the model checking fails due to the lack of sufficient resources, or even the refinement procedure can fail.

Until now there has not been an automatic refinement for data abstractions. Existing work on data abstractions only mentions that refinement is needed in order to continue the model checking procedure when spurious counterexamples are encountered but describe no method for the actual refinement (see, for example, [33]). Our goal was to develop algorithms for the analysis of spurious counterexamples and finally try to develop an automatic refinement algorithm based on the analysis algorithms. We introduce the notion of a *relevant location* for describing important variables in different objects which contribute to the occurrence of the spurious counterexample. The process of determining relevant locations starts with an analysis to the point of execution where the concrete model does not have any enabled event corresponding to the abstract counterexample. This analysis finds *initial relevant locations* which serve as a starting point for our analysis. Next, from the initial relevant locations we propagate relevant locations to other points of the counterexample trace using data flow analysis [30] shaped to this purpose. The data flow analysis can be thought of as applying program slicing [39, 35, 40] to the counterexample. The relevant locations guide the refinement to be done to the variables whose abstract domains' imprecision (introduced by abstraction) introduced the extra behavior causing the existence of the spurious counterexample.

We have also developed an algorithm for finding refinements for interval abstractions, a subset of data abstractions. An interval abstraction splits the set of integers to a set of intervals such that every integer belongs to one of the intervals and no integer belongs to two distinct intervals. A suitable refinement compromising between the likelihood that the abstraction is refined enough to remove the spurious counterexample and not too much to cause state space explosion is found by analyzing the expressions affecting the values of the relevant locations. The refinement splits the domains in the interval abstracted variables to smaller intervals from suitable places.

This thesis begins with a definition of a notion for a simple state machine model which will be used throughout the thesis as a target for all the algorithms. This notion captures all the important features of UML models with respect to our algorithms. The state machine model is introduced in Chapter 2. Chapter 3 describes the model checking procedure with abstractions and the abstractions used in this thesis. Feasibility analysis is described in Chapter 4. Methods for calculating relevant locations and refinement for interval abstractions are described in Chapter 5. Chapter 6 describes the actual implementation and the relationship between the state machine models used in this thesis and the UML 1.4 models. Finally, in Chapter 7 we summarize the earlier chapters and discuss possible improvements and current problems in the techniques described in the thesis.

Chapter 2

State Machine Model

Systems examined in this thesis are described with models. Our definition for a model has a lot of similarities with UML 1.4 [1] models because the techniques described in this thesis have been designed and implemented for a subset of UML 1.4. However we did not want to describe our algorithms using UML models because UML models contain a lot of unimportant detail in regard to techniques described in this thesis. Conversion from the subset of UML 1.4 models used in the SMUML project to the model formalism used in this thesis is straightforward (described in Section 6.3).

A model contains a set of active objects (instantiated from classes) communicating asynchronously with each other by sending messages or via shared variables. Every message has a type and a list of message parameters. Objects can also receive external messages from the environment. Every class contains a set of variables and a state machine that describes the behavior of the objects instantiated from it. Besides the variables described by the class, objects have an input queue for storing messages sent to the object and a defer queue for deferred messages.

A global configuration defines values of the variables, content of the queues, and active states in a model. Events modify the global configuration of a system.

A state machine consists of states and transitions between the states. One of the states is an initial state. Transitions have three components: a trigger, a guard and a list of effect statements. Each one of the components may be omitted. The trigger receives message of a signal specified in the trigger from the input queue of the object the state machine belongs to. The guard is a boolean expression. A transition is enabled if the object can receive a message of a signal specified in the trigger of the transition and the guard evaluates to true after the message has been received. The transition can be executed if it is enabled. Then the message is received and removed from the input queue of the object and message parameters are assigned to variables defined by the trigger. Statements in the effect of the transition are executed after the message has been received.

There are three different kinds of statements: assignments, statements for sending messages and assertions. Assignments assign a value of an expression to a variable, statements for sending messages send a message to one of the objects in the model, and assertions cause a runtime error if the boolean expression associated with an assertion evaluates to false.

If an object has no transition enabled in the global configuration, it can either implicitly consume the first message in its input queue or defer the reception of the message by moving it to the defer queue of the object. For messages of each signal either implicit consumption of deferring is allowed depending on the active state.

2.1 Formal Definition of a Model

In the following definition of a model (and also later in the thesis) the following functions will be used for sequences and sequence manipulation. A finite sequence of elements $x_1, \ldots, x_n$ is written as $\langle x_1, \ldots, x_n \rangle$. Infinite sequences are not needed in this thesis, from now on we refer to finite sequence simply by sequence. The empty sequence is $\langle\rangle$, every other sequence is non-empty. Function $head(\langle x_1, x_2, \ldots, x_n \rangle) = x_1$ gives the first element in the non-empty sequence. Function $tail(\langle x_1, x_2, \ldots, x_n \rangle) = \langle x_2, \ldots, x_n \rangle$ gives the sequence with its first element removed for non-empty sequences. Function $last(\langle x_1, x_2, \ldots, x_n \rangle) = x_n$ gives the last element in the non-empty sequence. Function $append(\langle x_1, \ldots, x_n \rangle, y) = \langle x_1, \ldots, x_n, y \rangle$ gives the original sequence with an element added to the end of the sequence. Function $concat(\langle x_1, \ldots, x_n \rangle, \langle y_1, \ldots, y_m \rangle) = \langle x_1, \ldots, x_n, y_1, \ldots, y_m \rangle$ concatenates two sequences.

2.1.1 Types, Variables, and Expressions

A *type* d is a set $\{v_1, \ldots, v_n\}$ of possible values. For example a boolean type is $\texttt{boolean} = \{true, false\}$, and 32-bit integer type is $\texttt{int} = \{-2^{31}, -2^{31}+1, \ldots, 2^{31}-1\}$. A type is coercible to another type if every value in the type can be represented unambiguously as a value of the other type. For example integers in the programing language C can be thought to be coercible to boolean values in a way that 0 is coerced to false and other values to true. If a type d_1 is coercible to a type d_2, $coercible(d_1, d_2)$, then function $coerce(v_1, d_1, d_2) = v_2$ gives the unambiguous value v_2 corresponding to the type d_1 value v_1 in the type d_2. A type d is always coercible to itself: $coerce(v, d, d) = v$ for all $v \in d$.

A *variable var* over a set of types D is a pair $\langle name, d \rangle$, where $name(var) = name$ is the name of the variable, and $type(var) = d \in D$ is the type of the variable.

Expressions are represented as Jumbala expression parse trees. Jumbala [18] is a Java-like action language for UML state machines. Expressions are divided into two distinct sets of expressions, compound and terminal expressions. A compound expression e over a set of variables $Vars$ is a tuple $\langle kind, id, d_1, d_2, op, \langle e_1, \ldots, e_n \rangle \rangle$, where

- $kind = kind(e) \in \{\text{COND}, \text{INFIX}, \text{UNARY}, \text{TCOND}\}$ is the kind of the expression e,
- id is the unique expression identifier distinguishing different expressions in the model (a special identifier is needed because different expressions can have identical other components),
- $d_1 = type(e)$ is the type of the expression e,

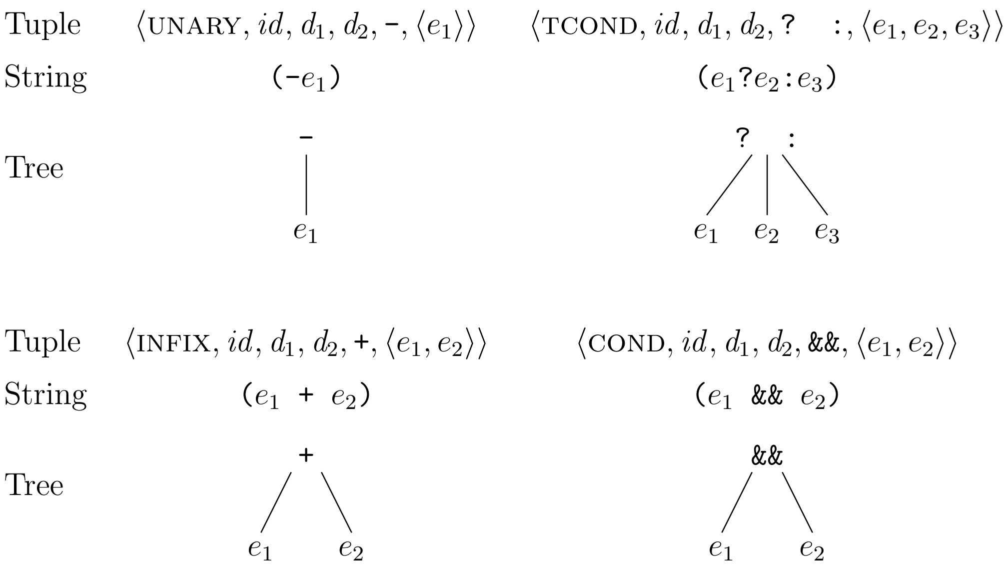

Figure 2.1: Examples of different compound expressions.

- $d_2 = opType(e)$ is the type on which the operator op operates in the expression e,
- $op = operator(e)$ is the operator of the expression e, and
- $subexpr(e) = \langle e_1, \ldots, e_n \rangle$ is the list of e's subexpressions. In tree representation, seen in Figure 2.1, the first subexpression e_1 is the leftmost subtree, the second subexpression e_2 is the second leftmost subtree, and so on. For COND, INFIX, and UNARY kind expressions every subexpression has to be a type that can be coerced to the type on which the operator operates, $\forall 1 \leq i \leq n : coercible(type(e_i), d_2)$. For TCOND kind expressions subexpressions e_2 and e_3 have to be of a type that can be coerced to the type on which the operator operates, $\forall i \in \{2, 3\} : coercible(type(e_i), d_2)$, and subexpression e_1 has to be a boolean type expression, $type(e_1) = \texttt{boolean}$.

Compound expressions of a kind UNARY have one subexpression, COND and INFIX have two subexpressions, and TCOND has three subexpressions. Figure 2.1 contains examples of different compound expressions. The string representation of expressions is used in some examples for the sake of simplicity.

A terminal expression e over a set of variables $Vars$ is a tuple $\langle kind, id, d, symbol \rangle$, where

- $kind = kind(e) \in \{\text{LIT}, \text{NAME}\}$ is the kind of the expression e,
- id is the unique expression identifier distinguishing different expressions in the model (a special identifier is needed because different expressions can have identical other components),
- $d = type(e)$ is the type of the expression e, and

- If $\mathit{kind} = \textsc{lit}$ then $\mathit{symbol} \in d$. Otherwise $\mathit{kind} = \textsc{name}\mathit{andsymbol}$ is a sequence $\langle \mathit{var}_1, \ldots, \mathit{var}_n \rangle$ where every var_i is a variable from a set of variables Vars, $\mathit{type}(\mathit{var}_n) = d$, and $\forall 1 \leq i \leq n-1 : \mathit{type}(\mathit{var}_i) = \texttt{reference}$. Type `reference` is used for accessing objects in a model and is defined later.

Example 2.1. Let expression e_1 represented as a string be `(2 != (5 +` $\langle \mathit{var} \rangle$`))`. Let e_1's subexpressions be `2` $= e_2$ and `(5 +` $\langle \mathit{var} \rangle$`)` $= e_3$. Let e_3's subexpressions be `5` $= e_4$ and $\langle \mathit{var} \rangle = e_5$. Then expression e_1 is formally represented as $\langle \textsc{infix}, \mathit{id}_1, \texttt{boolean}, \texttt{int}, \texttt{!=}, \langle e_2, e_3 \rangle \rangle$, its subexpressions are $e_2 = \langle \textsc{lit}, \mathit{id}_2, \texttt{int}, 2 \rangle$, and $e_3 = \langle \textsc{infix}, \mathit{id}_3, \texttt{int}, \texttt{int}, \texttt{+}, \langle e_4, e_5 \rangle \rangle$, and the expression e_3's subexpressions are $e_4 = \langle \textsc{lit}, \mathit{id}_4, \texttt{int}, 5 \rangle$, and $e_5 = \langle \textsc{name}, \mathit{id}_5, \texttt{int}, \langle \mathit{var} \rangle \rangle$. ■

2.1.2 Signals, Messages, Queues, and State Machines

A *signal sig* over a set of types D is a pair $\langle \mathit{name}, \mathit{paramtypes} \rangle$, where

- *name* is the name of the signal, and
- $\mathit{paramtypes} = \mathit{params}(\mathit{sig}) = \langle d_1, \ldots, d_n \rangle$ is the sequence of parameter types, where $\forall 1 \leq i \leq n : d_i \in D$.

A *message msg* of a signal *sig* is a tuple $\langle \mathit{id}_{msg}, \mathit{sig}, v_1, \ldots, v_n \rangle$, where id_{msg} is an identifier used for separating otherwise identical messages from each other and $\forall 1 \leq i \leq n : v_i \in d_i$ when $\mathit{params}(\mathit{sig}) = \langle d_1, \ldots, d_n \rangle$.

A *message queue* over a set of signals *Sigs* is a (possibly empty) finite sequence of messages over the signals in *Sigs*. Let $\mathit{queues}(D, \mathit{Sigs})$ represent all possible message queues over the types D and the signals *Sigs*.

A *state machine* over a set of variables *Vars*, system signals *SysSigs*, and external signals *ExtSigs* ($\mathit{Sigs} = \mathit{SysSigs} \cup \mathit{ExtSigs}$), is a tuple $\langle s_i, S, T, \mathit{defers}, \mathit{flush} \rangle$, where

- $s_i \in S$ is the *initial state*,
- $S = \mathit{states}(\mathit{sm})$ is the set of *states*
- T is the set of *transitions* $t = \langle \mathit{tid}, s_1, s_2, \mathit{trig}, g, \mathit{eff} \rangle$ between states, where
  - *tid* is the unique transition identifier distinguishing transitions where all the other components are identical in the state machine,
  - $s_1 \in S$ is the source state,
  - $s_2 \in S$ is the destination state,
  - the *trigger trig* is either ϵ or a tuple of the form $\langle \mathit{sig}, \langle p_1, \ldots, p_n \rangle \rangle$, where $\mathit{sig} \in \mathit{SysSigs} \cup \mathit{ExtSigs}$, $\mathit{params}(\mathit{sig}) = \langle d_1, \ldots, d_n \rangle$, and $\forall 1 \leq i \leq n : p_i \in \mathit{Vars}$, $\mathit{coercible}(d_i, \mathit{type}(p_i))$,
  - g is a boolean type expression, $\mathit{type}(g) = \texttt{boolean}$, over the set of variables *Vars* called the *guard*, and
  - *eff*, the *effect*, is a sequence of tuples of the form:

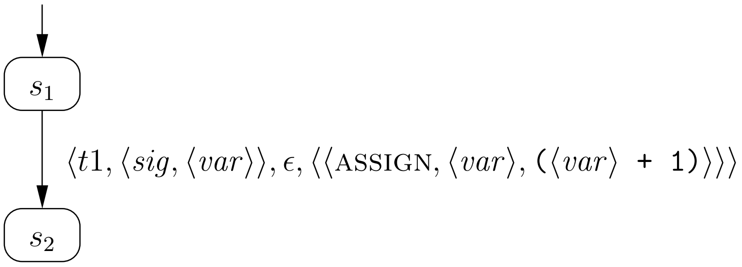

Figure 2.2: An example of a state machine.

- $*$ $\langle \text{SEND}, sig, \langle e_1, \ldots, e_n \rangle, tgt \rangle$, where $sig \in Sigs$, message parameter types are $params(sig) = \langle d_1, \ldots, d_n \rangle$, every e_i is an expression over the set of variables *Vars* such that $\forall 1 \leq i \leq n : coercible(type(e_i), d_i)$, and *tgt* is an object reference type expression, $type(tgt) = \texttt{reference}$ (`reference` defined later),
- $*$ $\langle \text{ASSIGN}, lhs, rhs \rangle$, where *lhs* is a NAME expression, *rhs* is an expression, and the type of expression *rhs* can be coerced to the type of expression *lhs*, $coercible(type(rhs), type(lhs))$, and
- $*$ $\langle \text{ASSERT}, e \rangle$, where e is a boolean type expression over the set of variables *Vars*.

- *defers* is a function giving the set of deferrable signals $defers(s) \subseteq SysSigs$ for each state $s \in S$, and
- *flush* $\subseteq S$ is a subset of the states in the state machine. Moving to these states in the state machine causes the *defer queue* to be flushed into the *input queue*. Input and defer queues are described in Section 2.1.3. For those familiar with UML, the set *flush* is used in the implementation of message deferral.

Example 2.2. In Figure 2.2 is a graphical representation of a state machine. Formally the state machine is $\langle s_1, S, T, defers, flush \rangle$, where the set of states is $S = \{s_1, s_2\}$ and the set of transitions $T = \{\langle t1, s_1, s_2, trig, \epsilon, \langle stmt \rangle\rangle\}$. In the graphical representation the initial state s_1 is marked with an arrow with no source state. The state machine has only one transition. Its trigger is $trig = \langle sig, \langle var \rangle\rangle$ and the only action in the effect is $stmt = \langle \text{ASSIGN}, \langle var \rangle, (\langle var \rangle \texttt{ + 1})\rangle$. The behavior of the function *defers* and the content of the set *flush* are not shown in the graphical representation. ■

2.1.3 Classes, Objects, Global Configurations, and Models

A class c over a set of types D, system signals *SysSigs*, and external signals *ExtSigs* is a pair $\langle Vars, sm \rangle$, where

- $Vars = Vars(c)$, the set of variables in c, is a set of variables over D, and
- $statemachine(c) = sm$, the state machine of c, is a state machine over the variables *Vars*, the system signals *SysSigs*, and the external signals *ExtSigs*.

An object o of a class $class(o) = c$ is a pair $\langle c, oid\rangle$, where c is the class of the object, and $oid = id(o)$ is the unique identifier of the object distinguishing it from other instances of the same class in the model. Let $Vars(o) = Vars(c)$ be the set of variables in the class of which the object is an instance.

Let O be a set of objects from a set of classes *Classes* defined over some set of types D, a set of system signals *SysSigs* and a set of external signals *ExtSigs*. The set O induces the set of *locations Locations*, which contains pairs of objects and variables $\{\langle o, var\rangle \mid o \in O, var \in Vars(o)\}$. These locations map variables to their values.

A *global configuration* C of a set of objects O from a set of classes *Classes* over types D and signals *Sigs*, and a set of locations *Locations* induced by O is a tuple $\langle sn, state, inputqueue, deferqueue, valuation\rangle$, where

- sn is the sequence number of C to track the order of global configurations in sequences of global configurations.
- $state(o) = s$, where $state$ is a function mapping objects $o \in O$ to active states $s \in states(statemachine(class(o)))$,
- $inputqueue(o) \in queues(D, Sigs)$, where $inputqueue$ is a function mapping an object $o \in O$ to a message queue,
- $deferqueue(o) \in queues(D, Sigs)$, where $deferqueue$ is a function mapping an object $o \in O$ to a message queue, and
- $valuation(loc) = v$, where $valuation$ is a function mapping locations $loc = \langle o, var\rangle \in Locations$ to values $v \in type(var)$.

Let $sn(C) = sn$ be the sequence number of the global configuration C. Let $Act(C, o) = state(o)$ be the active state in o in the global configuration C. Let $InputQueue(C, o) = inputqueue(o)$ be the input queue and $DeferQueue(C, o) = deferqueue(o)$ be the defer queue in a global configuration C. Let $VarValue(C, o, var) = valuation(\langle o, var\rangle)$ be the value of variable var in the global configuration C.

For a NAME expression e, function $resolve(C, o, e) = \langle o', var'\rangle$ gives the location the expression represents. When $e = \langle \textsc{name}, id, d, \langle var_1, \ldots, var_n\rangle\rangle$ and $n \geq 1$, $resolve(C, o, e)$

$$= \begin{cases} \langle o, var_1\rangle & \text{if } n = 1,\ o \in O,\ var_1 \in Vars(o), \\ & type(var_1) = d \\ resolve(C, o_1, e_1) & \text{if } n \geq 2,\ o \in O,\ var_1 \in Vars(o), \\ & type(var_1) = \texttt{reference},\ o_1 = VarValue(C, o, var_1), \\ & e_1 = \langle \textsc{name}, id, d, \langle var_2, \ldots, var_n\rangle\rangle \end{cases}$$

For an expression e, an object o, and a global configuration C, $eval(C, F, o, e) = v$ is the value of expression e evaluated in a context of the object o in the global configuration C when F is used for solving possible non-deterministic choices in the expressions. Function F and expression with non-determinism are introduced along with the abstractions in Chapter 3.

A *model* is a tuple $\langle C_{init}, D, Classes, O, Locations, SysSigs, ExtSigs\rangle$, where

- C_{init} is an initial global configuration, or simply the initial configuration, of the model, $sn(C_{init}) = 1$, input and defer queues for all objects in the initial configuration are empty, active state in all objects is the initial state in the object's state machine,
- D is the set of types which includes the boolean type $\texttt{boolean} = \{true, false\}$ and the object reference type $\texttt{reference} = O \cup \{null\}$,
- $SysSigs$ is the set of system signals over the set of types D,
- $ExtSigs$ is the set of external signals over the set of types D. Signals in the set of external signals cannot have parameters. External signals must be disjoint from the set of system signals, $SysSigs \cap ExtSigs = \emptyset$,
- $Classes$ is the set of classes over the set of types D, the set of system signals $SysSigs$, and the set of external signals $ExtSigs$,
- O is the set of objects in the model, and
- $Locations$ is the set of locations induced by O.

Example 2.3. A simple model with one class and one object instantiated from the only class in the model could be $\langle C_{init}, D, Classes, O, Locations, SysSigs, ExtSigs\rangle$, where

- $D = \{\texttt{int}, \texttt{boolean}, \texttt{reference}\}$,
- $SysSigs = \emptyset$,
- $ExtSigs = \{sig\}$,
- a set of classes $Classes = \{c\}$ contains only one class, $c = \langle\{var\}, sm\rangle$, where sm is the state machine from Figure 2.2,
- a set of objects $O = \{o\}$ contains the only object, $o = \langle c, 1\rangle$, which is instantiated from the only class in the model,
- $Locations = \{\langle o, var\rangle\}$, and
- $C_{init} = \langle 1, s_1, inputqueue, deferqueue, valuation\rangle$.

For our only object $inputqueue(o) = InputQueue(C_{init}, o) = \langle\rangle$, $deferqueue(o) = DeferQueue(C_{init}, o) = \langle\rangle$, and for the only variable in the object we can define, for example, $valuation(\langle o, var\rangle) = 0$. ■

2.2 Execution of Transition Components

Transitions consist of different configuration altering components. In this section we describe how these components modify the configuration. In Section 2.4 these definitions are used when the effects of a transition execution are introduced. The execution of components of transitions is partially defined, cases not defined are not used in the algorithms.

2.2.1 Receiving a Message

Receiving a message by a trigger $trig$ in a global configuration C in an object o produces a new global configuration $C' = exec_{recv}(C, o, trig)$ where C' is formed according to the following rules:

- If $trig = \epsilon$, then $C = C'$,
- if $trig = \langle sig, \langle p_1, \ldots, p_n \rangle \rangle$ and $head(InputQueue(C, o)) = \langle id_{msg}, sig, v_1, \ldots, v_n \rangle$, then $C' = C$ except that
  - The sequence number is incremented: $sn(C') = sn(C) + 1$,
  - The message to be received is removed from the input queue: $InputQueue(C', o) = tail(InputQueue(C, o))$, and
  - Message parameters are assigned to the variables specified in the trigger: $\forall 1 \le j \le n : VarValue(C', o, p_j) = v_j$.

2.2.2 Sending a Message

The execution of a send action $stmt = \langle \textsc{send}, sig, \langle e_1, \ldots, e_n \rangle, tgt \rangle$ in an object o in a global configuration C produces a new global configuration $C' = exec_{eff}(C, F, o, stmt)$ where $C' = C$ except that:

- The sequence number is incremented: $sn(C') = sn(C) + 1$,
- The object which receives the message is determined: $o'' = eval(C, F, o, tgt)$,
- The message is formed and the parameter values are determined: $msg = \langle sn(C), sig, v_1, \ldots, v_n \rangle$, $\forall 1 \le j \le n : v_j = eval(C, F, o, e_j)$, and
- The message is added to the end of the input queue of the object which receives the message: $InputQueue(C', o'') = append(InputQueue(C, o''), msg)$.

2.2.3 Assignment

Executing an assign action $stmt = \langle \textsc{assign}, lhs, rhs \rangle$ in an object o in a global configuration C produces a new global configuration $C' = exec_{eff}(C, F, o, stmt)$ where $C' = C$ except that:

- The sequence number is incremented: $sn(C') = sn(C) + 1$, and
- The value $v = eval(C, F, o, rhs)$ is assigned to a location indicated by $\langle o'', var \rangle = resolve(C, o, lhs)$: $VarValue(C', o'', var) = v$.

2.2.4 Assertion

The execution of an assertion action $stmt = \langle \text{ASSERT}, e\rangle$ in an object o in a global configuration C produces a new global configuration $C' = exec_{eff}(C, F, o, stmt)$ such that:

- If the condition in the assertion statement evaluates to true, $eval(C, F, o, e) = true$, then $C' = \langle sn + 1, state, inputqueue, deferqueue, valuation\rangle$, when $C = \langle sn, state, inputqueue, deferqueue, valuation\rangle$, or

- else $C' = \perp$, where $\perp$ is a special value indicating an assertion error in the execution.

2.2.5 Moving from a State to Another

Moving to a state s in a global configuration C in an object o produces a new global configuration $C' = exec_{goto}(C, o, s, b)$ when $b = true$ if $s \in flush$, otherwise $b = false$, and where $C' = C$ except that:

- The sequence number is incremented: $sn(C') = sn(C) + 1$,

- The active state of object o is set to the new state: $Act(C', o) = s$, and

- The messages in o's defer queue are moved to the beginning of o's input queue if $b = true$: $InputQueue(C', o) = concat(DeferQueue(C, o), InputQueue(C, o))$ and $DeferQueue(C', o) = \langle\rangle$, otherwise $InputQueue(C', o) = InputQueue(C, o)$ and $DeferQueue(C', o) = DeferQueue(C, o)$.

2.3 Enabled Events

Execution in the model proceeds by execution of events. Events are atomic, in particular the execution of a transition consists of several steps executed together as a single atomic event. An event can be executed only if it is enabled.

In a configuration C a transition $t = \langle tid, s_1, s_2, trig, g, eff\rangle$ execution event in an object o, $a = \langle \text{TRANS}, F, o, t\rangle$, is enabled, $enabled(C, a)$, if:

- The source state s_1 of the transition is an active state: $Act(C, o) = s_1$,

- The trigger is empty, $trig = \epsilon$, or there is a message of a signal corresponding to the trigger in the head of o's input queue: $trig = \langle sig, \langle p_1, \ldots, p_n\rangle\rangle$ when $head(InputQueue(C, o)) = \langle id_{msg}, sig, v_1, \ldots, v_n\rangle$, and

- The trigger is empty and the guard evaluates to true in the configuration C, or the trigger is non-empty and the guard evaluates to true in the configuration $C' = exec_{recv}(C, o, trig)$ which would result from receiving the message in the configuration C.

In configuration C deferring a message $a = \langle \text{DEFER}, o\rangle$ in an object o is enabled, $enabled(C, a)$, if:

- The input queue of o is not empty: $|InputQueue(C, o)| > 0$,
- The object o has no enabled transitions: $\forall t \in T : \neg enabled(C, \langle \textsc{trans}, F, o, t\rangle)$, when T is the set of transitions in the state machine of the object o,
- The message at the head of o's input queue can be deferred in the active state: $sig \in defers(Act(C, o))$, when $head(InputQueue(C, o)) = \langle id_{msg}, sig, v_1, \ldots, v_n\rangle$.

In configuration C, the execution of an implicit message consumption event $a = \langle \textsc{impl}, o\rangle$ in an object o is enabled, $enabled(C, a)$, if:

- The input queue of o is not empty: $|InputQueue(C, o)| > 0$,
- The object o has no enabled transitions: $\forall t \in T : \neg enabled(C, \langle \textsc{trans}, F, o, t\rangle)$, when T is the set of transitions in the state machine of the object o,
- The message at the head of o's input queue cannot be deferred in the active state: $sig \notin defers(Act(C, o))$, when $head(InputQueue(C, o)) = \langle id_{msg}, sig, v_1, \ldots, v_n\rangle$.

No event is enabled if $C = \perp$.

2.4 Execution of Events

Executions of events alter the global configuration in the following ways.

2.4.1 Execution of Transition Events

The execution of an enabled transition execution event $a = \langle \textsc{trans}, F, o, t\rangle$, where $t = \langle tid, s_1, s_2, trig, g, eff\rangle$, in a global configuration C produces a new global configuration $C' = exec(C, a)$ by the following steps:

- First the object receives a message corresponding to the trigger's signal: $C_t = exec_{recv}(C, o, trig)$,
- Then every action in $eff = \langle stmt_1, \ldots, stmt_n\rangle$ is executed:
  - $C_1 = exec_{eff}(C_t, F, o, stmt_1)$
  - $\forall 2 \leq i \leq n : C_i = exec_{eff}(C_{i-1}, F, o, stmt_i)$
  - If some action produces a global configuration $\perp$ representing an assertion error, then the execution is halted and the resulting global configuration is set to $C' = \perp$,
- The object moves from the state s_1 to the state s_2: $C' = exec_{goto}(C_n, o, s_2, b)$, where $b = true$ if $s_2 \in flush$, otherwise $b = false$.

2.4.2 Execution of Implicit Message Consumption

The execution of an enabled implicit message consumption event $a = \langle \text{IMPL}, o \rangle$, in a global configuration C produces a new global configuration $C' = exec(C, a)$ such that:

- The sequence number is incremented: $sn(C') = sn(C) + 1$,
- The message to be consumed is removed from the input queue: $InputQueue(C', o) = tail(InputQueue(C, o))$,
- Everything else is like it was before the message was consumed:
  - $\forall o' \in O : \forall var \in Vars(o') : VarValue(C', o', var) = VarValue(C, o', var)$
  - $\forall o' \in O \setminus \{o\} : InputQueue(C', o') = InputQueue(C, o')$
  - $\forall o' \in O : DeferQueue(C', o') = DeferQueue(C, o')$
  - $\forall o' \in O : Act(C', o') = Act(C, o')$.

2.4.3 Deferring a Message

The execution of an enabled message defer event $a = \langle \text{DEFER}, o \rangle$, $enabled(C, a)$, in a global configuration C produces a new global configuration $C' = exec(C, a)$ such that:

- The sequence number is incremented: $sn(C') = sn(C) + 1$,
- The message to be deferred is removed from the input queue: $InputQueue(C', o) = tail(InputQueue(C, o))$,
- The message is added to the defer queue: $DeferQueue(C', o) = append(DeferQueue(C, o), msg)$, when $InputQueue(C, o) = \langle msg, \ldots \rangle$,
- Everything else is like it was before the message was deferred:
  - $\forall o' \in O : \forall var \in Vars(o') : VarValue(C', o', var) = VarValue(C, o', var)$
  - $\forall o' \in O \setminus \{o\} : InputQueue(C', o') = InputQueue(C, o')$
  - $\forall o' \in O \setminus \{o\} : DeferQueue(C', o') = DeferQueue(C, o')$
  - $\forall o' \in O : Act(C', o') = Act(C, o')$

2.5 Executions in a Model

An execution in a model $M = \langle C_{init}, D, Classes, O, Locations, SysSigs, ExtSigs \rangle$ is a sequence $trace = \langle b_1, \ldots, b_n \rangle$, where $b_i = \langle C_i, a_i, C_{i+1} \rangle$, $1 \leq i \leq n$, such that

- $\forall 1 \leq i \leq n : a_i$ is a transition execution event, an implicit consumption event, or a message defer event,
- $\forall 1 \leq i \leq n + 1 : C_i$ is a global configuration,
- $C_1 = C_{init}$

- $\forall 1 \leq i \leq n : enabled(C_i, a_i)$.
- $\forall 1 \leq i \leq n : C_{i+1} = exec(C_i, a_i)$

Chapter 3

Model Checking with Abstractions

The objective in model checking [12] is to check whether some properties, which are for some reason interesting to the user, hold in the model to be model checked. The model checker takes a model and a set of properties as an input and outputs either a message that the properties hold in the model or gives a counterexample illustrating an execution in the model that violates one of the properties. Figure 3.1 illustrates the normal model checking procedure. In this thesis the properties we are model checking against are the absence of assertion failures and implicit message consumptions.

The need for abstraction techniques arises from the huge size of the state space all but the smallest models tend to have. The phenomenon is called state space explosion [36]. Different abstraction techniques have been developed to tackle this problem. In abstraction the idea is to create another model, abstract model, which represents the behavior of the original model but abstracts away some details from the original model to reduce the size of its state space. Then the abstract model is checked with a model checker for the same properties that we are interested in the original model.

The abstract model is constructed in such a way that some of the characteristics of the original model are preserved thus making it possible to prove some properties from the original model by using the abstract model. For example, over-approximative abstraction techniques add behavior to the abstract model compared to the concrete model but all the behaviors in the concrete model have a corresponding behavior in the abstract model. Therefore if the abstract model created in an over-approximative manner does not contain assertion failures, then the corresponding concrete model does not contain assertion failures either [13]. On the other hand if the abstract model contains assertion failures, we cannot tell without further analysis whether the concrete model contains assertion failures because the over-approximative abstraction

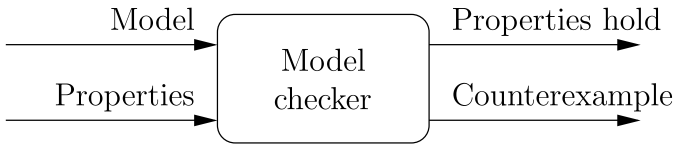

Figure 3.1: Conventional model checking procedure.

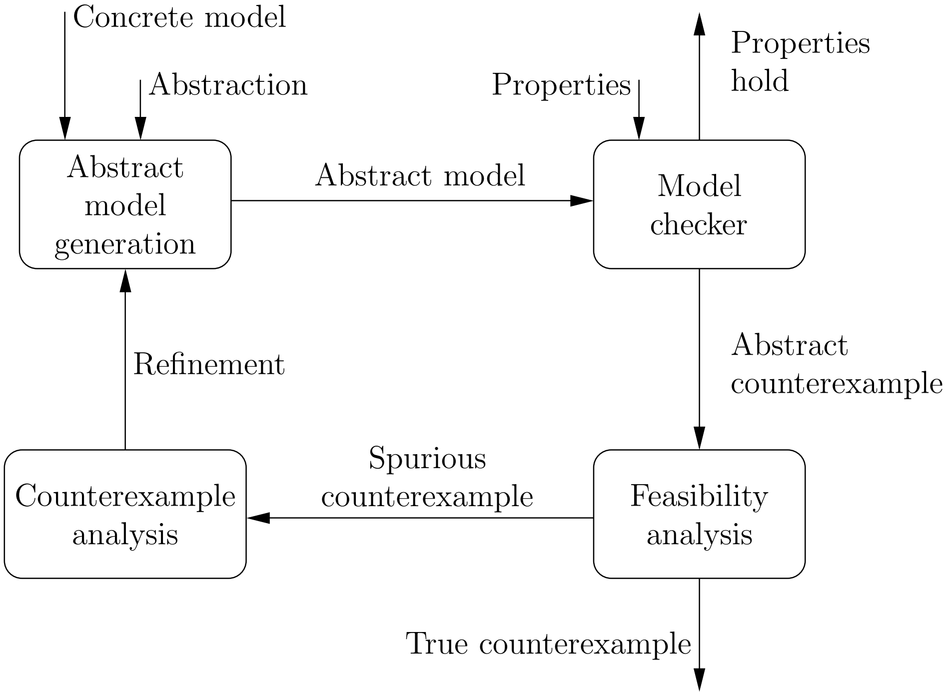

Figure 3.2: Model checking procedure with abstractions and abstraction refinement.

might have added new behaviors to the abstract model.

The process for determining whether an abstract counterexample corresponds to an execution in the concrete model is called *feasibility analysis* [33]. If an abstract counterexample has a corresponding execution in the concrete model then the counterexample is called *feasible*, otherwise it is *spurious*. The execution of a spurious counterexample in the abstract model violates the properties but its execution in the concrete model does not or the corresponding execution is not even possible in the concrete model. An example of a spurious counterexample can be found in Section 3.3.

Even if a spurious counterexample is encountered, we want to either prove that the properties hold in the concrete model or that there is a true counterexample demonstrating that a property does not hold in the concrete model. To be able to do this, we have to *refine* [10] the abstraction (make the abstraction more precise). The purpose of *refinement* is to change the abstraction in a way that the spurious counterexample does not appear in the abstract model constructed with the refined abstraction. After the abstraction is refined, a new abstract model is produced with the new abstraction and the whole process starts again. The refinement can be done by hand or automatically. In our case the counterexample analysis described in Section 5 analyzes automatically the way the abstraction needs to be refined. The model checking procedure with abstractions and abstraction refinement is shown in Figure 3.2.

Two different abstraction techniques, data abstractions and predicate abstractions, were considered in the SMUML project. In data abstractions the domains of variables are replaced with abstract domains having smaller size than the original ones. Because of the abstract domain's smaller size abstract values store only partial information on the values which appear in actual computations. Data abstraction can be implemented using value lattices [15] or with non-deterministic operators [26, 24], the latter being our choice. However, there was no prior algorithm for automatic refinement of data abstractions as there was for predicate abstractions (for example [8]).

The predicate abstraction approach was still not chosen to be included into the SMUML toolset because the object-oriented nature with concurrency and asynchronous message passing of the UML model is quite challenging for the predicate abstraction. Also only one existing implementation of predicate abstraction in an object-oriented environment was known (Java PathFinder [38]). On the other hand object-orientation, concurrency, and asynchronous message passing do not introduce any real extra difficulties for data abstractions. The second reason was that the implementation of a predicate abstraction engine usually needs a theorem prover. The prover may have to be called an exponential number of times at some step of the model checking procedure with abstractions [4, 25, 37].[1]

3.1 Data Abstractions

In this thesis we shall use data abstractions. In data abstractions an abstract model is created by changing the domains of variables in the concrete model to abstract domains. An abstract domain has to represent the same set of values as in the concrete domain, i.e. there is an abstraction function mapping every value in the concrete domain to a value in the abstract domain. The data abstractions we use are defined as over-approximations, therefore if the abstract model does not contain assertion failures or implicit message consumptions, neither does the concrete model. Also we allow the use of data abstraction only to integers. The formal definition of the relationship between concrete and abstract models is given in Section 3.2.

3.1.1 Evaluation of Abstract Expressions

Let expression e_c^1 as its string representation be $\texttt{(}\langle var_c^1\rangle\ \texttt{+}\ \langle var_c^2\rangle\texttt{)}$. Formally the expression is $e_1 = \langle \text{INFIX}, 1, \texttt{int}, \texttt{int}, \texttt{+}, \langle e_c^2, e_c^3\rangle\rangle$, and its subexpressions are $e_c^2 = \langle \text{NAME}, 2, \texttt{int}, \langle var_c^1\rangle\rangle$ and $e_c^3 = \langle \text{NAME}, 3, \texttt{int}, \langle var_c^2\rangle\rangle$.

Let `Sign` be an abstract type with three values, `NEG` representing all negative integers, `ZERO` representing 0, and `POS` representing all positive integers. Let variables var_a^1 and var_a^2 be `Sign`-abstracted variables corresponding to the variables var_c^1 and var_c^2. An expression $e_a^1 = \langle \text{INFIX}, 1, \texttt{Sign}, \texttt{Sign}, \texttt{+}, \langle e_a^2, e_a^3\rangle\rangle$ is a `Sign`-abstracted version of expression e_c^1. The subexpressions of e_a^1 are $e_a^2 = \langle \text{NAME}, 2, \texttt{Sign}, \langle var_a^1\rangle\rangle$ and $e_a^3 = \langle \text{NAME}, 3, \texttt{Sign}, \langle var_a^2\rangle\rangle$.

Let C_c be a global configuration describing the state of the concrete model and let o_c be an object in the concrete model. Let $VarValue(C_c, o_c, var_c^1) = 2$, and $VarValue(C_c, o_c, var_c^2) = -1$. Let C_a be a global configuration corresponding to C_c in the abstract model and let o_a be an object corresponding to o_c in the abstract model.

[1]The abstract model generator in SMUML toolkit still uses a theorem prover (SMT solver, for example [23, 5, 16]) in the generation of abstract operators.

Let C_a have values corresponding to the values in C_c (determined by the abstraction function): $VarValue(C_a, o_a, var_a^1) = \texttt{POS}$, $VarValue(C_a, o_a, var_a^2) = \texttt{NEG}$. When e_c^1 is evaluated in the object o_c using the global configuration C_c, we get -1 as a result. On the other hand when we want to evaluate e_a^1 in the object o_a using the global configuration C_a, we do not have one correct solution but multiple possible solutions: when adding a positive and a negative integer we cannot tell whether the result is positive, zero, or negative. This imprecision introduced by the abstraction introduces the extra behaviors to the abstract model.

This is where the function for solving non-determinism F comes into use. Function *eval* introduced in Section 2.1.3 evaluates expressions using a function given as second argument for solving possible non-determinism in the evaluation. The function F identifies the expressions using the unique identifier in the expressions of a model. For example, every time $eval(C_a, F, o_a, e_a^1)$ is called it gives us the same result because F fixes the non-determinism. Function F is used to make sure that the same choices are made in non-deterministic points of evaluation in the model checker and in the analysis of the counterexample.

3.2 Formal Definition of an Abstract Model

Using the definitions from Chapter 2, a concrete model is defined as a tuple $M_c = \langle C_{init\,c}, D_c, Classes_c, O_c, Locations_c, SysSigs_c, ExtSigs\rangle$, where the set of types $D_c = \{\texttt{int}, \texttt{boolean}, \texttt{reference}\}$. Type `int` represents 32-bit integers. Because abstract models are also models even though they have a special relationship to the corresponding concrete model, they also obey the definitions described in Chapter 2. In the following sections we describe the constraints that are used when abstract models are generated from concrete models. Basically the constraints just make sure that the abstract model has a suitable set of types, and its variables correspond to the concrete variables. We start our definitions from abstract types building up all the way to the definition of an abstract model. It should be noted that we only describe the constraints restricting the generation of abstract models, not an algorithm for generating abstract models. The abstract models appearing later in this thesis obey the definitions described in the following sections.

3.2.1 Abstract Types and Variables

An abstract type $d = \{v_1, \ldots, v_n\}$ is a set of values where each value represents a set of integer values. Thus, $coercible(\texttt{int}, d)$. Let *abstTypes* be a set of abstract types. An abstract type d_1 can be, but necessarily does not have to be, coercible to another abstract type d_2.

An abstract variable $var_a = \langle name_a, d_a\rangle$ of an abstract type d_a *corresponds* to a concrete variable $var_c = \langle name_c, d_c\rangle$ of a concrete type d_c if $name_a = name_c$, and either $d_a = d_c$ or $d_c = \texttt{int}$ and $d_a \in abstTypes$. We write $var_c \sim var_a$ to denote this correspondence.

A set of abstract variables $Vars_a = \{var_a^1, \ldots, var_a^n\}$ corresponds to a set of concrete variables $Vars_c = \{var_c^1, \ldots, var_c^m\}$, $Vars_c \sim Vars_a$, if $n = m$, and $var_c^i \sim var_a^i$

for all $1 \leq i \leq n$.

3.2.2 Abstract Expressions

Let $e_c = \langle kind, id, d_c^1, d_c^2, op, \langle e_c^1, \ldots, e_c^n \rangle \rangle$ be a compound expression (over concrete types D_c, variables $Vars_c$), and let $e_a = \langle kind, id, d_a^1, d_a^2, op, \langle e_a^1, \ldots, e_a^n \rangle \rangle$ be an abstract compound expression (over abstract types $D_a = D_c \cup abstTypes$, variables $Vars_a$). The expression e_a corresponds to the concrete expression e_c, $e_c \sim e_a$, if

- $d_a^1 \in abstTypes \cup \{\texttt{int}\}$ if $d_c^1 = \texttt{int}$, otherwise $d_a^1 = d_c^1$,
- $d_a^2 \in abstTypes \cup \{\texttt{int}\}$ if $d_c^2 = \texttt{int}$, otherwise $d_a^2 = d_c^2$, and
- $\forall 1 \leq i \leq n : e_c^i \sim e_a^i$.

Let $e_c = \langle kind, id, d_c, symbol_c \rangle$ be a terminal expression (over concrete types D_c, variables $Vars_c$), and let $e_c = \langle kind, id, d_a, symbol_a \rangle$ be an abstract terminal expression (over abstract types $D_a = D_c \cup abstTypes$, variables $Vars_a$). The expression e_a corresponds to the concrete expression e_c, $e_c \sim e_a$, if

- $d_a = abstTypes \cup \{\texttt{int}\}$ if $d_c = \texttt{int}$, otherwise $d_a = d_c$, and
- $symbol_a = symbol_c$ if $kind = \text{NAME}$, otherwise $symbol_a = coerce(symbol_c, d_c, d_a)$.

3.2.3 Abstract Classes

An abstract signal sig_a corresponding to a concrete signal sig_c, $sig_c \sim sig_a$, is a tuple $\langle name, \langle d_a^1, \ldots, d_a^n \rangle \rangle$, where $sig_c = \langle name, \langle d_c^1, \ldots, d_c^n \rangle \rangle$ and $\forall 1 \leq i \leq n$: $d_a^i = d_c^i$ if $d_c^i \in \{\texttt{boolean}, \texttt{reference}\}$, otherwise $d_a^i \in abstTypes \cup \{\texttt{int}\}$.

A set of abstract signals $Sigs_a = \{sig_a^1, \ldots, sig_a^n\}$ corresponds to a set of concrete signals $Sigs_c = \{sig_c^1, \ldots, sig_c^m\}$, $Sigs_c \sim Sigs_a$, if $n = m$, and $sig_c^i \sim sig_a^i$ for all $1 \leq i \leq n$.

An abstract transition t_a (over an abstract set of system signals $SysSigs_a$) corresponding to a transition $t_c = \langle tid, s_1, s_2, trig_c, g_c, eff_c \rangle$ (over a concrete set of system signals $SysSigs_c$, $SysSigs_c \sim SysSigs_a$), $t_c \sim t_a$, is $t_a = \langle tid, s_1, s_2, trig_a, g_a, eff_a \rangle$, where

- $trig_a = \langle sig_a, \langle p_c^1, \ldots, p_c^n \rangle \rangle$ corresponds to $trig_c = \langle sig_c, \langle p_c^1, \ldots, p_c^n \rangle \rangle$, $trig_c \sim trig_a$, where $sig_c \sim sig_a$ and $\forall 1 \leq i \leq n : p_c^i \sim p_a^i$.
- g_a is an abstract expression corresponding to g_c, $g_c \sim g_a$,
- $eff_a = \langle stmt_a^1, \ldots, stmt_a^n \rangle$ is a sequence of tuples corresponding to the concrete effect $eff_c = \langle stmt_c^1, \ldots, stmt_c^n \rangle$, $eff_c \sim eff_a$, where $stmt_a^i$ is
  - $\langle \text{SEND}, sig_a, \langle e_a^1, \ldots, e_a^m \rangle, tgt_a \rangle$ when $stmt_c^i = \langle \text{SEND}, sig_c, \langle e_c^1, \ldots, e_c^m \rangle, tgt_c \rangle$, where sig_a is a corresponding abstract signal $sig_c \sim sig_a$, the parameters are abstract expressions $\forall 1 \leq j \leq m : e_c^j \sim e_a^j$, and the target is an abstract expression $tgt_c \sim tgt_a$,

– $\langle \textsc{assign}, lhs_a, rhs_a \rangle$ if $stmt_c^i = \langle \textsc{assign}, lhs_c, rhs_c \rangle$, where $lhs_c \sim lhs_a$, rhs_a is the abstract expression $rhs_c \sim rhs_a$,

– $\langle \textsc{assert}, e_a \rangle$ if $stmt_c^i = \langle \textsc{assert}, e_c \rangle$, where e_a is the abstract expression corresponding to the concrete expression e_c, $e_c \sim e_a$,

A set of abstract transitions $T_a = \{t_a^1, \ldots, t_a^n\}$ corresponds to a set of concrete transitions $T_c = \{t_c^1, \ldots, t_c^m\}$, if $n = m$, and $t_c^i \sim t_a^i$ for all $1 \leq i \leq n$.

An abstract state machine $sm_a = \langle s_i, S, T_a, \mathit{defers}_a, \mathit{flush} \rangle$ corresponds to the state machine $sm_c = \langle s_i, S, T_c, \mathit{defers}_c, \mathit{flush} \rangle$, $sm_c \sim sm_a$, if T_a is an abstract set of transitions corresponding to the set of transitions T_c, and $\forall s \in S : \mathit{defers}_c(s) \sim \mathit{defers}_a(s)$.

An abstract class $c_a = \langle \mathit{Vars}_a, sm_a \rangle$ corresponds to a class $c_c = \langle \mathit{Vars}_c, sm_c \rangle$, $c_c \sim c_a$, if $\mathit{Vars}_c \sim \mathit{Vars}_a$ and $sm_c \sim sm_a$. A set of abstract classes $\mathit{Classes}_a = \{c_a^1, \ldots, c_a^n\}$ corresponds to a set of concrete classes $\mathit{Classes}_c = \{c_c^1, \ldots, c_c^m\}$, if $n = m$, and $c_c^i \sim c_a^i$ for all $1 \leq i \leq n$.

3.2.4 Abstract Models

Let $M_c = \langle C_{\mathit{init}c}, D_c, \mathit{Classes}_c, O_c, \mathit{Locations}_c, \mathit{SysSigs}_c, \mathit{ExtSigs} \rangle$ be a concrete model. An abstract model $M_c = \langle C_{\mathit{init}a}, D_a, \mathit{Classes}_a, O_a, \mathit{Locations}_a, \mathit{SysSigs}_a, \mathit{ExtSigs} \rangle$ corresponds to the concrete model M_c, if

- Set of types $D_a = D_c \cup \mathit{abstTypes}$ contains abstract types,

- $\mathit{SysSigs}_a$ is a set of abstract system signals corresponding to the system signals $\mathit{SysSigs}_c$ in the concrete model.

- $\mathit{Classes}_a$ is a set of classes corresponding to the set of classes $\mathit{Classes}_c$ in the concrete model.

- O_a is a set of objects corresponding to the objects in the set O_c. For every $o_c = \langle c_c, \mathit{oid} \rangle \in O_c$, there is a unique corresponding object $o_a = \langle c_a, \mathit{oid} \rangle \in O_a$, such that $c_c \sim c_a$. There are no other objects in the set O_a.

- $\mathit{Locations}_a$ is a set of locations induced by O_a

- $C_{\mathit{init}a} = \langle \mathit{state}, \mathit{inputqueue}_a, \mathit{deferqueue}_a, \mathit{valuation}_a \rangle$ is an initial global configuration when $C_{\mathit{init}c} = \langle \mathit{state}, \mathit{inputqueue}_c, \mathit{deferqueue}_c, \mathit{valuation}_c \rangle$ is the initial global configuration of the concrete model, input and defer queues for all objects are empty in the initial configuration, and $\forall \langle o_a, \mathit{var}_a \rangle \in \mathit{Locations}_a$: $\mathit{valuation}_a(\langle o_a, \mathit{var}_a \rangle) = \mathit{coerce}(\mathit{valuation}_c(\langle o_c, \mathit{var}_c \rangle), d_c, d_a)$, where $o_c \sim o_a$, $d_c = \mathit{type}(\mathit{var}_c)$, $d_a = \mathit{type}(\mathit{var}_a)$, and $\mathit{var}_c \sim \mathit{var}_a$.

3.3 Counterexamples

Counterexamples are represented as counterexample traces describing executions violating a property in the model. Formally a counterexample trace is a sequence of tuples representing execution events of the form $\langle \textsc{trans}, F, \mathit{oid}, \mathit{tid} \rangle$, $\langle \textsc{impl}, F, \mathit{oid} \rangle$,

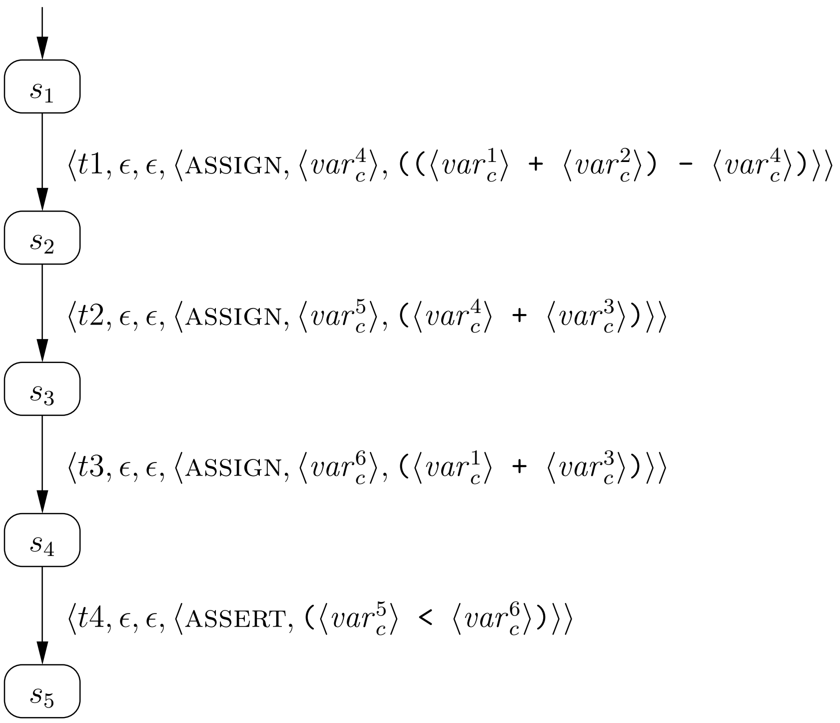

Figure 3.3: The concrete state machine

and $\langle \text{DEFER}, F, oid \rangle$. A tuple of the form $\langle \text{TRANS}, F, oid, tid \rangle$ represents the execution of a transition with transition identifier *tid* in the object having object identifier *oid*. A tuple of the form $\langle \text{IMPL}, F, oid \rangle$ represents implicit message consumption in the object with object identifier *oid* and a tuple of the form $\langle \text{DEFER}, F, oid \rangle$ represents deferring a message in the object with object identifier *oid*. F is the function solving the possible non-deterministic choices in the evaluation of the expressions. Every execution event has its own function for solving non-deterministic choices.

Example 3.1. Let $M_c = \langle C_{initc}, D_c, Classes_c, O_c, Locations_c, SysSigs_c, ExtSigs \rangle$ be a concrete model with one class $c_c = \langle Vars_c, sm_c \rangle$, $Classes_c = \{c_c\}$, and one object $o_c = \langle c_c, 1 \rangle$, $O_c = \{o_c\}$, which is instantiated from the class. The set of types in the model is $D_c = \{\texttt{int}, \texttt{boolean}, \texttt{reference}\}$. The class c_c has a set of variables $Vars_c = \{var_c^1, var_c^2, var_c^3, var_c^4, var_c^5, var_c^6\}$ and the type of each variable is `int`. A graphical representation of the state machine sm_c is shown in Figure 3.3. In the initial configuration $C_{initc} = \langle 1, state, inputqueue, deferqueue, valuation_c \rangle$, the active state of the only object is s_1, the queues are empty and the valuation gives $valuation_c(\langle o_c, var_c^1 \rangle) = 1$, $valuation_c(\langle o_c, var_c^2 \rangle) = -2$, $valuation_c(\langle o_c, var_c^3 \rangle) = 5$, $valuation_c(\langle o_c, var_c^4 \rangle) = 0$, $valuation_c(\langle o_c, var_c^5 \rangle) = 0$, and $valuation_c(\langle o_c, var_c^6 \rangle) = 0$. The model has no system or external signals.

A corresponding abstract model with all the variables abstracted with `Sign`-abstraction (see Section 3.1.1) is $M_a = \langle C_{inita}, D_a, Classes_a, O_a, Locations_a, SysSigs_a, ExtSigs \rangle$, where the set of types $D_a = \{\texttt{int}, \texttt{boolean}, \texttt{reference}, \texttt{Sign}\}$, the sets of system and external signals are empty, the set of classes $Classes_a = \{c_a\}$ contains one class $c_a = \langle \{var_a^1, var_a^2, var_a^3, var_a^4, var_a^5, var_a^6\}, sm_a \rangle$, the set of objects, $O_a = \{o_a\}$,

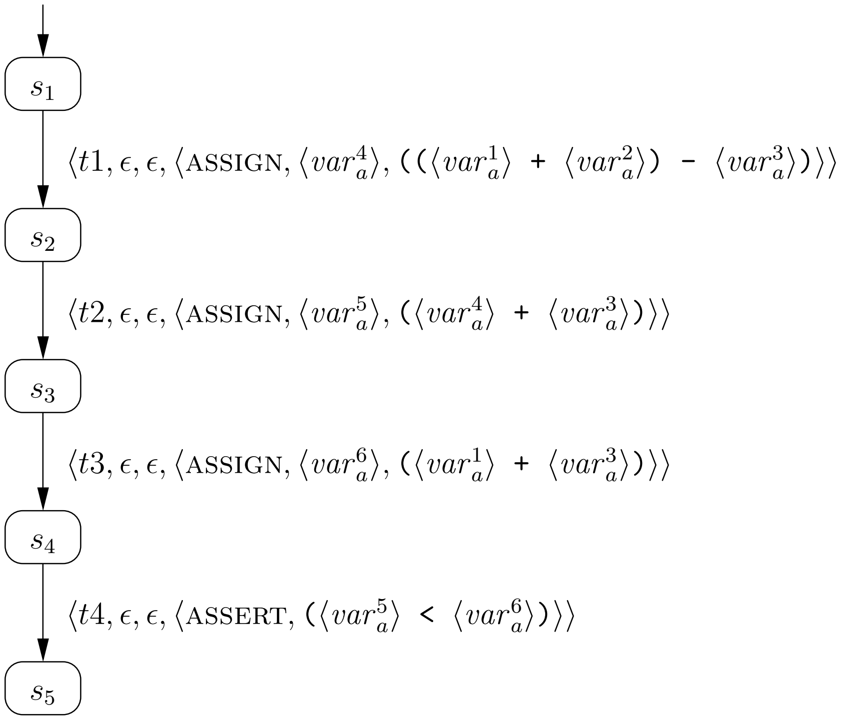

Figure 3.4: An abstract state machine

Table 3.1: Values of variables in the object o_a at different points of execution.

i	var_a^1	var_a^2	var_a^3	var_a^4	var_a^5	var_a^6
1	POS	NEG	POS	ZERO	ZERO	ZERO
2	POS	NEG	POS	POS	ZERO	ZERO
3	POS	NEG	POS	POS	POS	ZERO
4	POS	NEG	POS	POS	POS	POS

contains one object, $o_a = \langle c_a, 1 \rangle$, instantiated from the only class in the model, and in the initial configuration $C_{init\,a} = \langle 1, state, inputqueue, deferqueue, valuation_a \rangle$, the active state of the only object is s_1, the input and the defer queues are empty, and the valuation gives:

$$
\begin{aligned}
valuation_a(\langle o_a, var_c^1 \rangle) &= \texttt{POS} \\
valuation_a(\langle o_a, var_c^2 \rangle) &= \texttt{NEG} \\
valuation_a(\langle o_a, var_c^3 \rangle) &= \texttt{POS} \\
valuation_a(\langle o_a, var_c^4 \rangle) &= \texttt{ZERO} \\
valuation_a(\langle o_a, var_c^5 \rangle) &= \texttt{ZERO} \\
valuation_a(\langle o_a, var_c^6 \rangle) &= \texttt{ZERO}
\end{aligned}
$$

A graphical representation of the abstract state machine sm_a is shown in Figure 3.4.

The concrete model contains only one execution but because of the the over-

Table 3.2: Values of variables in the object o_c at different points of execution.

i	var_c^1	var_c^2	var_c^3	var_c^4	var_c^5	var_c^6
1	1	-2	5	0	0	0
2	1	-2	5	-6	0	0
3	1	-2	5	-6	-11	0
4	1	-2	5	-6	-11	6

approximative abstraction the abstract model contains several executions. Let an abstract counterexample trace be:

$$\langle\langle \text{TRANS}, F_1, 1, t1\rangle, \langle \text{TRANS}, F_2, 1, t2\rangle, \langle \text{TRANS}, F_3, 1, t3\rangle, \langle \text{TRANS}, F_4, 1, t4\rangle\rangle$$

Functions F_i, $1 \leq i \leq 4$, fixes the non-deterministic choices made in the execution in such a way that the assertion in transition with identifier $t4$ fails in the abstract model. Table 3.1 shows the values of the variables in the object o_a before the ith event of the counterexample trace is executed . When the trace is executed in the concrete model, the assertion holds. Thus the counterexample trace is spurious. Table 3.2 shows the values of the variables in the object o_c in states s_i at the execution of the counterexample trace. ■

Chapter 4

Feasibility Analysis

To determine the feasibility of an abstract counterexample, we use stepwise simulation of the concrete model to try to execute the events fixed by the abstract counterexample on the concrete model in the same order in which they appear in the abstract counterexample [33]. Because every concrete model always has a fixed initial configuration, there are no external signals with parameters coming from the environment, and each event determines via the object identifier the object where the event is executed, the steps to be taken in the simulation are completely determined by the events fixed by the abstract counterexample.

If all the events in the counterexample can be simulated successfully without assertion failures or implicit message consumptions, then the counterexample is spurious because the property that was expected to be violated was not violated in the simulation of the counterexample.

4.1 Assertion Failures

If the property violation in the abstract model was an assertion failure, the execution of a feasible counterexample in the concrete model results to an assertion failure. Then the simulation of the execution of a transition of a state machine results in a violation of an assertion in the concrete model, we have found an error in the concrete model and have a counterexample to demonstrate it.

Assertion failures in the concrete model can also occur unintentionally, i.e. when the counterexample was supposed to demonstrate implicit message consumption, message deferral, or failure of another assertion. This happens when an assertion executed prior to the intended property violation holds in the abstract model (due to the extra behavior introduced by the abstraction) but fails in the concrete model. For example, let variables var_c^1 and var_c^2 have values 1 and 2, respectively, and let the corresponding abstract variables var_a^1 and var_a^2 be abstracted with expression `Sign`-abstraction and have values `POS` and `POS`, respectively. When a concrete expression `(`$\langle var_c^1 \rangle$ `==` $\langle var_c^2 \rangle$`)` is evaluated the result is false, but when the corresponding abstract expression `(`$\langle var_a^1 \rangle$ `==` $\langle var_a^2 \rangle$`)` is evaluated the result can be either true or false. If these expressions are the conditions in corresponding assertion statements we could very well have a trace where an assertion fails in the concrete model but holds in the abstract model. From the user's point of view it is not important which kind of failure the original counterexample was supposed to demonstrate; in any case we have a true counterexample which leads to the violation of an assertion.

4.2 Implicit Message Consumptions

We assume that implicit message consumptions are always forbidden and thus an implicit message consumption executed in the concrete model yields a true counterexample. When the algorithm for feasibility checking is introduced in Chapter 4.4, we shall describe changes needed to the algorithm to disable checking against implicit message consumptions.

4.3 Action Not Enabled

If the action to be executed next is not enabled in the current configuration of the concrete model the counterexample is spurious.

If the action is the execution of a transition, there are two possible reasons why the transition is not enabled:

- The guard condition of the transition executed next in the abstract counterexample evaluates to false in the current configuration of the concrete model, or
- the transition to be executed next (on some object in the concrete model) is not enabled because there is no message with the correct signal in the head of the object's input queue. The contents of the corresponding message queues can differ because the abstraction may introduce non-determinism to the targets of send statements. For example, constructs like ($condition$?o_1:o_2) can easily cause this kind of situation.

If the action is an implicit consumption or a deferral of a message, then the reason why the action is not enabled can be either of the two cases:

- There are no messages in the input queue. The corresponding input queue in the abstract model however has at least one message because otherwise the counterexample trace could not have such an event at this point. The reason for this kind of situation is always a send statement in the abstract model sending a message to an object which does not correspond to the target object of the corresponding send statement in the concrete model. If all send statements in the concrete and abstract model had sent the messages to corresponding objects, the queues in all objects would contain the same number of messages.
- There is a transition enabled in the concrete model in the object where the event should have been executed. The corresponding transition in the abstract model is not enabled because otherwise there could not be an implicit consumption or a message deferral event in the counterexample trace. The reason for the transition to be not enabled in the abstract model can be either input queue with no message matching the trigger in the head of the queue or a guard that

1: **function** IS_COUNTEREXAMPLE_FEASIBLE(M_c, *Trace*)
2: # $M_c = \langle C_{init\,c}, D_c, Classes_c, O_c, Locations_c, SysSigs_c, ExtSigs \rangle$
3: $C \leftarrow C_{init\,c}$
4: **while** $|Trace| > 0$ **do**
5: $event \leftarrow head(Trace)$
6: $Trace \leftarrow tail(Trace)$
7: **if** *event* is of the form $\langle \text{TRANS}, F, oid, tid \rangle$ **then**
8: $o \leftarrow$ the object with id *oid* in O_c
9: $t \leftarrow$ the transition with id *tid* in $statemachine(class(o))$
10: $a \leftarrow \langle \text{TRANS}, F, o, t \rangle$
11: **else** # $event = \langle etype, F, oid \rangle$ for some *etype* in $\{\text{IMPL}, \text{DEFER}\}$
12: $o \leftarrow$ the object with id *oid* in O_c
13: $a \leftarrow \langle etype, o \rangle$
14: **if** $enabled(C, a)$ **then**
15: **if** a is of the form $\langle \text{IMPL}, o \rangle$ **then**
16: **return** *true*
17: $C \leftarrow exec(C, a)$
18: **if** $C = \perp$ **then**
19: **return** *true*
20: **else**
21: **return** *false*
22: **return** *false*

Figure 4.1: Algorithm IS_COUNTEREXAMPLE_FEASIBLE checks whether the counterexample trace *Trace* from abstract model M_a is feasible in the concrete model M_c.

evaluates to false. If the object cannot receive message with a correct signal, then we have a pair of corresponding message send statements that have sent their messages to non-corresponding objects, just like in the case with an empty input queue.

4.4 Algorithm for Checking Feasibility

Algorithm IS_COUNTEREXAMPLE_FEASIBLE in Figure 4.1 checks whether a counterexample is feasible. It takes the concrete model M_c and the counterexample trace *Trace* whose feasibility is to be checked and returns *true* if the counterexample is feasible, otherwise *false* is returned.

The algorithm tries to execute the trace step by step. In lines 7–13 the algorithm forms the event a to be executed. The object identifier *oid* can be mapped unambiguously to the object o because there are no two different objects with the same object identifier in the model. Respectively transition identifier *tid* can be mapped unambiguously to the transition t. In lines 14–19 the event is executed if it is enabled in the concrete model. The if-clause in line 15 checks whether we have a feasible counterexample demonstrating an implicit message consumption. If we do not want to check against implicit consumptions lines 15–16 should be removed from the algorithm. After we have checked whether the event is an implicit consumption the event is executed in line 17 and checked whether an assertion failed in the execution of the event. If the event was not enabled in the concrete model then the counterexample is not feasible and *false* is returned in line 21. Finally, if the trace was successfully executed in the concrete model, then the counterexample is spurious and *false* is returned.

Example 4.1. In the feasibility analysis for abstract counterexample introduced in Example 3.1 the first three events are executed successfully in the concrete model. Also the execution of the fourth event, the transition with the assertion action, succeeds without runtime errors. Thus the abstract counterexample trace is executed in the concrete model without property violations and the abstract counterexample is spurious. ■

Chapter 5

Counterexample Analysis

Finding a spurious counterexample in the abstract model prevents us from either verifying that the properties hold in the concrete model or that there is a counterexample which demonstrates a failing property. To be able to continue the model checking procedure we need to refine the abstraction in order to remove the spurious counterexample. A good refinement removes the spurious counterexample (and possibly other spurious counterexamples which we have not encountered yet) but still avoids the state explosion problem.

First we identify the parts in the model that contribute to the existence of the spurious counterexample by identifying the *relevant locations*, i.e. for each point in the trace the locations that contribute to the existence of the counterexample. The algorithm for doing this is described in Section 5.2. It is intuitive that the refinement is done to the parts of the model that are relevant to the existence of the spurious counterexample. In Section 5.3 use relevant locations as a guide for calculating a refinement for models abstracted with *interval abstractions* (a subset of data abstractions).

To facilitate the discussion of the counterexample analysis, we introduce a construction for *analysis traces*. An analysis trace contains all the information needed in the counterexample analysis from the spurious counterexample and its execution in the concrete and in the abstract model. Analysis traces are described in Section 5.1.

5.1 Forming an Analysis Trace

Before the analysis methods are described we form an *analysis trace* by executing the counterexample trace in both the concrete and the abstract models. The purpose of the analysis trace is to bundle all the relevant information needed for the analysis into one sequence for easier accessibility.

The analysis trace needs to have a finer granularity than the execution traces because the analysis we are going to do needs to distinguish the effects of different parts (actions) of the transitions. The analysis trace is formed by the algorithm TRANSFORM and is a sequence of tuples of the form $\langle \widetilde{C}, \langle sn, o_c, o_a, \widetilde{b}\rangle, \widetilde{C}'\rangle$, where

- $\widetilde{C} = \langle C_c, C_a\rangle$ is a combined configuration, a tuple containing the configuration of the concrete and the abstract model before execution of the action,

- $sn = sn(C_c) = sn(C_a)$ is a sequence number of the concrete and the abstract configurations before the execution of the action,
- o_c is the concrete model object in which the action is executed,
- o_a is the abstract model object in which the action is executed,
- $\widetilde{b}$ is the tuple describing the type the action describes (these are specified below), and
- $\widetilde{C}' = \langle C'_c, C'_a \rangle$ is a tuple containing the configuration of the concrete and the abstract model after the execution of the action. If either C'_c or C'_a is $\perp$, then $\widetilde{C}' = \perp$. This represents the point where the executions in the concrete and in the abstract model differ so much that we can not execute the trace further in either one of them.

Component $\widetilde{b}$ in the tuple describes what kind of action the tuple represents. $\widetilde{b}$ can represent one of the following things: the reception of a message (generated from a trigger of an executed transition), assuming of a boolean expression to be true (generated from a guard of an executed transition), an assertion, an assignment, the sending of a message, an implicit message consumption, and the deferral of a message.

Reception of messages by corresponding triggers $\langle sig_c, \langle p_c^1, \ldots, p_c^m \rangle\rangle$ in the concrete model and $\langle sig_a, \langle p_a^1, \ldots, p_a^m \rangle\rangle$ in the abstract model are encoded as a tuple $\widetilde{b} = \langle \text{RECV}, sig_c, sig_a, \langle p_c^1, \ldots, p_c^m \rangle, \langle p_a^1, \ldots, p_a^m \rangle\rangle$.

Guards of transitions are not included in the analysis trace unless a guard is the reason for the counterexample to be spurious. If a guard is the reason for the counterexample to be spurious then the guard is encoded as a tuple $\widetilde{b} = \langle \text{ASSUME}, F, g_c, g_a \rangle$ describing a boolean expression that is assumed to evaluate to true when g_c is the guard of the transition in the concrete model and g_a is the corresponding guard in the abstract model.

Sending a message by corresponding statements $\langle \text{SEND}, sig_c, \langle e_c^1, \ldots, e_c^q \rangle, tgt_c \rangle$ in the concrete model and $\langle \text{SEND}, sig_a, \langle e_a^1, \ldots, e_a^q \rangle, tgt_a \rangle$ in the abstract model are encoded as a tuple $\widetilde{b} = \langle \text{SEND}, F, sig_c, sig_a, \langle e_c^1, \ldots, e_c^q \rangle, \langle e_a^1, \ldots, e_a^q \rangle, tgt_c, tgt_a \rangle$.

Corresponding assignments by statements $\langle \text{ASSIGN}, lhs_c, rhs_c \rangle$ in the concrete model and $\langle \text{ASSIGN}, lhs_a, rhs_a \rangle$ in the abstract model are encoded as a tuple $\widetilde{b} = \langle \text{ASSIGN}, F, lhs_c, lhs_a, rhs_c, rhs_a \rangle$.

Corresponding assertions $\langle \text{ASSERT}, e_c \rangle$ in the concrete model and $\langle \text{ASSERT}, e_a \rangle$ in the abstract model are encoded as a tuple $\widetilde{b} = \langle \text{ASSERT}, F, e_c, e_a \rangle$.

Implicit consumptions and deferrals of messages are encoded as tuples $\langle \text{IMPL} \rangle$ and $\langle \text{DEFER} \rangle$, respectively.

The algorithm TRANSFORM is represented in Figure 5.1. The algorithm calls, for every event in the counterexample trace, a function TF_TRANS for handling the execution of transitions or TF_IMPL_DEFER for handling implicit consumptions or message deferrals. These functions execute actions associated with the event and return a new version of the analysis trace L with components produced from the associated actions added. The function $lastconf(L)$ returns the last combined configuration from the analysis trace L, i.e. the last element of the last tuple in the sequence L. When

```
function TRANSFORM(M_c, M_a, Trace)
    # M_c = ⟨C_initc, D_c, Classes_c, O_c, Locations_c, SysSigs_c, ExtSigs⟩
    # M_a = ⟨C_inita, D_a, Classes_a, O_a, Locations_a, SysSigs_a, ExtSigs⟩
    Trace = ⟨event_1, event_2, ..., event_n⟩
    L ← ⟨⟩
    C̃ ← ⟨C_initc, C_inita⟩
    for i = 1 to n do
        if event_i is of the form ⟨TRANS, F, oid, tid⟩ then
            o_c ← the object with id oid in O_c
            o_a ← the object with id oid in O_a
            t_c ← the transition with id tid in statemachine(class(o_c))
            t_a ← the transition with id tid in statemachine(class(o_a))
            L ← TF_TRANS(M_c, M_a, L, C̃, F, o_c, o_a, t_c, t_a)
        else # event_i is of the form ⟨etype, F, oid⟩, where etype ∈ {IMPL, DEFER}
            o_c ← the object with id oid in O_c
            o_a ← the object with id oid in O_a
            L ← TF_IMPL_DEFER(M_c, M_a, L, C̃, F, etype, o_c, o_a)
        if lastconf(L) = ⊥ then
            return L
```

Figure 5.1: The algorithm TRANSFORM forms an analysis trace L from a spurious counterexample *Trace* by simulating the execution in the concrete model M_c and in the abstract model M_a.

the analysis trace has $\perp$ as its last combined configuration we have reached the point of execution where the counterexample trace cannot be executed further either in the concrete or in the abstract model. This means that we have all the information we need for analysis and the analysis trace is returned.

5.1.1 Transition Execution Events

The algorithm TF_TRANS in Figure 5.2 adds execution steps produced from the transition execution counterexample event to the analysis trace and returns the resulting analysis trace. The algorithm is divided to two separate parts. The first, starting from line 5, handles events that are enabled in the concrete model[1] and the latter one, starting from line 28, events that are not enabled in the concrete model.

Enabled transitions are executed component by component in both the concrete and the abstract model until either all components are executed or an assertion has failed and the analysis trace to be returned ends with the combined configuration $\perp$. The algorithm uses the function EXECUTE_ACTION for the actual execution. The pseudocode for the function EXECUTE_ACTION can be found in Figure 5.4.

The processing of enabled transition events starts with receiving messages by possible triggers. Triggers are executed in the code following line 6. Guards are skipped in

[1]Events are always enabled in the abstract model because the counterexample trace was generated from the abstract model

```
function TF_TRANS(M_c, M_a, L, \tilde{C}, F, o_c, o_a, t_c, t_a)
    # \tilde{C} = ⟨C_c, C_a⟩
    # t_c = ⟨tid, s_1, s_2, trig_c, g_c, ⟨stmt_c^1, ..., stmt_n⟩⟩
    # t_a = ⟨tid, s_1, s_2, trig_a, g_a, ⟨stmt_a^1, ..., stmt_n⟩⟩
    if enabled(C_c, ⟨TRANS, F, o_c, t_c⟩) then
        if trig_c ≠ ϵ then
            # trig_c = ⟨sig_c, ⟨p_c^1, ..., p_c^m⟩⟩ and trig_a = ⟨sig_a, ⟨p_a^1, ..., p_a^m⟩⟩
            \tilde{b} ← ⟨RECV, sig_c, sig_a, ⟨p_c^1, ..., p_c^m⟩, ⟨p_a^1, ..., p_a^m⟩⟩
            L ← append(L, EXECUTE_ACTION(\tilde{C}, o_c, o_a, F, \tilde{b}))
            \tilde{C} ← lastconf(L)
        for i = 1 to n do
            if stmt_c^i is of the form ⟨SEND, sig_c, ⟨e_c^1, ..., e_c^q⟩, tgt_c⟩ then
                # stmt_a^i is of the form ⟨SEND, sig_a, ⟨e_a^1, ..., e_a^q⟩, tgt_a⟩
                \tilde{b} ← ⟨SEND, F, sig_c, sig_a, ⟨e_c^1, ..., e_c^q⟩, ⟨e_a^1, ..., e_a^q⟩, tgt_c, tgt_a⟩
            else if stmt_c^i is of the form ⟨ASSIGN, lhs_c, rhs_c⟩ then
                # stmt_a^i is of the form ⟨ASSIGN, lhs_a, rhs_a⟩
                \tilde{b} ← ⟨ASSIGN, F, lhs_c, lhs_a, rhs_c, rhs_a⟩
            else if stmt_c^i is of the form ⟨ASSERT, e_c⟩ then
                # stmt_a^i is of the form ⟨ASSERT, e_a⟩
                \tilde{b} ← ⟨ASSERT, F, e_c, e_a⟩
            L ← append(L, EXECUTE_ACTION(\tilde{C}, o_c, o_a, F, \tilde{b}))
            \tilde{C} ← lastconf(L)
            if \tilde{C} = ⊥ then return L
        # statemachine(class(o_c)) = ⟨s_i, S, T, defers, flush⟩
        if s_2 ∈ flush then
            return append(L, EXECUTE_ACTION(\tilde{C}, o_c, o_a, F, ⟨GOTO, s_2, true⟩))
        return append(L, EXECUTE_ACTION(\tilde{C}, o_c, o_a, F, ⟨GOTO, s_2, false⟩))
    else
        if trig_c ≠ ϵ then
            # trig_c = ⟨sig_c, ⟨p_c^1, ..., p_c^m⟩⟩ and trig_a = ⟨sig_a, ⟨p_a^1, ..., p_a^m⟩⟩
            \tilde{b} ← ⟨RECV, sig_c, sig_a, ⟨p_c^1, ..., p_c^m⟩, ⟨p_a^1, ..., p_a^m⟩⟩
            if InputQueue(C_c, o_c) = ⟨⟩ then
                return append(L, ⟨\tilde{C}, ⟨sn(C_c), o_c, o_a, \tilde{b}⟩, ⊥⟩)
            # InputQueue(C_c, o_c) = ⟨⟨id_{msg_c}, sig'_c, v_c^1, ..., v_c^q⟩, ...⟩
            # InputQueue(C_a, o_a) = ⟨⟨id_{msg_a}, sig_a, v_a^1, ..., v_a^q⟩, ...⟩
            if sig_c ≠ sig'_c then
                return append(L, ⟨\tilde{C}, ⟨sn(C_c), o_c, o_a, \tilde{b}⟩, ⊥⟩)
            L ← append(L, EXECUTE_ACTION(\tilde{C}, o_c, o_a, F, \tilde{b}))
            \tilde{C} ← lastconf(L)
        return append(L, ⟨\tilde{C}, ⟨sn(C_c), o_c, o_a, ⟨ASSUME, F, g_c, g_a⟩⟩, ⊥⟩)
```

Figure 5.2: Algorithm for producing analysis trace steps from the transition execution events in the counterexample trace.

enabled transitions because they do not alter the configuration. Effects are executed in the for-clause starting from line 11. First the type of the component is parsed and a tuple $\widetilde{b}$ representing the component is formed in lines 12–20. Then it is executed in the abstract and in the concrete model with the function EXECUTE_ACTION. Assertion failures are checked in line 23. After all parts of the effect are processed we move the state machine to the target state s_2 of the transition by executing a GOTO action. This also flushes the defer queue if the target state s_2 is an element of the set *flush*.

If the transition execution event is not enabled in the concrete model, then either the trigger of the transition can not be executed in the concrete model, or the guard of the transition evaluates to false. If the transition has a trigger, we analyze in lines 29–37 whether the transition is not enabled in the concrete model because the trigger can not be executed. The trigger can not be executed if the input queue is either empty or if the type of the message at the head of the queue does not match the type of the signal in the trigger. Otherwise the message is received by execution of the EXECUTE_ACTION function in line 38. If there is no trigger or the trigger can be executed without errors, the guard evaluates to false in the concrete model. Then we add an assume step with an ASSUME action to the analysis trace and return.

5.1.2 Implicit Consumptions and Message Deferrals

The algorithm TF_IMPL_DEFER in Figure 5.3 adds execution steps produced from the implicit consumption or message deferral counterexample event to the analysis trace and returns the resulting analysis trace. The algorithm is divided into two separate parts just like the TF_TRANS algorithm above.

If the event is enabled in the concrete model then we execute the event by calling EXECUTE_ACTION and return the produced trace.

The event might not be enabled in the concrete model because of two reasons. Firstly, the input queue of the object where the event is to be executed might be empty. In the algorithm this is handled in the if-clause starting from line 6 by adding a step corresponding to the type of the event that results in a combined configuration $\perp$.

Secondly, there might be a transition enabled in the object where the event is to be executed. In that case we have to analyze whether the reason for the corresponding transition to be not enabled in the abstract model lies in the trigger part or in the guard part of the transition. If the transitions (the concrete and the corresponding abstract one) share a non-empty trigger, then the reason for the abstract transition to be not enabled can be determined by checking the signals of messages in front of both, the concrete and the abstract, message queues. If the signals are not corresponding ones, then the reason is in the trigger parts and we return an analysis trace with message receiving action resulting in the combined configuration $\perp$ added to the end of the trace. The signal correspondence check is done in line 19 in the algorithm. If the signals in the messages are corresponding then the reason for the abstract transition to be not enabled lies in the guard part because the enabledness of the transition is determined by the trigger and guard parts, not by the effect part. If the reason lies in the guard part then a step corresponding to the triggers of the transitions is added to the analysis trace and after that an ASSUME step resulting in the combined

```
**function** TF_IMPL_DEFER($M_c, M_a, L, \widetilde{C}, F, etype, o_c, o_a$)
    # $\widetilde{C} = \langle C_c, C_a \rangle$
    **if** $enabled(C_c, \langle etype, o_c \rangle)$ **then**
        **return** $L \leftarrow append(L,$ EXECUTE_ACTION$(\widetilde{C}, o_c, o_a, F, \langle etype \rangle))$
    **else**
        **if** $InputQueue(C_c, o_c) = \langle\rangle$ **then**
            **return** $append(L, \langle \widetilde{C}, \langle sn(C_c), o_c, o_a, \langle etype \rangle \rangle, \bot \rangle)$
        **else**
            # There has to be enabled transition $t_c$ in the concrete model.
            let $t_c$ be some transition enabled in $C_c$ in $M_c$
            # $t_c = \langle tid, s_1, s_2, trig_c, g_c, eff_c \rangle$
            # $t_c \sim t_a = \langle tid, s_1, s_2, trig_a, g_a, eff_a \rangle$
            **if** $trig_c \neq \epsilon$ **then**
                # $trig_c = \langle sig_c, \langle p_c^1, \ldots, p_c^m \rangle \rangle$
                # $trig_a = \langle sig_a, \langle p_a^1, \ldots, p_a^m \rangle \rangle$
                # $InputQueue(C_c, o_c) = \langle \langle id_{msg_c}, sig_c, v_c^1, \ldots, v_c^q \rangle, ... \rangle$
                # $InputQueue(C_a, o_a) = \langle \langle id_{msg_a}, sig'_a, v_a^1, \ldots, v_a^q \rangle, ... \rangle$
                $\widetilde{b} \leftarrow \langle$RECV$, sig_c, sig_a, \langle p_c^1, \ldots, p_c^m \rangle, \langle p_a^1, \ldots, p_a^m \rangle \rangle$
                **if** $sig_a \neq sig'_a$ **then**
                    **return** $append(L, \langle \widetilde{C}, \langle sn(C_c), o_c, o_a, \widetilde{b} \rangle, \bot \rangle)$
                $L \leftarrow append(L,$ EXECUTE_ACTION$(\widetilde{C}, o_c, o_a, F, \widetilde{b}))$
                $\widetilde{C} \leftarrow lastconf(L)$
            **return** $append(L, \langle \widetilde{C}, \langle sn(C_c), o_c, o_a, \langle$ASSUME$, F, g_c, g_a \rangle \rangle, \bot \rangle)$
```

Figure 5.3: Algorithm for producing analysis trace steps from the implicit consumption and message defer events in the counterexample trace.

configuration $\bot$ is added to represent the guards that are evaluated with different results.

5.1.3 Execution of Actions for Analysis Trace

We have extracted the actual execution of actions in the concrete and in the abstract model to a separate algorithm in the construction of an analysis trace. The algorithm in Figure 5.4 takes a combined configuration, a concrete object, an abstract object, a function for solving non-determinism, and information about the type of the action as arguments and returns an analysis trace step where the action is executed. The algorithm is very straightforward and is here only to shorten the previous, more complicated, algorithms.

function $\text{EXECUTE\_ACTION}(\widetilde{C}, o_c, o_a, F, \widetilde{b})$
    # $\widetilde{C} = \langle C_c, C_a \rangle$
    if $\widetilde{b}$ is of the form $\langle \text{RECV}, sig_c, sig_a, \langle p_c^1, \ldots, p_c^n \rangle, \langle p_a^1, \ldots, p_a^n \rangle \rangle$ **then**
        $C_c' \leftarrow exec_{recv}(C_c, o_c, \langle sig_c, \langle p_c^1, \ldots, p_c^n \rangle \rangle)$
        $C_a' \leftarrow exec_{recv}(C_a, o_a, \langle sig_a, \langle p_a^1, \ldots, p_a^n \rangle \rangle)$
        $\widetilde{C}' \leftarrow \langle C_c', C_a' \rangle$
    else if $\widetilde{b}$ is of the form $\langle \text{SEND}, F, sig_c, sig_a, E_c, E_a, tgt_c, tgt_a \rangle$ **then**
        # E_c is of the form $\langle e_c^1, \ldots, e_c^n \rangle$ and E_a is of the form $\langle e_a^1, \ldots, e_a^n \rangle$
        $C_c' \leftarrow exec_{eff}(C_c, F, o_c, \langle \text{SEND}, sig_c, \langle e_c^1, \ldots, e_c^n \rangle, tgt_c \rangle)$
        $C_a' \leftarrow exec_{eff}(C_a, F, o_a, \langle \text{SEND}, sig_a, \langle e_a^1, \ldots, e_a^n \rangle, tgt_a \rangle)$
        $\widetilde{C}' \leftarrow \langle C_c', C_a' \rangle$
    else if $\widetilde{b}$ is of the form $\langle \text{ASSIGN}, F, lhs_c, lhs_a, rhs_c, rhs_a \rangle$ **then**
        $C_c' \leftarrow exec_{eff}(C_c, F, o_c, \langle \text{ASSIGN}, lhs_c, rhs_c \rangle)$
        $C_a' \leftarrow exec_{eff}(C_a, F, o_a, \langle \text{ASSIGN}, lhs_a, rhs_a \rangle)$
        $\widetilde{C}' \leftarrow \langle C_c', C_a' \rangle$
    else if $\widetilde{b}$ is of the form $\langle \text{ASSERT}, F, e_c, e_a \rangle$ **then**
        # Assertion has to hold in the concrete model because
        # the counterexample was spurious
        $C_c' \leftarrow exec_{eff}(C_c, F, o_c, \langle \text{ASSERT}, e_c \rangle)$
        $C_a' \leftarrow exec_{eff}(C_a, F, o_a, \langle \text{ASSERT}, e_a \rangle)$
        if $C_a' = \perp$ **then**
            $\widetilde{C}' \leftarrow \perp$
        else
            $\widetilde{C}' \leftarrow \langle C_c', C_a' \rangle$
    else if $\widetilde{b}$ is of the form $\langle \text{GOTO}, s, b \rangle$ **then**
        $C_c' \leftarrow exec_{goto}(C_c, o_c, s, b)$
        $C_a' \leftarrow exec_{goto}(C_a, o_a, s, b)$
        $\widetilde{C}' \leftarrow \langle C_c', C_a' \rangle$
    else if $\widetilde{b}$ is of the form $\langle \text{IMPL} \rangle$ **then**
        $\widetilde{C}' \leftarrow \langle exec(C_c, \langle \text{IMPL}, o_c \rangle), exec(C_a, \langle \text{IMPL}, o_a \rangle) \rangle$
    else if $\widetilde{b}$ is of the form $\langle \text{DEFER} \rangle$ **then**
        $\widetilde{C}' \leftarrow \langle exec(C_c, \langle \text{DEFER}, o_c \rangle), exec(C_a, \langle \text{DEFER}, o_a \rangle) \rangle$
    return $\langle \widetilde{C}, \langle sn(C_c), o_c, o_a, \widetilde{b} \rangle, \widetilde{C}' \rangle$

Figure 5.4: Algorithm for creating analysis trace steps.

Table 5.1: Values of variables in the object o_a in global configurations C_a^i.

i	var_a^1	var_a^2	var_a^3	var_a^4	var_a^5	var_a^6
1	$\underline{\texttt{POS}}$	$\underline{\texttt{NEG}}$	$\underline{\texttt{POS}}$	ZERO	ZERO	ZERO
2	POS	NEG	$\underline{\texttt{POS}}$	$\underline{\texttt{POS}}$	ZERO	ZERO
3	POS	NEG	POS	POS	$\underline{\texttt{POS}}$	ZERO
4	POS	NEG	POS	POS	$\underline{\texttt{POS}}$	POS

Table 5.2: Values of variables in the object o_c in global configurations C_c^i.

i	var_c^1	var_c^2	var_c^3	var_c^4	var_c^5	var_c^6
1	$\underline{1}$	$\underline{-2}$	$\underline{5}$	0	0	0
2	1	-2	$\underline{5}$	$\underline{-6}$	0	0
3	1	-2	5	-6	$\underline{-11}$	0
4	1	-2	5	-6	$\underline{-11}$	6

5.2 Relevant Locations

One way to analyze the counterexample is to isolate variables in objects in the abstract model that can actually affect the execution of the action whose corresponding action in the concrete model could not be executed. Affecting variables can be different at different points of the trace.

Example 5.1. In the abstract model introduced in Example 3.1, the spurious counterexample was produced because the assertion fails in the abstract model even though the corresponding assertion in the concrete model holds. An analysis trace produced from the counterexample trace, the concrete model, and the abstract model is

$$\langle\langle\langle C_c^1, C_a^1\rangle, \langle 1, o_c, o_a, \widetilde{b_1}\rangle, \langle C_c^2, C_a^2\rangle\rangle, \langle\langle C_c^2, C_a^2\rangle, \langle 2, o_c, o_a, \widetilde{b_2}\rangle, \langle C_c^3, C_a^3\rangle\rangle,$$
$$\langle\langle C_c^3, C_a^3\rangle, \langle 3, o_c, o_a, \widetilde{b_3}\rangle, \langle C_c^4, C_a^4\rangle\rangle, \langle\langle C_c^4, C_a^4\rangle, \langle 4, o_c, o_a, \widetilde{b_4}\rangle, \perp\rangle\rangle,$$

where

$$\begin{aligned}
\widetilde{b_1} &= \langle \text{ASSIGN}, F_1, \langle var_c^4\rangle, \langle var_a^4\rangle, \texttt{((}\langle var_c^1\rangle \texttt{ + } \langle var_c^2\rangle\texttt{) - }\langle var_c^3\rangle\texttt{)}, \\
&\quad \texttt{((}\langle var_a^1\rangle \texttt{ + } \langle var_a^2\rangle\texttt{) - }\langle var_a^3\rangle\texttt{)}\rangle \\
\widetilde{b_2} &= \langle \text{ASSIGN}, F_2, \langle var_c^5\rangle, \langle var_a^5\rangle, \texttt{(}\langle var_c^4\rangle \texttt{ + } \langle var_c^3\rangle\texttt{)}, \texttt{(}\langle var_a^4\rangle \texttt{ + } \langle var_a^3\rangle\texttt{)}\rangle \\
\widetilde{b_3} &= \langle \text{ASSIGN}, F_3, \langle var_c^6\rangle, \langle var_a^6\rangle, \texttt{(}\langle var_c^1\rangle \texttt{ + } \langle var_c^3\rangle\texttt{)}, \texttt{(}\langle var_a^1\rangle \texttt{ + } \langle var_a^3\rangle\texttt{)}\rangle \\
\widetilde{b_4} &= \langle \text{ASSERT}, F_4, \texttt{(}\langle var_c^5\rangle \texttt{ < } \langle var_c^6\rangle\texttt{)}, \texttt{(}\langle var_a^5\rangle \texttt{ < } \langle var_a^6\rangle\texttt{)}\rangle
\end{aligned}$$

The values of the variables in different points of the analysis trace are shown in Table 5.1 and in Table 5.2. The assertion fails in the abstract model because the value of the variable var_a^5 does not correspond to the value of the corresponding variable var_c^5. On the other hand the second action in the trace assigns values to variable var_a^5

and its corresponding variable var_c^5. Therefore the values of these variables before the action do not affect the feasibility of the counterexample but the values of var_c^4, var_a^4, var_c^3, and var_a^3 do. Similarly the first action assigns the value calculated from the values of var_a^1, var_a^2, and var_a^3 to var_a^4 in the abstract model and the value calculated from the values of var_c^1, var_c^2, and var_c^3 to var_c^4 in the concrete model. In Table 5.1 and in Table 5.2 the values affecting the occurrence of the spurious counterexample are underlined. Ultimately our objective is to remove the spurious counterexample with refinement of the types of a subset of variables affecting the occurrence of the spurious counterexample. ■

To put this observation into use we introduce the concept of a *relevant location.* Relevant locations describe a set of components affecting the occurrence of the spurious counterexample at each point in the trace. We have two types of relevant locations, relevant locations concerning variables and relevant locations concerning message parameters, which are both represented with tuples containing three elements.

A relevant location concerning a variable in an object is a tuple $\langle sn, oid, name\rangle$ where sn is a sequence number indicating the concrete and abstract global configurations relating to this entry, oid is the object identifier of the object this entry is related to, and $name$ is the name of the variable this entry is related to. The actual location in the model can be found by converting the object identifier and the variable name to the corresponding object and variable.

Example 5.2. In example 5.1, there is a relevant location $\langle 2, id(o_c), name(var_c^4)\rangle$ (note that $id(o_c) = id(o_a)$ and $name(var_c^4) = name(var_a^4)$) meaning that the values of variables var_c^4 and var_a^4 in objects o_c and o_a, are relevant in the global configurations C_c^2 and C_a^2, respectively. ■

A relevant location concerning a parameter in a message is a tuple $\langle sn, id_{msg}, i\rangle$ where sn is a sequence number indicating the global configurations relating to this entry, id_{msg} is a message identifier identifying the message this entry is related to, and i is an index identifying the ith parameter in the message as the parameter this entry is related to.

Our concept of relevant locations includes also message parameters and thus here the meaning of the word location corresponds to the meaning of the word location in Chapter 2 only when speaking about variables in objects.

In practice relevant locations are found by analyzing first the action which could not be executed in the concrete model. This action is found from the last step in the analysis trace. The analysis of the last action is described in Section 5.2.1. The analysis produces a set of *initial relevant locations.* After we have found the initial relevant locations, we propagate relevant locations to the other points of the trace by following data flow paths of the initial relevant locations in the last action. This process is described in Section 5.2.2.

5.2.1 Initial Relevant Locations

The process of finding initial relevant locations depends on the type of the action in the last step of the analysis trace. If the action is either ASSUME or ASSERT, the

```
function EVAL_CORR(C_c, C_a, F, o_c, o_a, e_c, e_a)
    if type(e_c) = reference then
        if eval(C_c, F, o_c, e_c) ~ eval(C_a, F, o_a, e_a) then
            return true
        else
            return false
    else
        v_c ← eval(C_c, F, o_c, e_c)
        v_a ← eval(C_a, F, o_a, e_a)
        if coerce(v_c, type(e_c), type(e_a)) = v_a then
            return true
        else
            return false
```

Figure 5.5: Function comparing the value of the concrete expression coerced to the abstract type to the value of the corresponding abstract value.

corresponding condition expressions in the concrete and in the abstract models are evaluated differently. Thus, this expression seems like a logical place for the search of initial relevant locations.

5.2.1.1 Finding Initial Relevant Locations from Expressions

The simplest way for finding initial relevant locations from the expressions is to add all the locations appearing in the expressions to the set of initial relevant locations. This would ensure that all the relevant locations are included but on the other hand the set might contain a large number of non-relevant locations. We chose to analyze the expressions heuristically to narrow down the number of the relevant locations.

The idea in the search for the initial relevant locations from the expressions is to traverse the concrete expression tree and the abstract expression tree, and to focus on the subexpressions that do not evaluate correspondingly. This can be easily done because the corresponding expressions have the same tree structure.

Example 5.3. For example, in the analysis trace introduced in Example 5.1 the initial relevant locations consist of only one location, $\langle 4, id(o_c), name(var_c^5)\rangle$. This can also be seen in tables 5.1 and 5.2 in lines where $i = 4$. The initial relevant locations have been searched from the expression `(`$\langle var_c^5\rangle$ `<` $\langle var_c^6\rangle$`)` and its abstract counterpart `(`$\langle var_a^5\rangle$ `<` $\langle var_a^6\rangle$`)`. The first subexpressions $\langle var_c^5\rangle$ and $\langle var_a^5\rangle$ have non-corresponding values -11 and `POS`. The second subexpressions $\langle var_c^6\rangle$ and $\langle var_a^6\rangle$ have corresponding values 6 and `POS`. Thus the initial relevant locations are searched from the first subexpressions, from which only one location, $\langle id(o_c), name(var_c^5)\rangle$, is found. ■

In the analysis of the expressions, comparison of the correspondence of the values of the expressions is often needed. Function EVAL_CORR described in Figure 5.5 is used for checking whether the concrete and the abstract expression evaluate correspondingly. If the expressions evaluate as objects, the correspondence of the resulting objects is compared and the result of the comparison is returned. Otherwise it evaluates a given concrete expression e_c in the concrete model, then coerces the value got to the type of the corresponding abstract expression, and finally compares the coerced value to the value the corresponding abstract expression e_a evaluates to in the abstract model. The result of the comparison is returned.

Function RELEVANT described in Figure 5.6 is used for finding initial relevant locations from expressions. It takes a concrete configuration C_c, an abstract configuration C_a, a function F for solving non-deterministic choices in the evaluation of abstract expressions, a concrete object o_c, a corresponding abstract object o_a, a concrete expression e_c, and a corresponding abstract expression e_a as arguments. The expressions e_c and e_a are assumed to evaluate non-correspondingly between the concrete and the abstract models (specified by the configurations, the objects, and the function F).

If the expressions are of the kind LIT or NAME, we collect locations appearing in these expressions using function COLLECT_LOCATIONS which is described in more detail later.

If the expressions are of the kind COND, we analyze which one of the two subexpressions causes the non-corresponding evaluation of the expression. The expressions of the kind COND describe logical *and* and *or* expressions for which Jumbala uses short-circuit evaluation. Both cases are analyzed in the same way. If the first subexpressions evaluate non-correspondingly we search for the initial relevant locations from those subexpressions, otherwise from the second subexpressions. The reason for the identical treatment is the assumption that the expressions evaluate non-correspondingly between the concrete and the abstract model. In that case if the first subexpressions evaluate correspondingly, then the second subexpressions do not. On the other hand if the first subexpressions evaluate non-correspondingly we do not evaluate the second subexpression at all either in the concrete or in the abstract model because of the short-circuit evaluation used, making the second subexpressions irrelevant for the analysis.

If the expressions are of the kind INFIX, we check both subexpression pairs whether they evaluate correspondingly. If the expressions in only one of the pairs evaluate non-correspondingly, then we search initial relevant locations from the subexpressions in that pair. If both pairs evaluate non-correspondingly, then we can not make a rational choice over the other. Thus, we search for initial relevant locations from both of the pairs and return the union of the relevant locations found. If both of the pairs evaluate correspondingly, then the operation of the abstract expression is non-deterministic and in the counterexample the abstract value corresponding to the concrete one was not chosen in the evaluation of the abstract expression. In this case we do not know which of the locations appearing in the expressions are best choices as initial relevant locations. Thus, all the locations appearing in the expressions are returned as initial relevant locations.

If the expressions are of the kind UNARY, we check whether the only subexpressions evaluate correspondingly. If they do not we recursively call the function RELEVANT. On the other hand, if they do the operation in the abstract expression is non-deterministic and like with the INFIX expressions, we just collect all the locations

```
function RELEVANT(C_c, C_a, F, o_c, o_a, e_c, e_a)
    if kind(e_c) ∈ {LIT, NAME} then
        return COLLECT_LOCATIONS(C_c, C_a, o_c, o_a, e_c, e_a)
    if kind(e_c) = COND then
        # subexpr(e_c) = ⟨e_c^1, e_c^2⟩ and subexpr(e_a) = ⟨e_a^1, e_a^2⟩
        if EVAL_CORR(C_c, C_a, F, o_c, o_a, e_c^1, e_a^1) = false then
            return RELEVANT(C_c, C_a, F, o_c, o_a, e_c^1, e_a^1)
        else
            return RELEVANT(C_c, C_a, F, o_c, o_a, e_c^2, e_a^2)
    if kind(e_c) = INFIX then
        # subexpr(e_c) = ⟨e_c^1, e_c^2⟩ and subexpr(e_a) = ⟨e_a^1, e_a^2⟩
        c_1 ← EVAL_CORR(C_c, C_a, F, o_c, o_a, e_c^1, e_a^1)
        c_2 ← EVAL_CORR(C_c, C_a, F, o_c, o_a, e_c^2, e_a^2)
        if c_1 = false ∨ c_2 = false then
            result ← {}
            if c_1 = false then
                result ← result ∪ RELEVANT(C_c, C_a, F, o_c, o_a, e_c^1, e_a^1)
            if c_2 = false then
                result ← result ∪ RELEVANT(C_c, C_a, F, o_c, o_a, e_c^2, e_a^2)
            return result
        else
            return COLLECT_LOCATIONS(C_c, C_a, o_c, o_a, e_c, e_a)
    if kind(e_c) = UNARY then
        # subexpr(e_c) = ⟨e_c^1⟩ and subexpr(e_a) = ⟨e_a^1⟩
        if EVAL_CORR(C_c, C_a, F, o_c, o_a, e_c^1, e_a^1) then
            return COLLECT_LOCATIONS(C_c, C_a, o_c, o_a, e_c, e_a)
        else
            return RELEVANT(C_c, C_a, F, o_c, o_a, e_c^1, e_a^1)
    if kind(e_c) = TCOND then
        # subexpr(e_c) = ⟨e_c^1, e_c^2, e_c^3⟩ and subexpr(e_a) = ⟨e_a^1, e_a^2, e_a^3⟩
        c_1 ← EVAL_CORR(C_c, C_a, F, o_c, o_a, e_c^1, e_a^1)
        if c_1 = false then
            return RELEVANT(C_c, C_a, F, o_c, o_a, e_c^1, e_a^1)
        else
            if eval(C_c, F, o_c, e_c^1) = true then
                return RELEVANT(C_c, C_a, F, o_c, o_a, e_c^2, e_a^2)
            else
                return RELEVANT(C_c, C_a, F, o_c, o_a, e_c^3, e_a^3)
```

Figure 5.6: Algorithm for finding initial relevant locations from expressions.

function COLLECT_LOCATIONS($C_c, C_a, o_c, o_a, e_c, e_a$)
  if $kind(e_c) = \text{NAME}$ **then**
    # e_c is of the form $\langle \text{NAME}, id, d_c, \langle var_c^1, \ldots, var_c^n \rangle \rangle$
    # e_a is of the form $\langle \text{NAME}, id, d_a, \langle var_a^1, \ldots, var_a^n \rangle \rangle$
    return $\{$SAFE_RESOLVE$(C_c, C_a, o_c, o_a, e_c, e_a)\}$
  else if $kind(e_c) = \text{LIT}$ **then**
    return $\emptyset$
  else
    # $subexpr(e_c)$ is of the form $\langle e_c^1, \ldots, e_c^n \rangle$
    # $subexpr(e_a)$ is of the form $\langle e_a^1, \ldots, e_a^n \rangle$
    $result \leftarrow \bigcup_{i=1}^{n}$ COLLECT_LOCATIONS$(C_c, C_a, o_c, o_a, e_c^i, e_a^i)$
    return $result$

Figure 5.7: Algorithm for finding all the locations appearing in the expression.

in the expression because we do not have a better judgement of the locations that should be included in the initial relevant locations.

If the expressions are of the kind TCOND, we first check whether the condition subexpressions evaluate correspondingly. If they do not, there is no point for concentrating on the subexpressions evaluating the value because in one model the second subexpression is evaluated but in the other the third subexpression is evaluated. Thus in this case we search initial relevant locations from the condition subexpressions. If the condition subexpressions evaluate correspondingly, we search the subexpressions that actually determine the value of the expression. The choice of the subexpression pair depends naturally on the value of the condition subexpression.

Locations are picked up from expressions using a function COLLECT_LOCATIONS described in Figure 5.7. If the corresponding expressions e_c and e_a are of the kind NAME, we find the locations the expressions represent by using SAFE_RESOLVE function which is described in Figure 5.8 and explained in the next paragraph. If the expressions are of the kind LIT, we return an empty set because no locations appear in the expressions. When the expressions are compound expressions, we call COLLECT_LOCATIONS for all the subexpression pairs and return the union of locations appearing in the subexpressions.

The function SAFE_RESOLVE is used for resolving locations from NAME kind expressions. The algorithm for the function is shown in Figure 5.8. The function takes a concrete and an abstract configuration, corresponding concrete and abstract objects, and corresponding expressions. A tuple containing an object identifier and a variable name indicating a location is returned. The idea is to iterate through the variables appearing in the expressions while fetching the objects corresponding to the variables from the configurations at each step in the iteration. At each step we also check whether the objects between the concrete and the abstract model are corresponding ones. If they are not, we stop the iteration and return a tuple pointing to a location from where the non-corresponding objects were found because there is no point in continuing the iteration when the values in the concrete and in the abstract model come from non-corresponding objects. The logical relevant locations in such a case

function SAFE_RESOLVE$(C_c, C_a, o_c, o_a, e_c, e_a)$
    # e_c is of the form $\langle \textsc{name}, id_e, d_c, \langle var_c^1, \ldots, var_c^n \rangle\rangle$
    # e_a is of the form $\langle \textsc{name}, id_e, d_a, \langle var_a^1, \ldots, var_a^n \rangle\rangle$
    for i=1 **to** n-1 **do**
        $o'_c \leftarrow VarValue(C_c, o_c, var_c^i)$
        $o'_a \leftarrow VarValue(C_a, o_a, var_a^i)$
        if $id(o'_c) \neq id(o'_a)$ **then**
            return $\langle id(o_c), name(var_c^i)\rangle$
        else
            $o_c \leftarrow o'_c$
            $o_a \leftarrow o'_a$
    return $\langle id(o_c), name(var_c^n)\rangle$

Figure 5.8: Algorithm for returning a tuple representing a location the NAME kind expression represents. If the concrete and the abstract expression does not represent corresponding locations a tuple representing the location before the first non-corresponding location is returned.

are the locations containing the first non-corresponding objects.

Example 5.4. Let o_c^1, o_c^2, and o_c^3 be objects instantiated from a class c_c in the concrete model. The corresponding abstract objects, instantiated from a class c_a, are o_a^1, o_a^2, and o_a^3, respectively. Let $var_c^1 \in Vars(c_c)$, $var_a^1 \in Vars(c_a)$, $var_c^1 \sim var_a^1$, and $type(var_c^1) = type(var_c^1) = \texttt{reference}$. Let $var_c^2 \in Vars(c_c)$, $var_a^2 \in Vars(c_a)$, $var_c^2 \sim var_a^2$, $type(var_c^2) = type(var_c^2) = \texttt{int}$. In the global configuration C_c and in the corresponding abstract global configuration C_a some of the values of the variables are $VarValue(C_c, o_c^1, var_c^1) = o_c^2$, $VarValue(C_a, o_a^1, var_a^1) = o_a^3$, $VarValue(C_c, o_c^2, var_c^2) = 1$, $VarValue(C_a, o_a^2, var_a^2) = 1$, $VarValue(C_c, o_c^3, var_c^2) = -1$, and $VarValue(C_a, o_a^3, var_a^2) = -1$. The evaluation of corresponding expressions $\langle \textsc{name}, id, \texttt{int}, \langle var_c^1, var_c^2\rangle\rangle$ and $\langle \textsc{name}, id, \texttt{int}, \langle var_a^1, var_a^2\rangle\rangle$ in the global configurations C_c and C_a gives us non-corresponding results, 1 in the concrete model and -1 in the abstract model. Still the location $\langle o_c^2, var_c^2\rangle$ has a value corresponding to the value of the location $\langle o_a^2, var_a^2\rangle$ as well as the location $\langle o_c^3, var_c^2\rangle$ has a value corresponding to the value of the location $\langle o_a^3, var_a^2\rangle$. If we are searching for relevant locations from these expressions in the global configurations described, there is no point in choosing locations represented by a tuple $\langle id(o_c^2), name(var_c^2)\rangle$, $\langle id(o_c^3), name(var_c^2)\rangle$ or both as relevant locations but instead the locations, represented by a $\langle id(o_c^1), name(var_c^1)\rangle$, which causes the non-corresponding results in the evaluation of the expressions. ■

5.2.1.2 Target Correspondence Check

If the last action is either IMPL, DEFER, or RECV, the reason for the abstract counterexample to be spurious is a send action that sends messages to non-corresponding objects in the concrete and in the abstract model as was described in Section 4.3.[2] A

[2] Implicit message consumptions and message defers caused by a guard which is false in the abstract model and true in the concrete model have been converted to RECV and ASSUME actions in

function CHECK_TARGET_CORRESPONDENCE($\widetilde{T}, o_c, o_a, id$)
  # $\widetilde{T}$ is of the form $\langle step_1, \ldots, step_n \rangle$
  for $i = 1$ **to** id **do**
    # $step_i$ is of the form $\langle\langle C_c, C_a\rangle, \langle \mathrm{i}, o'_c, o'_a, \widetilde{b}\rangle, \widetilde{C'}\rangle$
    if $\widetilde{b}$ is of the form $\langle \text{SEND}, F, sig_c, sig_a, params_c, params_a, tgt_c, tgt_a\rangle$ **then**
      $o_{tgt_c} \leftarrow eval(C_c, F, o'_c, tgt_c)$
      $o_{tgt_a} \leftarrow eval(C_a, F, o'_a, tgt_a)$
      if $(o_c = o_{tgt_c} \wedge o_a \neq o_{tgt_a}) \vee (o_c \neq o_{tgt_c} \wedge o_a = o_{tgt_a})$ **then**
        return i
  return -1

Figure 5.9: Algorithm for checking correspondence of target objects in signal send actions.

logical refinement to the abstraction would be one that makes the send action send messages to corresponding objects instead of non-corresponding objects. Thus, we look for the initial relevant locations in the target expressions of the send action.

First we have to find the action. For this we use a simple algorithm that examines all send actions in the trace to find the first send action which sends a message to non-corresponding objects and the target object is the object in the last action in the trace either in the concrete or in the abstract model but not in both models. This point of the trace is chosen because this is the point where the input queues of the desired target objects begin to diverge between the concrete and the abstract model. After the action is found, the target expressions of the action are analyzed using the function RELEVANT, just like we did with the ASSUME and ASSERT cases.

Function CHECK_TARGET_CORRESPONDENCE described in Figure 5.9 is used for comparison of the targets of the actions in the trace. It takes an analysis trace $\widetilde{T}$, target objects o_c and o_a from the concrete and the abstract model, and a sequence number id indicating the last step in the analysis trace that needs to be checked. Function CHECK_TARGET_CORRESPONDENCE returns the sequence number of the first step where a message is sent to the target object either in the concrete or in the abstract model but not in the other model. If such a step can not be found -1 is returned.

The last step to be checked is provided as an optimization when the last place that can affect the first message in the input queues in the current location is already known. For example if in the last step of the analysis trace the concrete object has a message $\langle 4, sig_c, p_c^1, \ldots, p_c^n\rangle$ in the head of the input queue and the corresponding abstract object has a message $\langle 7, sig_a, p_a^1, \ldots, p_{1_m}\rangle$ in the head of the input queue, then we know that there is a send action sending messages to non-corresponding objects at the latest in the fourth action. When searching for the initial relevant locations the optimization is actually irrelevant (because there is always a send action with non-corresponding targets) but in the propagation of the relevant locations (introduced in Section 5.2.2) the function is called in situations when there is not necessarily send

the construction of the analysis trace.

1: **function** INITIAL_RELEVANT_LOCATIONS($\widetilde{T}$)
2: # $\widetilde{T}$ is of the form $\langle step_1, \ldots, step_l \rangle$
3: # $last(\widetilde{T})$ is of the form $\langle \langle C_c, C_a \rangle, \langle id, o_c, o_a, \widetilde{b} \rangle, \perp \rangle$
4: $R \leftarrow \emptyset$
5: **if** $\widetilde{b}$ is of the form $\langle \text{ASSUME}, F, e_c, e_a \rangle$ or $\langle \text{ASSERT}, F, e_c, e_a \rangle$ **then**
6: **for all** $\langle oid, name \rangle \in \text{RELEVANT}(C_c, C_a, F, o_c, o_a, e_c, e_a)$ **do**
7: $R \leftarrow R \cup \{\langle l, oid, name \rangle\}$
8: **else**
9: # $\widetilde{b}$ is of the form $\langle \text{IMPL} \rangle$, $\langle \text{DEFER} \rangle$, or
10: # $\langle \text{RECV}, sig_c, sig_a, \langle p_c^1, \ldots, p_c^n \rangle, \langle p_a^1, \ldots, p_a^n \rangle \rangle$
11: $i \leftarrow$ CHECK_TARGET_CORRESPONDENCE($\widetilde{T}, o_c, o_a, id$)
12: # $step_i = \langle \langle C_c^i, C_a^i \rangle, \langle i, o_c^i, o_a^i, \widetilde{b_i} \rangle, \widetilde{C_{i+1}} \rangle$
13: # $\widetilde{b_i} = \langle \text{SEND}, F_i, sig_c^i, sig_a^i, \langle e_c^1, \ldots, e_c^m \rangle, \langle e_a^1, \ldots, e_a^m \rangle, tgt_c^i, tgt_a^i \rangle$
14: **for all** $\langle oid, name \rangle \in \text{RELEVANT}(C_c^i, C_a^i, F_i, o_c^i, o_a^i, tgt_c^i, tgt_a^i)$ **do**
15: $R \leftarrow R \cup \{\langle i, oid, name \rangle\}$
16: **return** R

Figure 5.10: Algorithm for calculating initial relevant locations. The algorithm takes an analysis trace $\widetilde{T}$ as argument and returns a set of relevant locations R containing the initial relevant locations.

action with non-corresponding targets.

5.2.1.3 Algorithm for Finding Initial Relevant Locations

The algorithm in Figure 5.10 is used for finding initial relevant locations for an analysis trace. It takes an analysis trace as an argument and returns a set containing the initial relevant locations. Actions ASSUME and ASSERT are handled in lines 5–7. Condition expressions (the concrete and the corresponding abstract) in these actions are searched for initial relevant locations by a call to the function RELEVANT which searches for initial relevant locations heuristically from the expressions given.

Example 5.5. In the analysis trace introduced in Example 5.1, the initial relevant locations are searched from the concrete expression ($\langle var_c^5 \rangle$ < $\langle var_c^6 \rangle$) and the corresponding abstract expression ($\langle var_a^5 \rangle$ < $\langle var_a^6 \rangle$). The set of initial relevant locations found from the expressions is $\{\langle 4, id(o_c), name(var_c^5) \rangle\}$. ■

Actions IMPL, DEFER, and RECV are handled in lines 8–15. The first send action that sends a message to a target object either in the concrete model or in the abstract model but not in the other model is found by a call to the function CHECK_TARGET_CORRESPONDENCE. After the sequence number i of such an action is found, the target expressions in the send action in the ith step is searched for initial relevant locations using function the RELEVANT.

5.2.2 Propagation of Relevant Locations

After a set of initial relevant locations has been found, we propagate relevant locations to other points in the analysis trace. The idea is to traverse through the analysis trace backwards and to follow where the values of the relevant locations come from. After this data flow analysis is done, we have found relevant locations for every step in the trace.

Example 5.6. After the initial relevant locations $\{\langle 4, id(o_c), name(var_c^5)\rangle\}$ in Example 5.1 are found, the relevant locations are propagated to other points in the trace. Relevant locations concerning C_c^3 and C_a^3 are propagated from the relevant locations concerning C_c^4 and C_a^4. Because the assignment action in the third step in the trace does not modify values in relevant locations concerning C_c^4 and C_a^4, the same locations are relevant before the execution of the assignment action. Thus the set of relevant locations is augmented with $\langle 3, id(o_c), name(var_c^5)\rangle$.

The second action on the other hand assigns a value to var_c^5 in o_c in the concrete model and to var_a^5 in o_a in the abstract model. These locations were relevant but because the action modifies the values in these locations the value before the action is not relevant and thus we do not augment the set of relevant locations with tuple $\langle 2, id(o_c), name(var_c^5)\rangle$. Instead we find what locations are used in the calculation of the values assigned to these locations. In the concrete model values in locations $\langle o_c, var_c^3\rangle$ and $\langle o_c, var_c^4\rangle$ are used and in the abstract model values in locations $\langle o_a, var_a^3\rangle$ and $\langle o_a, var_a^4\rangle$ are used. The set of relevant variables is augmented with $\langle 2, id(o_c), name(var_c^3)\rangle$ and $\langle 2, id(o_c), name(var_c^4)\rangle$.

Similarly the first action assigns values to relevant locations in C_c^2 and C_a^2 and thus the concrete and abstract expressions for calculating the values assigned are searched for new relevant locations. The set of relevant locations is augmented with entries $\langle 1, id(o_c), name(var_c^1)\rangle$, $\langle 1, id(o_c), name(var_c^2)\rangle$, and $\langle 1, id(o_c), name(var_c^3)\rangle$. It should be noted that $\langle 1, id(o_c), name(var_c^3)\rangle$ would be in the set of relevant locations even if the locations $\langle o_c, var_c^3\rangle$ or $\langle o_a, var_a^3\rangle$ are not be used in the expressions evaluating the values to be assigned in the action. ■

In the case where a message is sent to non-corresponding objects between the concrete and the abstract model, the initial relevant locations are not related to the global configurations before the last action in the trace. From the propagation's point of view this is not important, the propagation works in the same way as in the situation where the initial relevant locations are related to the global configurations before the execution of the last action in the trace.

The propagation of the relevant locations differs from traditional data flow analysis as in the traditional data flow analysis all the variables used in the calculation of the value of an important variable are collected but in the propagation of the relevant locations all the locations are not necessarily collected but instead only the locations that contribute to the search for the abstraction refinement.

5.2.2.1 Algorithm for the Propagation of Relevant Locations

The algorithm presented in Figure 5.11 is used for the propagation of the relevant locations. It takes an analysis trace $\widetilde{T}$ and a set of initial relevant locations R as argu-

function PROPAGATE_RELEVANT_LOCATIONS$(\widetilde{T}, R)$
  # $\widetilde{T}$ is of the form $\langle step_1, \ldots, step_n \rangle$
  for $i = n - 1$ **to** 1 **do**
    # $step_i$ is of the form $\langle \langle C_c, C_a \rangle, \langle i, o_c, o_a, \widetilde{b} \rangle, \widetilde{C}' \rangle$
    for all $\langle id', x, y \rangle \in R : id' = i + 1$ **do** $R \leftarrow R \cup \{\langle i, x, y \rangle\}$
    if $\widetilde{b}$ is of the form $\langle \text{ASSIGN}, F, lhs_c, lhs_a, rhs_c, rhs_a \rangle$ **then**
      $\langle o'_c, var'_c \rangle \leftarrow resolve(C_c, o_c, lhs_c)$
      $\langle o'_a, var'_a \rangle \leftarrow resolve(C_a, o_a, lhs_a)$
      if $\langle i+1, id(o'_c), name(var'_c) \rangle \in R \vee \langle i+1, id(o'_a), name(var'_a) \rangle \in R$ **then**
        if $o'_c \sim o'_a$ **then**
          $R \leftarrow R \setminus \{\langle i, id(o'_c), name(var'_c) \rangle\}$
          for all $\langle oid, name \rangle \in \text{REF}(C_c, C_a, F, o_c, o_a, rhs_c, rhs_a)$ **do**
            $R \leftarrow R \cup \{\langle i, oid, name \rangle\}$
        else
          $\langle oid, name \rangle \leftarrow \text{SAFE\_RESOLVE}(C_c, C_a, o_c, o_a, lhs_c, lhs_a)$
          $R \leftarrow R \setminus \langle i, id(o'_c), name(var'_c) \rangle$
          $R \leftarrow R \setminus \langle i, id(o'_a), name(var'_a) \rangle$
          $R \leftarrow R \cup \{\langle i, oid, name \rangle\}$
    if $\widetilde{b}$ is of the form $\langle \text{RECV}, sig_c, sig_a, \langle p_c^1, \ldots, p_c^m \rangle, \langle p_a^1, \ldots, p_a^m \rangle \rangle$ **then**
      if $\exists 1 \leq j \leq m : \langle i + 1, id(o_c), name(p_c^j) \rangle \in R$ **then**
        # $InputQueue(C_c, o_c) = \langle \langle id_{msg_c}, sig_c, v_c^1, \ldots, v_c^m \rangle, \ldots \rangle$
        $id \leftarrow \text{CHECK\_TARGET\_CORRESPONDENCE}(\widetilde{T}, o_c, o_a, i)$
        if $id \neq -1$ **then**
          $R \leftarrow \emptyset$
          # $step_{id}$ is of the form $\langle \langle C'_c, C'_a \rangle, \langle id, o'_c, o'_a, \widetilde{b}' \rangle, \widetilde{C}'' \rangle$
          # $\widetilde{b}'$ is of the form $\langle \text{SEND}, F', sig'_c, sig'_a, E'_c, E'_a, tgt'_c, tgt'_a \rangle$
          for all $\langle oid, name \rangle \in \text{RELEVANT}(C'_c, C'_a, F', o'_c, o'_a, tgt'_c, tgt'_a)$ **do**
            $R \leftarrow R \cup \{\langle id, oid, name \rangle\}$
          return PROPAGATE_RELEVANT_LOCATIONS$(\widetilde{T}, R)$
        for $j = 1$ **to** m **do**
          if $\langle i + 1, id(o_c), name(p_c^j) \rangle \in R$ **then**
            $R \leftarrow R \setminus \{\langle i, id(o_c), name(p_c^j) \rangle\}$
            $R \leftarrow R \cup \{\langle i, id_{msg_c}, j \rangle\}$
    if $\widetilde{b}$ is of the form $\langle \text{SEND}, F, sig_c, sig_a, E_c, E_a, tgt_c, tgt_a \rangle$ **then**
      # E_c is of the form $\langle e_c^1, \ldots, e_c^m \rangle$ and E_a is of the form $\langle e_a^1, \ldots, e_a^m \rangle$
      for $j = 1$ **to** m **do**
        # messages sent from this point have the message id i
        if $\langle i + 1, i, j \rangle \in R$ **then**
          $R \leftarrow R \setminus \{\langle i, i, j \rangle\}$
          for all $\langle oid, name \rangle \in \text{REF}(C_c, C_a, F, o_c, o_a, e_c^j, e_a^j)$ **do**
            $R \leftarrow R \cup \{\langle i, oid, name \rangle\}$
  return R

Figure 5.11: Algorithm for propagating relevant locations.

ments. The algorithm traverses through the analysis trace backwards and propagates relevant locations from the step visited earlier by the algorithm.

First we copy all relevant locations from the succeeding step. A set of locations obtained this way is then modified depending on the type of the action in the current step.

If the action is of the type ASSIGN, we modify the set of relevant locations only if the assignment modifies a value in one of the relevant locations. If a value in a relevant location is modified the first thing is to check that the location the assignment is modifying is the same one in both models. If it is, the location modified is not relevant earlier in the trace because the location does not get its value until the current step. On the other hand, locations used for the calculation of the values assigned (found by a call to a function REF described later) are relevant and thus added to the set of relevant locations (the location to which the value is assigned might be used in the calculation of the value in which case that location is relevant also earlier in the trace). This is done in lines 10–13 in the algorithm.

If the locations modified differ between the concrete and the abstract model, then the values assigned are irrelevant. This is because we have no reason to expect that the locations where the value is assigned only in one model but not in the other should have corresponding values. For the same reason there is no point in being interested about the values of the locations before the current step. Thus the locations are removed from the set of relevant locations. Instead we turn our attention to the expressions defining the locations modified. The reason why the assignment is done to non-corresponding locations is found from these expressions. We use SAFE_RESOLVE to get a location where the determination of the target location of the assignment takes differing paths between the concrete and the abstract models. We add this location to the set of relevant locations because the non-corresponding values in the concrete and in the abstract model affect the values affecting the occurrence of the spurious counterexample.

It could be argued that the locations modified only in either of the concrete and abstract models should not be relevant after the assignment. This would lead to the question of relevance of the locations whose values were calculated using values in locations modified only in one of the models. Because later (Section 5.3) we are interested only in the first relevant location having non-corresponding values, we shall not modify relevant locations later in the trace.

If the action is of the type RECV, we modify the set of relevant locations only if one of the assignments done in the process of receiving a message modifies the value of a location belonging to the set of relevant locations. Because the locations where values are assigned in RECV actions are always local, i.e. the assignment can be made without dereferences, the assignments are always done to corresponding locations. If there are such assignments, we first make sure that the messages received are the corresponding ones. This is done by checking that all message sending actions up to this point send a message either to both objects, the concrete and the abstract, of this action or neither of them. For this check the function CHECK_TARGET_CORRESPONDENCE is used. If it finds a place where a message is sent to only one of the objects of this action, then we start over the search for relevant locations by calling PROPAGATE_RELEVANT_LOCATIONS with a set of initial relevant locations found from the target expression of the send action pointed out by function CHECK_TARGET_CORRESPONDENCE. The idea behind restarting is to help the search for a refinement focus on the locations which cause the target expressions to evaluate non-correspondingly and thus try to force the send action to send messages to corresponding objects after refinement. Message queue comparison and the possible restarting of the search for relevant locations are done in lines 23–29 in the algorithm. Making sure that the message queue histories of the message receiving objects correspond in both models might very well be too cautious. This topic is discussed further in Section 7.1.

When we have ensured that the input queue histories correspond (up to the position at which the message is received) in the objects of the action, we remove the relevant locations modified in the message receiving action from the set of relevant locations and for every relevant location assigned we add the corresponding message parameter to the set of relevant locations. This is done in lines 30–33 in the algorithm.

If the action is of the type SEND, we check whether there are relevant message parameters in the message the action sends. For each relevant message parameter we remove that message parameter from the set of relevant locations (after all, before the execution of the send action there is no such message) and add all the locations used in the calculation of the parameter to the set of relevant locations. This is done in lines 34–41 in the algorithm.

The algorithm returns a set of relevant locations in the trace.

5.2.2.2 Finding Locations Used in Expression Evaluation

The function REF described in Figure 5.12 is used for finding locations used in the evaluation of a pair of corresponding expressions. It takes a concrete and an abstract configuration, a function for solving non-deterministic choices in the abstract evaluation, corresponding objects for the concrete and the abstract model, and corresponding expressions as arguments. The algorithm returns a set of locations containing one location per a pair of corresponding NAME kind of subexpressions appearing in the pair of corresponding expressions given to the algorithm. The location for a pair of NAME kind of expressions is got using the function SAFE_RESOLVE.

If the expressions are of the type LIT or NAME, function COLLECT_LOCATIONS is used to determine the set of locations returned. If the expressions are of the type COND, we call REF recursively for the subexpressions that are evaluated, either in the concrete or in the abstract model. Short-circuit evaluation used in the evaluation of COND expressions in Jumbala is taken into account when deciding which subexpressions are searched for the locations. If the expressions are of the type INFIX, both subexpression pairs are searched for locations, recursively. If the expressions are of the type UNARY, the only subexpression pair is searched for locations. If the expressions are of the type TCOND, we search for the locations from the condition expressions and depending on the values of the condition expressions from the expressions used for determining the values.

The algorithm used for the search of locations used in the evaluation of a pair of corresponding expressions is relatively simple for the reason that we did not want to make things too complex in the development of the algorithm for propagating relevant locations. One point for optimization might be the handling of ternary conditional expressions. In the case of condition expressions evaluating non-correspondingly between the concrete and the abstract model it could be reasonable to add just the locations used for the evaluation of the condition expressions to the set of relevant locations.

function $\text{REF}(C_c, C_a, F, o_c, o_a, e_c, e_a)$
  if $\mathit{kind}(e_c) \in \{\text{LIT}, \text{NAME}\}$ **then**
    return $\text{COLLECT\_LOCATIONS}(C_c, C_a, o_c, o_a, e_c, e_a)$
  if $\mathit{kind}(e_c) = \text{COND}$ **then**
    # $\mathit{subexpr}(e_c) = \langle e_c^1, e_c^2 \rangle$ and $\mathit{subexpr}(e_a) = \langle e_a^1, e_a^2 \rangle$
    if $\mathit{operator}(e_c) = \texttt{\&\&}$ **then**
      if $\mathit{eval}(C_c, F, o_c, e_c^1) = \mathit{false} \wedge \mathit{eval}(C_a, F, o_a, e_a^1) = \mathit{false}$ **then**
        return $\text{REF}(C_c, C_a, F, o_c, o_a, e_c^1, e_a^1)$
      else
        return $\text{REF}(C_c, C_a, F, o_c, o_a, e_c^1, e_a^1) \cup \text{REF}(C_c, C_a, F, o_c, o_a, e_c^2, e_a^2)$
    else if $\mathit{operator}(e_c) = \texttt{||}$ **then**
      if $\mathit{eval}(C_c, F, o_c, e_c^1) = \mathit{true} \wedge \mathit{eval}(C_a, F, o_a, e_a^1) = \mathit{true}$ **then**
        return $\text{REF}(C_c, C_a, F, o_c, o_a, e_c^1, e_a^1)$
      else
        return $\text{REF}(C_c, C_a, F, o_c, o_a, e_c^1, e_a^1) \cup \text{REF}(C_c, C_a, F, o_c, o_a, e_c^2, e_a^2)$
  if $\mathit{kind}(e_c) = \text{INFIX}$ **then**
    # $\mathit{subexpr}(e_c) = \langle e_c^1, e_c^2 \rangle$ and $\mathit{subexpr}(e_a) = \langle e_a^1, e_a^2 \rangle$
    return $\text{REF}(C_c, C_a, F, o_c, o_a, e_c^1, e_a^1) \cup \text{REF}(C_c, C_a, F, o_c, o_a, e_c^2, e_a^2)$
  if $\mathit{kind}(e_c) = \text{UNARY}$ **then**
    # $\mathit{subexpr}(e_c) = \langle e_c^1 \rangle$ and $\mathit{subexpr}(e_a) = \langle e_a^1 \rangle$
    return $\text{REF}(C_c, C_a, F, o_c, o_a, e_c^1, e_a^1)$
  if $\mathit{kind}(e_c) = \text{TCOND}$ **then**
    # $\mathit{subexpr}(e_c) = \langle e_c^1, e_c^2, e_c^3 \rangle$ and $\mathit{subexpr}(e_a) = \langle e_a^1, e_a^2, e_a^3 \rangle$
    $\mathit{result} \leftarrow \text{REF}(C_c, C_a, F, o_c, o_a, e_c^1, e_a^1)$
    if $\mathit{eval}(C_c, F, o_c, e_c^1) = \mathit{true} \vee \mathit{eval}(C_a, F, o_a, e_a^1) = \mathit{true}$ **then**
      $\mathit{result} \leftarrow \mathit{result} \cup \text{REF}(C_c, C_a, F, o_c, o_a, e_c^2, e_a^2)$
    if $\mathit{eval}(C_c, F, o_c, e_c^1) = \mathit{false} \vee \mathit{eval}(C_a, F, o_a, e_a^1) = \mathit{false}$ **then**
      $\mathit{result} \leftarrow \mathit{result} \cup \text{REF}(C_c, C_a, F, o_c, o_a, e_c^3, e_a^3)$
    return result

Figure 5.12: Algorithm for finding locations used in the evaluation of a pair of corresponding expressions.

5.3 Refining Interval Abstractions

In interval abstractions every abstract type used in the abstract model divides the domain of `int` into integer intervals. For example 32-bit signed integers could be divided to intervals $[-2^{31}, -5], [-4, 1], [2, 2], [3, 2^{31} - 1]$. The evaluation of an interval abstracted expression can result in any interval containing an integer resulting from an evaluation of the corresponding non-abstracted expression with some combination of integers in the abstracted operands' intervals as the operands of the non-abstracted expression. For example $[2, 2] + [-4, 1] = x$, where $x \in \{[-4, 1], [2, 2], [3, 2^{31} - 1]\}$. Interval abstractions can be intuitively refined by splitting the intervals. For example, $[-2^{31}, -5], [-4, 1], [2, 2], [3, 2^{31} - 1]$ can be refined by splitting the interval $[3, 2^{31} - 1]$ into three new intervals $[3, 5]$, $[6, 12]$, and $[13, 2^{31} - 1]$ producing a new set of intervals $[-2^{31}, -5], [-4, 1], [2, 2], [3, 5], [6, 12], [13, 2^{31} - 1]$. We describe a process for finding a suitable refinement automatically when using interval abstractions. This makes possible a fully automatic model checking procedure with interval abstractions.

After an abstract counterexample has been identified as a spurious one, an analysis trace representing the counterexample is formed. The idea is to calculate relevant locations in the counterexample using the algorithms described in Section 5.2. After the relevant locations have been determined, we find the first point in the trace where the value of a relevant location in the concrete model does not correspond to the value in the abstract model. Because the definition of an abstract model ensures that initially values in all locations have corresponding values between the concrete and the abstract model, we have an action in the trace that has produced the non-correspondence between the values in the concrete and abstract model. Hence after we have found the first point with non-corresponding values in the relevant locations, we search for the action that caused the locations to have non-corresponding values. This action has to be the action executed just before the point where the relevant location with non-corresponding values has been found. Otherwise the values in this location would have been non-corresponding earlier in the trace.

The expressions determining the values assigned to the relevant location in the action are analyzed much in the same fashion as are the expressions analyzed when searching for the set of initial relevant locations. The idea is to search for a set of locations and a set of values from the pair of corresponding expressions to guide the refinement process. Variables in the classes corresponding to the set of locations found are refined in such a way that for every variable having an abstract type, a new type is created. The domain of the new type is the domain of the old type with all the integer values found in the expression analysis separated to distinct intervals. For example if a concrete variable var_c with `int` domain was abstracted to a corresponding variable var_a with domain containing intervals $[-2^{31}, -5], [-4, 1], [2, 2], [3, 2^{31}-1]$ and our analysis has found that the type of var_a needs to be refined with value -13, then the domain of the new type for var_a is $[-2^{31}, -14], [-13, -13], [-12, -5][-4, 1], [2, 2], [3, 2^{31}-1]$. With this refinement we try to make the expression equivalent to the expression causing the first non-corresponding values in the relevant locations to evaluate correspondingly in the abstract model generated with the refined abstraction.

function FIND_REFINEMENT($M_c, M_a, \widetilde{T}, R$)
  # M_c is of the form $\langle C_{init\,c}, D_c, Classes_c, O_c, Locations_c, SysSigs_c, ExtSigs\rangle$
  # M_a is of the form $\langle C_{init\,a}, D_a, Classes_a, O_a, Locations_a, SysSigs_a, ExtSigs\rangle$
  # $\widetilde{T}$ is of the form $\langle step_1, \ldots, step_n\rangle$
  for $i = 1$ **to** $n - 1$ **do**
    # $step_i$ is of the form $\langle\langle C_c, C_a\rangle, \langle i, o_c, o_a, \widetilde{b}\rangle, \langle C'_c, C'_a\rangle\rangle$
    if $\widetilde{b}$ is of the form $\langle$ASSIGN$, F, lhs_c, lhs_a, rhs_c, rhs_a\rangle$ **then**
      $\langle o'_c, var'_c\rangle \leftarrow resolve(C'_c, o_c, lhs_c)$
      if $\langle i+1, id(o'_c), name(var'_c)\rangle \in R$ **then**
        if EVAL_CORR($C'_c, C'_a, F, o_c, o_a, lhs_c, lhs_a$) $= false$ **then**
          return REF_EXPR($C_c, C_a, F, o_c, o_a, rhs_c, rhs_a$)
    if $\widetilde{b}$ is of the form $\langle$SEND$, F, sig_c, sig_a, E_c, E_a, tgt_c, tgt_a\rangle$ **then**
      # E_c is of the form $\langle e_c^1, \ldots, e_c^m\rangle$
      # E_a is of the form $\langle e_a^1, \ldots, e_a^m\rangle$
      if $\exists\langle i, x, y\rangle \in R \wedge \nexists\langle i+1, z, w\rangle \in R$ **then**
        return REF_EXPR($C_c, C_a, F, o_c, o_a, tgt_c, tgt_a$)
      for $j = 1$ **to** m **do**
        if $\langle i+1, i, j\rangle \in R$ **then**
          if EVAL_CORR($C_c, C_a, F, o_c, o_a, e_c^j, e_a^j$) $= false$ **then**
            return REF_EXPR($C_c, C_a, F, o_c, o_a, e_c^j, e_a^j$)
  # $step_n$ is of the form $\langle\langle C_c, C_a\rangle, \langle i, o_c, o_a, \widetilde{b}\rangle, \widetilde{C}'\rangle$
  # $\widetilde{b}$ is of the form $\langle$ASSUME$, F, e_c, e_a\rangle$ or $\langle$ASSERT$, F, e_c, e_a\rangle$
  return REF_EXPR($C_c, C_a, F, o_c, o_a, e_c, e_a$)

Figure 5.13: Algorithm for finding expressions where the refinements are searched for.

5.3.1 Finding Expressions for Analysis

The algorithm shown in Figure 5.13 determines a pair of corresponding expressions from where a suitable refinement is searched for. After the expression pair has been found, the refinement calculated by the function REF_EXPR (described in Section 5.3.2) is returned.

The algorithm takes a concrete model, an abstract model corresponding to the concrete model, an analysis trace, and a set of relevant locations in the trace as arguments. We iterate through the steps in the analysis trace. For each step we check whether the action in the step produced non-corresponding values to the relevant locations. Only ASSIGN and SEND actions need to be checked because RECV action assigns message parameters with a type coercible to the type of the variable the value is assigned to. Thus, if the value of an abstract message parameter corresponds to the value of the corresponding concrete message parameter, then the value of the variable where the value of the message parameter is assigned to corresponds to the value of the corresponding concrete variable after the assignment.

If the action is of the type ASSIGN, and the location where the assignment assigns a value is relevant after the assignment, but the value in the location in the abstract model does not correspond to the value in the concrete model after the assignment, we search for a refinement from the right-hand side expressions. In the algorithm this is done in lines 7–11.

If the action is of the type SEND, and there are relevant locations before the send action but not after, we have searched for the initial relevant locations from the target expressions of the send action. In this situation we search for the refinement from the target expressions in the action. This case is handled in lines 15–16 in the algorithm.

If the target expressions of the action were not searched for initial relevant locations, we check whether each relevant abstract message parameter corresponds to the concrete one. The pair of expressions evaluating the first pair of message parameters with non-corresponding values is searched for the refinement. In the algorithm this is done in lines 17–20.

If an expression to be searched for refinement has not been found before the last step in the trace, we have a situation where the type of the last step in the trace is either ASSUME or ASSERT and all the relevant locations in the trace have had corresponding values. In this case we search the condition expressions in the action for refinement. In the algorithm this is done in line 23.

5.3.2 Analysis of Expressions for Refinement

When the expressions to be searched for abstraction refinement are found, the function REF_EXPR described in Figure 5.14 is used for the search for refinement. The function traverses the expression trees and focuses to subexpressions evaluating non-correspondingly. When no pair of corresponding subexpressions can be chosen over the other or all the subexpression pairs evaluate correspondingly, we collect all the locations with an abstract type appearing in the expressions to a set of locations to be refined[3] and evaluate values of the subexpressions of the current concrete expression and collect all integer values from those to form a set of values guiding the refinement.

Example 5.7. In the example abstract model introduced in Example 3.1 (analysis trace matching the counterexample was described in Example 5.1), the abstract type $\texttt{Sign} = \{\texttt{NEG}, \texttt{ZERO}, \texttt{POS}\}$ is an interval abstraction. The value `NEG` represents the interval $[-2^{31}, -1]$, the value `ZERO` represents the interval $[0, 0]$, and the value `POS` represents the interval $[1, 2^{31}-1]$. In the analysis trace of the model (introduced in Section 5.2) the first point with non-corresponding relevant locations is after the execution of the first action. Because the first action is an assignment, the refinement is searched from the expressions defining the assigned values. In this case the expressions are $\texttt{((}\langle var_c^1\rangle \texttt{ + } \langle var_c^2\rangle\texttt{) - }\langle var_c^3\rangle\texttt{)}$ and $\texttt{((}\langle var_a^1\rangle \texttt{ + } \langle var_a^2\rangle\texttt{) - }\langle var_a^3\rangle\texttt{)}$. The first subexpressions $e_c = \texttt{(}\langle var_c^1\rangle \texttt{ + } \langle var_c^2\rangle\texttt{)}$ and $e_a = \texttt{(}\langle var_a^1\rangle \texttt{ + } \langle var_a^2\rangle\texttt{)}$ evaluate as -1 and `POS`, respectively, in the trace. The second subexpressions $\langle var_c^3\rangle$ and $\langle var_a^3\rangle$ evaluate as 5 and `POS`, respectively. So, the first subexpressions evaluate non-correspondingly and the second subexpressions evaluate correspondingly. Thus the first subexpressions are searched further for the refinement.

[3]As stated earlier, the actual type of the locations is not refined but the type of the variable in a class defining the type of the location.

```
function REF_EXPR(C_c, C_a, F, o_c, o_a, e_c, e_a)
    if kind(e_c) ∈ {LIT, NAME} then
        return ⟨COLLECT_INT_LOCS(C_c, F, o_c, e_c), VALUES(C_c, F, o_c, e_c)⟩
    if kind(e_c) = COND then
        # subexpr(e_c) = ⟨e_c^1, e_c^2⟩ and subexpr(e_a) = ⟨e_a^1, e_a^2⟩
        if EVAL_CORR(C_c, C_a, F, o_c, o_a, e_c^1, e_a^1) = false then
            return REF_EXPR(C_c, C_a, F, o_c, o_a, e_c^1, e_a^1)
        else
            return REF_EXPR(C_c, C_a, F, o_c, o_a, e_c^2, e_a^2)
    if kind(e_c) = INFIX then
        # subexpr(e_c) = ⟨e_c^1, e_c^2⟩ and subexpr(e_a) = ⟨e_a^1, e_a^2⟩
        c_1 ← EVAL_CORR(C_c, C_a, F, o_c, o_a, e_c^1, e_a^1)
        c_2 ← EVAL_CORR(C_c, C_a, F, o_c, o_a, e_c^2, e_a^2)
        if c_1 = c_2 then
            return ⟨COLLECT_INT_LOCS(C_c, F, o_c, e_c), VALUES(C_c, F, o_c, e_c)⟩
        else
            if c_1 = false then
                return REF_EXPR(C_c, C_a, F, o_c, o_a, e_c^1, e_a^1)
            if c_2 = false then
                return REF_EXPR(C_c, C_a, F, o_c, o_a, e_c^2, e_a^2)
    if kind(e_c) = UNARY then
        # subexpr(e_c) = ⟨e_c^1⟩ and subexpr(e_a) = ⟨e_a^1⟩
        if EVAL_CORR(C_c, C_a, F, o_c, o_a, e_c^1, e_a^1) then
            return ⟨COLLECT_INT_LOCS(C_c, F, o_c, e_c), VALUES(C_c, F, o_c, e_c)⟩
        else
            return REF_EXPR(C_c, C_a, F, o_c, o_a, e_c^1, e_a^1)
    if kind(e_c) = TCOND then
        # subexpr(e_c) = ⟨e_c^1, e_c^2, e_c^3⟩ and subexpr(e_a) = ⟨e_a^1, e_a^2, e_a^3⟩
        c_1 ← EVAL_CORR(C_c, C_a, F, o_c, o_a, e_c^1, e_a^1)
        if c_1 = false then
            return REF_EXPR(C_c, C_a, F, o_c, o_a, e_c^1, e_a^1)
        else
            if eval(C_c, F, o_c, e_c^1) = true then
                return REF_EXPR(C_c, C_a, F, o_c, o_a, e_c^2, e_a^2)
            else
                return REF_EXPR(C_c, C_a, F, o_c, o_a, e_c^3, e_a^3)
```

Figure 5.14: Algorithm for finding a concrete subexpression guiding the refinement.

Because the subexpressions of the expressions e_c and e_a evaluate correspondingly, the locations and values guiding the refinement is searched from the expressions e_c and e_a. The set of locations is $\{\langle o_c, var_c^1\rangle, \langle o_c, var_c^2\rangle\}$ and the set of values is $\{-2, -1, 1\}$. The type of the variables var_a^1 and var_a^2 is `Sign`. After the refinement the type of the variables $\widehat{var}_a^1$ and $\widehat{var}_a^2$ corresponding to the concrete variables var_c^1 and var_c^2

is representing the intervals $[-2^{31}, -3], [-2, -2], [-1, -1], [0, 0], [1, 1], [2, 2^{31} - 1]$. In the abstract model, constructed with the refined abstraction, all the properties hold indicating the properties hold also in the concrete model. ■

5.3.2.1 Algorithm for Analysis of Expressions

The heuristic behind the subexpression analysis is that we assume that the non-correspondingly evaluating abstract subexpression causes the abstract expression to evaluate non-correspondingly. This assumption does not necessarily hold and therefore in the refined abstraction the same action can cause the first non-corresponding values in a relevant location. Our approach still forces the abstract expression to eventually evaluate correspondingly (if the concrete data types are finite and the engine generating abstract models does not over-approximate in places where it is not necessary) because every refinement cycle refines the abstraction resulting in the end an abstraction where every abstract value corresponds to a single integer value. At this point the concrete and the abstract model are in practice the same model and there is no extra behavior in the abstract model to produce spurious counterexamples. Our approach still aims to refine the model enough so that the spurious counterexample is removed from the abstract model but not too much to avoid state space explosion, the very problem abstractions try to solve.

The function REF_EXPR takes a concrete and an abstract configuration, a function F for solving non-determinism in the evaluation, a concrete and an abstract object, and a concrete and an abstract expression as arguments.

If the expressions are of the kind LIT or NAME, we collect locations and all integer subexpression values for refinement from the concrete expression.

If the expressions are of the kind COND, the first non-correspondingly evaluating subexpression pair is searched recursively for refinement. If the first pair of subexpressions does not evaluate correspondingly, the other pair of subexpressions (even though it might evaluate non-correspondingly as well) is not searched for refinement because it is evaluated only either in the concrete or in the abstract model but not in the other due to Jumbala's short-circuit evaluation semantics.

If the expressions are of the kind INFIX, we check whether the subexpression pairs evaluate correspondingly. If exactly one of the subexpression pairs evaluate non-correspondingly, the refinement is searched from that pair recursively. Otherwise we collect locations and all integer subexpression values for refinement from the entire concrete INFIX expression.

If the expressions are of the kind UNARY, we search for a refinement recursively from the subexpressions if the subexpressions evaluate non-correspondingly. Otherwise all locations and all integer subexpression values from the concrete UNARY expression are collected for refinement.

When the expressions are of the kind TCOND, the condition expressions are first checked for corresponding values. If they evaluate to non-corresponding values, the condition subexpressions are searched for the refinement recursively. Otherwise subexpressions evaluating the actual value of the current expressions are searched recursively for refinement.

The actual generation of a refined abstract model is not described here. It has to be in accordance with the restrictions relating to abstract models described in Section 3.2. Thus, assigning a refined type to a variable affects the types of expressions and may affect the types of other variables.

For completeness, Figure 5.15 describes the function COLLECT_INT_LOCS used for finding locations with type `int` and Figure 5.16 describes the function VALUES used for the evaluation of all `int` type subexpressions (except the subexpressions not evaluated because of the short-circuit evaluation).

```
function COLLECT_INT_LOCS(C, F, o, e)
    if kind(e) = NAME then
        # e is of the form ⟨NAME, id, d, ⟨var_1, ..., var_n⟩⟩
        if d = int then
            return {resolve(C, o, e) }
        else
            return ∅
    else if kind(e) = LIT then
        return ∅
    else if kind(e) = COND then
        # subexpr(e) = ⟨e_1, e_2⟩
        if operator(e) = && then
            if eval(C, F, o, e_1) = false then
                return COLLECT_INT_LOCS(C, F, o, e_1)
            else
                return ⋃_{i∈{1,2}} COLLECT_INT_LOCS(C, F, o, e_i)
        else if operator(e) = || then
            if eval(C, F, o, e_1) = true then
                return COLLECT_INT_LOCS(C, F, o, e_1)
            else
                return ⋃_{i∈{1,2}} COLLECT_INT_LOCS(C, F, o, e_i)
    else if kind(e) = TCOND then
        # subexpr(e) = ⟨e_1, e_2, e_3⟩
        if eval(C, F, o, e_1) = true then
            return ⋃_{i∈{1,2}} COLLECT_INT_LOCS(C, F, o, e_i)
        else
            return ⋃_{i∈{1,3}} COLLECT_INT_LOCS(C, F, o, e_i)
    else
        # subexpr(e) is of the form ⟨e_1, ..., e_n⟩
        return ⋃_{i∈{1,...,n}} COLLECT_INT_LOCS(C, F, o, e_i)
```

Figure 5.15: Algorithm for finding all the locations in the expression with a type `int`.

```
function VALUES(C, F, o, e)
    if kind(e) ∈ {LIT, NAME} then
        if type(e) = int then
            return {eval(C, F, o, e)}
        else
            return ∅
    if kind(e) = COND then
        # subexpr(e) = ⟨e1, e2⟩
        if operator(e) = && then
            if eval(C, F, o, e1) = false then
                return VALUES(C, F, o, e1)
            else
                return VALUES(C, F, o, e1) ∪ VALUES(C, F, o, e2)
        else if operator(e) = || then
            if eval(C, F, o, e1) = true then
                return VALUES(C, F, o, e1)
            else
                return VALUES(C, F, o, e1) ∪ VALUES(C, F, o, e2)
    if kind(e) = INFIX then
        # subexpr(e) = ⟨e1, e2⟩
        if type(e) = int then
            return {eval(C, F, o, e)} ∪ VALUES(C, F, o, e1) ∪ VALUES(C, F, o, e2)
        else
            return VALUES(C, F, o, e1) ∪ VALUES(C, F, o, e2)
    if kind(e) = UNARY then
        # subexpr(e) = ⟨e1⟩
        if type(e) = int then
            return {eval(C, F, o, e)} ∪ VALUES(C, F, o, e1)
        else
            return VALUES(C, F, o, e1)
    if kind(e) = TCOND then
        # subexpr(e) = ⟨e1, e2, e3⟩
        if eval(C, F, o, e1) = true then
            return VALUES(C, F, o, e1) ∪ VALUES(C, F, o, e2)
        else
            return VALUES(C, F, o, e1) ∪ VALUES(C, F, o, e3)
```

Figure 5.16: Algorithm for evaluating all `int` type subexpressions of the expression given. A set of `int` subexpression values is returned.

Chapter 6

Implementation

The techniques described in this thesis have been implemented as part of the Symbolic Methods for UML Behavioural Diagrams (SMUML) project (see `http://www.tcs.hut.fi/Research/Logic/SMUML.shtml`) in the Laboratory for Theoretical Computer Science at Helsinki University of Technology. The purpose of the SMUML project was to develop new techniques for analysis of dynamic behavior of models described in UML. In the SMUML project the version 1.4 of UML was used. The reason for the use of this particular version is that the SMUML toolkit is built upon the meta-modelling toolkit Coral [3], developed at Åbo Akademi, and UML 1.4 was the latest UML version supported by Coral.

6.1 SMUML Toolkit

The tools developed in the SMUML project form the SMUML toolkit. The UML models supported in the SMUML toolkit describe a set of objects (instantiated from classes) communicating asynchronously via message passing and shared variables. The action language Jumbala [18] is used for representation of guards and effects of transitions.

With the symbolic model checker `NuSMV` [9] or a Satisfiability Modulo Theories (SMT, see e.g. [17, 31, 7]) solver, the tools in the SMUML toolkit offer a bounded model checking [6] procedure with automatically refined data abstractions. The absence of assertion failures or implicit message consumptions are the properties supported with the model checking procedure with abstractions.

The abstract model generation [34] is done by rewriting expressions in the model so that abstract types and abstract operations are encoded as integers and operations between integers. Other tools have also used similar approach [26, 24, 28]. The abstract models produced in this way can be model checked and simulated by the tools used in the model checking and simulation of the concrete models except that the tools have to additionally handle the non-determinism introduced by the abstraction.

The generated abstract model can be encoded [20, 21, 22] to a format understood by `NuSMV`, then model checked with `NuSMV`, and finally the result got from `NuSMV` is interpreted. The result can either be that the properties hold in the model or that we get an abstract counterexample demonstrating a property violation in the abstract model. Alternatively the abstract model can be model checked with a bounded model checker [19] using an external SMT solver (for example [23, 5, 16]) as a back-end.

The counterexample analysis (including feasibility analysis) are implemented into a counterexample analyzer `canal` [32]. The analyzer takes the concrete UML model, the corresponding abstract UML model, and the counterexample trace as an input and either states that the counterexample is feasible, or in the case of spurious counterexample, returns a set of variables and values as a recommendation for the refinement. If the counterexample is a spurious one, a new abstract model is generated based on the recommendation and the model checking is carried out again with the refined model.

6.2 Counterexample Analyzer in SMUML Toolkit

The techniques described in this thesis have been implemented as a part of the counterexample analyzer `canal`. The UML models supported by `canal` consist of objects instantiated from classes whose behavior is represented with UML state machines. Objects may communicate with each other by message passing or by shared variables. The analyzer can handle only non-hierarchical UML models. This limitation was imposed to keep the implementation simple in order to make the development of new algorithms easy. The SMUML toolset contains tools for converting hierarchical UML models to non-hierarchical ones. The semantics used for the interpretation of UML models are described in [21].

The analyzer supports the following UML and Jumbala features: We support UML state machines that have only one UML composite state, otherwise composite states are forbidden. UML simple states, final states, and from UML pseudo states, initial states and choice states are supported. Every other state type is forbidden. States cannot have any internal behavior like entry, exit, or doActivities.

Only UML signal events are supported as triggers of transitions in UML state machines. The UML models have to have primitive components describing “int” and “bool” data types as these are Jumbala’s primitive data types. Reference types are modelled using UML associations. Arrays are not supported.

6.3 Conversion from UML 1.4

The conversion from UML 1.4 models supported by `canal` to a model notion used in this thesis is fairly straightforward. The actual model checking still have to be done with the UML model because the transition relation of the model notion used in this thesis allows more transitions to be taken than the corresponding UML model. Even though all the execution paths in our model definition cannot be taken in the corresponding UML model, all the executions possible in the UML model are possible in the model converted from the UML model to our notion for a model. After the model checking is done, the concrete and the abstract UML model as well as the counterexample can be converted to the model notion of this thesis and the algorithms introduced in this thesis can be used to the converted models and the converted counterexample.

Informally, the mapping from a UML model to a model notion of this thesis is the following: For every UML class and object there is a corresponding class and object, respectively. Type "int" and "bool" class attributes have corresponding `int` and `boolean` type variables in the class corresponding to the UML class. UML associations are converted to variables of type `reference` in the class. The UML state machine of a class is converted to a corresponding state machine so that all states in the UML state machine have a corresponding state in the model, and the initial state is the initial state of the UML state machine. UML signal events deferrable in the states are converted to a *defers* function of the state machine. The *flush* set of the state machine contains all states whose UML counterparts are simple states. This set models the states where deferred messages are moved back to the input queue. For every UML transition there is a corresponding transition with a trigger corresponding to the UML trigger of the UML transition receiving messages of a corresponding signal and assigning values to corresponding variables. The guard of the transition is the same expression converted to use the variables of the model instead of the ones in the UML model. Jumbala statements in the effect of the UML transition are converted to corresponding actions in the new transition. UML signals are converted to corresponding signals. Finally the initial configuration matching the initial configuration of the UML model is formed.

Chapter 7

Conclusion

In this thesis we have discussed the analysis of abstract counterexamples in the model checking process with data abstractions. The notation and semantics of models used in describing the methods introduced in this thesis were described in Chapter 2.

In Chapter 3 we described how abstractions were utilized in model checking in the SMUML project. We also defined the relationship between the data abstracted model and the original model. A check whether an abstract counterexample has a corresponding execution in the concrete model, feasibility analysis using a stepwise simulation of events in a counterexample, was described in Chapter 4. A similar approach is used in Bandera [33].

We have introduced a method to calculate relevant locations at different points of a spurious counterexample trace (Section 5.2). The calculation starts by searching initial relevant locations from one point of the trace. After the initial relevant locations are found, relevant locations are propagated to other points of the trace using data flow analysis shaped for this purpose. The propagation might be restarted with new initial relevant variables if the input queues of objects are not corresponding.

Before this thesis automatic abstraction refinement was done only with BDD-based [10] and predicate abstractions [8]. Data abstraction papers only mentioned that there is a need for refinement but did not describe how the refinement should be done [33]. In Section 5.3 we introduced a method for determining a suitable refinement automatically. It first searches the action whose evaluation in the abstract model causes one of the relevant locations to contain non-corresponding values in the concrete and the abstract model. The expression evaluating this value in the action is analyzed with a heuristic algorithm which determines which of the variables appearing in the expression need to be refined and how they should be refined.

Actual implementation of these techniques was done as part of the SMUML project in the Laboratory for Theoretical Computer Science at Helsinki University of Technology. This implementation, which works on UML state machine models, was described in Chapter 6.

7.1 Future Work

In the future the techniques described in this thesis should be tested with real life case studies to see how the techniques perform in practice. At this time the SMUML toolkit does not offer a good platform for performance tests, even though the techniques have been implemented to `canal`, because our implementation of abstractions in the SMUML toolkit proved to be quite inefficient. In the SMUML toolkit the abstractions were implemented by inlining the abstract types and operations using Jumbala's integer data type. This can lead, in the worst case, to an exponential increase in the size of the expressions. The huge size of the expressions causes problems to the Jumbala parser utilized in the counterexample analysis and possibly also to the model checker. Since the Jumbala parser, being implemented in Python, is not very efficient, the problem with the size of the expressions becomes emphasized.

We have also developed several ideas for improving the algorithms. First, when an assignment is done to a relevant location in the message reception in concrete object o_c and in the corresponding abstract object o_a, the current algorithm checks that all send actions send a message either to the concrete object o_c and the abstract object o_a or neither of them. This ensures that the queues are corresponding sequences of messages in all points of the trace up to the point where the message received in the message reception is sent. Furthermore this ensures that the messages received in the concrete and in the abstract model are sent in the same point of the trace. It would also be possible to use a weaker condition that allows the message queues of the concrete and the abstract object to be non-corresponding so long as the concrete message and the abstract message received (containing relevant locations) are the corresponding ones. With this modification it is possible to get the spurious counterexample out of the abstract model with less refined abstraction because the modifications may allow some messages to be sent either to the concrete target object o_c but not to the corresponding abstract object or vice versa even if there is a send action sending messages with relevant message parameters to the object o_c and the corresponding abstract object.

In the algorithm for propagating relevant locations we could more carefully analyze the expressions assigning values to relevant locations. Instead of simply marking all the locations containing values used in the evaluation of the expression as relevant, we could analyze the expressions and mark only locations causing the expressions to evaluate non-correspondingly as relevant locations. The analysis would be similar to the expression analysis made in the search for initial relevant locations. It is possible that with this improvement the spurious counterexample could be removed with a smaller refinement but the details of the implementation and the effects of the implementation are not yet thoroughly studied.

When a pair of corresponding expressions is searched for locations and values to guide the refinement, we refine all the types of the locations with the same set of values. Instead we could refine the type of each location in the expression tree with values appearing in the path leading to the subexpression in which the location appears. This is in line with the constraints relating to the types of subexpressions in the model. This technique may produce more types with smaller domains compared to the current algorithm. It is unclear and most likely also depends on the actual implementation whether the modification actually improves the performance of the model checking procedure. The main reason for the use of the simple method in this thesis and in `canal` is that this way the construction of the abstract model is also more simple.

The refinement does not necessarily remove the spurious counterexample from the abstract model (Section 5.3.2). To ensure that the refined abstract model does not contain the spurious counterexample, we could simulate all the executions matching the counterexample's transition sequence in the refined abstract model and if we this way obtain a spurious counterexample (all the counterexamples found are bound to be spurious because the concrete model did not contain a counterexample matching the transition sequence), we would refine the abstraction again until we cannot find spurious counterexamples matching the transition sequence.

Abstract loop counters are a general problem with abstractions and also our data abstractions do not perform well with abstracted loop counters. This is because the loop have to be unrolled exactly same number of times in the concrete and in the abstract execution. However the extra behavior introduced by the abstraction can allow the abstract model to "jump over" some of the unrollings. Though this is not due the counterexample analysis, a more sophisticated way of handling loops would benefit the whole model checking procedure.

Bibliography

[1] OMG unified modelling language specification, version 1.4. Object Management Group, 2001.

[2] UML 2.0 superstructure specification. Object Management Group, 2005.

[3] Marcus Alanen and Ivan Porres. Coral: A metamodel kernel for transformation engines. In *Proc. Second European Workshop on Model Driven Architecture (MDA)*, number 17-04 in Tech. Report, pages 165–170. Computing Laboratory, Univ. of Kent, 2004.

[4] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. *ACM SIGPLAN Notices*, 36(5):203–213, 2001.

[5] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer-Verlag, July 2007. Berlin, Germany.

[6] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *TACAS '99: Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 193–207, London, UK, 1999. Springer-Verlag.

[7] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi Junttila, Peter van Rossum, Stephan Schulz, and Roberto Sebastiani. MathSAT: Tight integration of SAT and mathematical decision procedures. *Journal of Automated Reasoning*, 35(1–3):265–293, October 2005.

[8] S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient verification of sequential and concurrent C programs. *Formal Methods in System Design*, 25(2-3):129–166, 2004.

[9] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)*, volume 2404 of *LNCS*, Copenhagen, Denmark, July 2002. Springer.

[10] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[11] E. Clarke and R. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, 1996.

[12] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. The MIT Press, 1999.

[13] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

[14] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448. Association for Computing Machinery, 2000.

[15] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction and approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL 1977)*, pages 238–252. Association for Computing Machinery, 1977.

[16] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963/2008 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin / Heidelberg, 2008.

[17] Leonardo de Moura, Bruno Dutertre, and Natarajan Shankar. A tutorial on satisfiability modulo theories. In *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 20–36. Springer Berlin / Heidelberg, 2007.

[18] Jori Dubrovin. Jumbala — an action language for UML state machines. Master's thesis, Helsinki University of Technology, Department of Engineering Physics and Mathematics, 2006.

[19] Jori Dubrovin. SMUML/Suboco 1.10 — an SMT-based UML bounded model checker, 2007. Computer program in the SMUML Software Release 1.0.0, available at `http://www.tcs.hut.fi/Research/Logic/SMUML.shtml`.

[20] Jori Dubrovin. SMUML/Uboco 1.10 — a translator from UML models to NuSMV programs, 2007. Computer program in the SMUML Software Release 1.0.0, available at `http://www.tcs.hut.fi/Research/Logic/SMUML.shtml`.

[21] Jori Dubrovin and Tommi Junttila. Symbolic model checking of hierarchical UML state machines. In Jonathan Billington, Zhenhua Duan, and Maciej Koutny, editors, *Proceedings of the 8th International Conference on Application of Concurrency to System Design (ACSD'08)*, pages 108–117, Xi'an, China, June 2008. IEEE Press.

[22] Jori Dubrovin, Tommi Junttila, and Keijo Heljanko. Symbolic step encodings for object based communicating state machines. In Gilles Barthe and Frank S. de Boer, editors, *Proceedings of the 10th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'08)*, volume 5051 of *Lecture Notes in Computer Science*, pages 96–112, Oslo, Norway, June 2008. Springer-Verlag.

[23] Bruno Dutertre and Leonardo de Moura. System description: Yices 1.0, August 2006. SMT-COMP 2006 system description paper. `http://www.csl.sri.com/users/demoura/smt-comp/descriptions/yices-smtcomp06.pdf`.

[24] M. Gallardo, J. Martínez, P. Merino, and E. Pimentel. αSPIN: A tool for abstract model checking. *International Journal on Software Tools for Technology Transfer (STTT)*, 5(2–3):165–184, 2004.

[25] S. Graf and H. Saïdi. Constructing abstract state graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV 1997)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer-Verlag, 1997.

[26] J. Hatcliff, M. B. Dwyer, C. S. Păsăreanu, and Robby. Foundations of the Bandera abstraction tools. In *The Essence of Computation, Complexity, Analysis, Transformation: Essays Dedicated to Neil D. Jones*, volume 2566 of *Lecture Notes in Computer Science*, pages 172–203. Springer-Verlag, 2002.

[27] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BLAST. In *Proceedings of the 10th International SPIN Workshop on Model Checking Software (SPIN 2003)*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer-Verlag, 2003.

[28] G. J. Holzmann and M. H. Smith. An automated verification method for distributed systems software based on model extraction. *IEEE Transactions on Software Engineering*, 28(4), 2002.

[29] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[30] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[31] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *Journal of the ACM*, 53(6):937–977, 2006.

[32] Vesa Ojala. SMUML/canal 1.0.0 — a counterexample analyzer for analyzing abstract counterexamples from data abstracted UML models, 2007. Computer program.

[33] C. S. Păsăreanu, M. B. Dwyer, and W. Visser. Finding feasible abstract counterexamples. *International Journal on Software Tools for Technology Transfer (STTT)*, 5(1):34–48, 2003.

[34] Heikki Tauriainen. SMUML/abstractor 1.0.0 — data abstraction and abstract type generation tools for abstracting UML state machines, 2007. Computer program in the SMUML Software Release 1.0.0, available at `http://www.tcs.hut.fi/Research/Logic/SMUML.shtml`.

[35] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[36] A. Valmari. The state explosion problem. *Lectures on Petri Nets I: Basic Models*, 1491:193–207, 1998.

[37] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[38] W. Visser, S. Park, and J. Penix. Using predicate abstraction to reduce object-oriented programs for model checking. In *Proceedings of the 3rd Workshop on Formal Methods in Software Practice (FMSP 2000)*, pages 3–12. Association for Computing Machinery, 2000.

[39] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[40] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.