From Trees to DAGs: A General Lattice Model for Symbolic Execution

Von Bäumen zu gerichteten azyklischen Graphen: Ein allgemeines Verbandsmodell für symbolische Ausführung
Master-Thesis von Dominic Scheurer aus Darmstadt
Tag der Einreichung:

1. Gutachten: Prof. Dr. Reiner Hähnle
2. Gutachten: Nathan Wasser, Dr. Richard Bubel

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Software Engineering (SE)

From Trees to DAGs:
A General Lattice Model for Symbolic Execution
Von Bäumen zu gerichteten azyklischen Graphen: Ein allgemeines Verbandsmodell für symbolische Ausführung

Vorgelegte Master-Thesis von Dominic Scheurer aus Darmstadt

1. Gutachten: Prof. Dr. Reiner Hähnle
2. Gutachten: Nathan Wasser, Dr. Richard Bubel

Tag der Einreichung:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 9. April 2015

(D. Scheurer)

Abstract

Symbolic Execution is a precise static program analysis technique for software testing and verification. In the course of the analysis, programs are transformed into symbolic execution trees containing up to exponentially many branches in the number of branch points. We address this so-called “path explosion problem” in the context of program verification by proposing a general lattice-based framework for join operations that allows for the merging of branches during symbolic execution. Several concrete join techniques are presented as instances of this framework and are implemented for the deductive verification system KeY. We show that our operations indeed reduce the number of states and branches significantly for certain examples, and apply a join technique to information flow analysis in a short case study to demonstrate that state joining can increase the precision of analyses in principle.

Zusammenfassung (German Abstract)

Symbolische Ausführung ist eine präzise Technik zur statischen Analyse von Programmen im Bereich des Testens und der Verifikation von Software. Im Zuge der Analyse werden Programme in symbolische Ausführungsbäume transformiert, welche bis zu exponentiell viele Zweige (in Abhängigkeit von der Zahl der Verzweigungspunkte) enthalten. Wir gehen dieses sogenannte “Pfadexplosionsproblem” im Kontext der Programmverifikation an, indem wir ein allgemeines verbandsbasiertes Rahmenwerk für Verbindungsoperationen angeben, welches die Zusammenführung von Zuständen während der symbolischen Ausführung erlaubt. Verschiedene konkrete Verbindungstechniken werden als Instanzen dieses Rahmenwerks vorgestellt und implementiert für das deduktive Verifikationssystem KeY. Wir zeigen, dass unsere Operationen für gewisse Beispiele tatsächlich die Zahl der Zustände und Abzweigungen signifikant reduzieren, und wenden eine Verbindungstechnik auf den Bereich der Informationsflussanalyse an, um zu demonstrieren, dass die Zusammenführung von Zuständen prinzipiell dazu in der Lage ist, die Präzision von Analyseverfahren in diesem Bereich zu erhöhen.

Contents

1 Introduction

1.1 Motivation

Symbolic Execution [Bur74; Kin76] is a method to systematically explore all execution paths in a program for all possible input values. In contrast to concrete execution, symbolic execution treats input values as symbols. Whenever the execution depends on the unknown concrete value of a program variable, it splits into subbranches that are thereupon followed independently (→ Figure 1.1). The result is a *symbolic execution tree*, consisting of *symbolic execution states*, which resembles the unrolled control flow graph. Since its inception in the 1970s, symbolic execution has been employed in two fundamentally different scenarios: (i) The *state exploration* for the purpose of, for instance, test case generation or debugging [BEL75; Kin76; God12; JMN13; CS13], and (ii) the *formal verification* of programs against functional properties [Bur74; DE82; BHS07]. The strength of symbolic execution is its precision. However, there are some drawbacks: First, "classic" symbolic execution is not capable of, for instance, fixpoint iteration for unbounded loops, and relies on repeated loop unwinding. Extensions allow for the manual specification of loop invariants to facilitate the termination of such executions. Thus, symbolic execution techniques usually lack full automation. Second, the splits of the tree at branch points where the execution depends on concrete values cause an up to exponential increase of the tree size ("*path explosion problem*") (see, for example, [CS13]). Existing approaches in literature addressing the path explosion problem in a debugging / testing context often use subsumption techniques to stop execution of redundant paths [APV06; BCE08; Jaf+12; JMN13; CJM14] or employ guided search strategies for finding good test cases faster [BS08; Xie+09]. In a verification context, these techniques are not applicable since the complete symbolic execution tree has to be considered to prove the desired properties.

Abstract Interpretation [CC77; Cou01] is a static analysis method in which concrete values are abstracted to suitable values of a chosen abstract domain. The analysis follows the control flow of a program; in particular, and in contrast to symbolic execution, branches are merged at places where the control flow converges. Thereby, abstract values are joined according to the join operation of the abstract domain. Systems based on abstract interpretation can achieve full automation [Cou+05], in particular because of their capability of performing a fixpoint iteration for (unbounded) loops. The abstraction, on the other hand, induces a natural loss of precision. A wrong choice of the abstract domain can render it impossible to prove certain kinds of properties.

An obvious idea for tackling the path explosion problem in a verification context is to take up ideas of abstract interpretation by joining branches of symbolic execution trees at suitable points. In the course of this, the underlying data structure for symbolic execution is transformed from a tree to a *Directed Acyclic Graph (DAG)*. Several approaches [HSS09; Kuz+12; Sen+14] realize this by using if-then-else constructs to merge symbolic execution states. All these approaches preserve the precision of symbolic

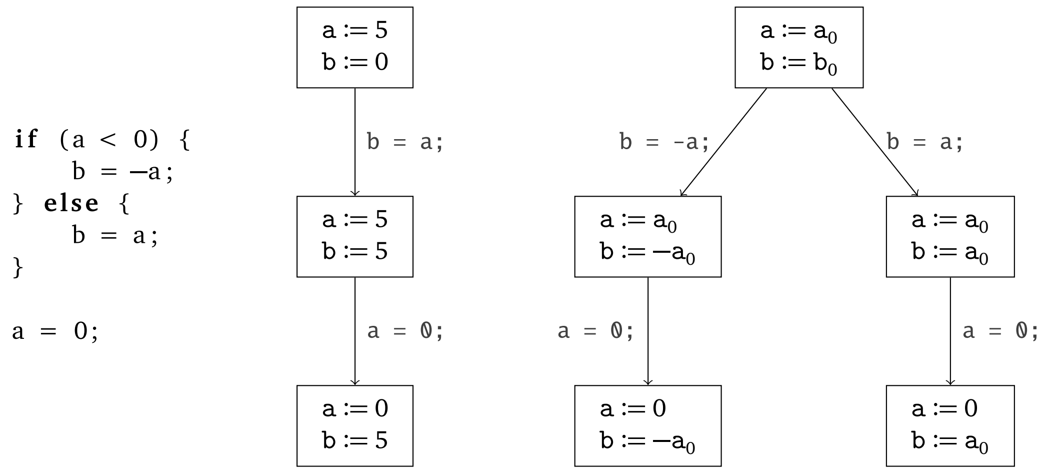


Figure 1.1: Concrete Execution vs. Symbolic Execution

execution and propose different kinds of optimizations. However, they are inflexible regarding their restriction to a fixed kind of join technique, considering that standard abstract interpretation systems allow for an arbitrary selection of abstract domains.

KeY [BHS07] is a deductive verification system for first-order Java Dynamic Logic. During the verification process, Java programs are executed symbolically by the means of special calculus rules, resulting in so-called *updates* representing the effects of the analyzed program. Afterward, properties about the program can be proven by first-order reasoning (augmented by the treatment of updates) with a sequent calculus. The proof procedure works in a semi-automatic fashion: Powerful automatic strategies in KeY significantly reduce the amount of interaction, which might though be required for instance in the case of difficult existential quantifiers or for the specification of loop invariants. The system, in its role as a symbolic execution engine, is the foundation of further applications like symbolic debuggers [Häh+10] and visualizers [HHB14].

In this thesis, we propose a novel framework for embedding join operations into symbolic execution in the context of software verification. Our goal is to contribute to solving the path explosion problem and, by making symbolic execution more flexible, to open it up to new kinds of program analyses built thereupon.

1.2 Outline

Chapter 2 contains preliminaries for the remainder of the thesis. In Chapter 3, we define the notion of *concretizations* of symbolic execution states and a partial order relation called *weakening* between symbolic execution states. In the tradition of [CC77], we base our framework upon *lattice structures* induced by join operations. Besides basic lattice properties, the join operations are required to satisfy two additional correctness properties. We define a join rule for operations conforming with the lattice framework and formally prove a corresponding soundness theorem. Furthermore, we specify concrete join operations based on, e.g., if-then-else constructs and lattice-based abstraction in the sense of abstract interpretation, and show that these operations conform with our framework. We implemented the presented operations in the KeY system; the implementation is outlined in Chapter 4. In Chapter 5, we report the results of a first evaluation of our implementation. Our experiments show that joining branches in KeY proof trees reduces the number of nodes and branches in the tree for several example programs, hence our approach indeed constitutes a step towards solving the path explosion problem. A small application to information flow analysis suggests that branch joining might be able to improve the precision of information flow analyses based on symbolic execution, or could serve as a basis for new analyses. Chapter 6 provides a comparison to related work as well as an outlook on possible future improvements and extensions of our system.

2 Preliminaries

This chapter introduces basic mathematical notions employed in the thesis, selected important concepts concerning the syntax and semantics of Java Dynamic Logic (Java DL), which is the logic that our formalisms are based on, and some fundamentals on Symbolic Execution (SE).

2.1 Basic Definitions

Definition 2.1 (Power Set and Star Operation)**.** For a set A, the power set (i.e., set of all subsets) 2^A is defined as $2^A := \{S : S \subseteq A\}$. We denote by A^n the set of tuples in $\underbrace{A \times A \times \cdots \times A}_{n \text{ times}}$, where $\times$ is the Cartesian set product, by A^* the set $\bigcup_{n\geq 0} A^n$, and by A^+ the set $\bigcup_{n\geq 1} A^n$. $\Diamond$

Definition 2.2 (Projection of Tuples)**.** For sets $A_1, \ldots, A_n$, let $\overline{a} = (a_1, \ldots, a_n) \in A_1 \times \cdots \times A_n$ be a tuple of length n. Then we denote by $proj_i(\overline{a})$ the i-th projection of $\overline{a}$, i.e. $proj_i(\overline{a}) := a_i$. For sets of tuples $A \subseteq A_1 \times \cdots \times A_n$, we define $proj_i(A) := \left\{proj_i(\overline{a}) : \overline{a} \in A\right\}$. $\Diamond$

Lattices and Semilattices

The concept of a *lattice* can be defined in two seemingly different ways: (i) as a partially ordered set ("poset") with special properties, namely the *existence* of unique least upper and greatest lower bounds, and (2) as a structure / algebra with operations $\sqcup$ (join) and $\sqcap$ (meet) for *computing* least upper and greatest lower bounds. As shown in [Grä78, Theorem 1], these definitions are actually equivalent. From a poset lattice, one can construct an equivalent algebra lattice by defining $a \sqcap b := \inf\{a, b\}$ and $a \sqcup b := \sup\{a, b\}$; from an algebra lattice, one can construct a poset lattice by defining $a \preceq b$ iff $a \sqcap b = a$ (or equivalently, $a \preceq b$ iff $a \sqcup b = b$). The restriction of an algebra lattice to only one of the operations $\sqcap$ and $\sqcup$ yields the notion of a *semilattice*, that is a join-semilattice or a meet-semilattice, depending on the included operation. We subsequently provide a definition of semilattices as structures following [Grä78].

Definition 2.3 (Semilattice)**.** A *semilattice* $(A, \circ)$ consists of a non-empty set A and one binary operation $\circ$, such that the properties (L1), (L2) and (L3) are satisfied for $a, b, c \in A$:

(L1)	*Idempotency:*	$a \circ a = a$.
(L2)	*Commutativity:*	$a \circ b = b \circ a$
(L3)	*Associativity:*	$(a \circ b) \circ c = a \circ (b \circ c)$

$\Diamond$

Control Flow Graphs

A Control Flow Graph (CFG) [All70] is a directed graph capturing the control flow relationships in a program. The CFG serves as a basis for many kinds of program analyses like abstract interpretation [CC77]. It consists of a unique entry node START and other nodes called *basic blocks* comprising a linear sequence of program instructions. The edges of the graph correspond to the control flow of the underlying program. Edges arising from conditional branchings in the control flow are labeled with the corresponding branch condition.

Example 2.4. Figure 2.1 shows the CFG for the simple Java program of Listing 2.1. The boxes are basic blocks, the last block `y = 0;` is a "program terminating block". $\Diamond$

2.2 Dynamic Logic

Java Dynamic Logic (Java DL) extends (typed) first-order logic by including Java programs as well as syntactic elements capturing state changes in the language. For programs, there exist two *modalities*: $\langle p \rangle\, \varphi$ expresses that the program p terminates and afterward the formula φ holds, whereas $[p]\, \varphi$ expresses the weaker condition that *if* p terminates, the formula φ holds afterward. State changes can be declared by so-called (syntactic) *updates* which roughly correspond to substitution functions. Subsequently, we introduce syntactic and semantic concepts of Java DL which are used in this thesis. For all notions in this section that are mentioned but not explicitly defined, we refer to [Ben11] (the main source for the fundamental definitions) and [BHS07, Chapter 3].

```java
x = 0;
while (z < y) {
    z = z + y;
    x++;
}
y = 0;
```

Listing 2.1: Example Java program

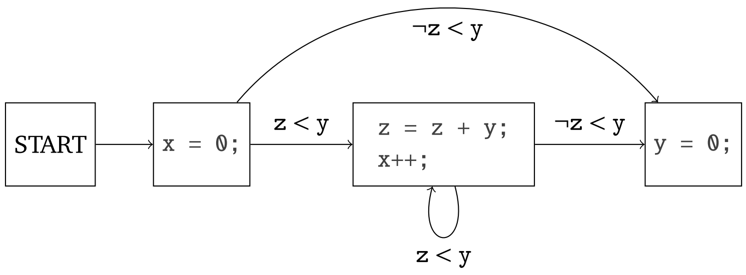


Figure 2.1: CFG for Listing 2.1.

2.2.1 Java DL Syntax

Subsequently we provide the definitions for the syntactic categories of signatures, terms, formulae, and updates of Java DL. All definitions originate from [Ben11], but may be slightly simplified and shortened. For the complete definitions, please consult [Ben11].

A Java DL signature is basically a signature of typed first-order logic with subtyping, equipped with a Java program *Prg*. As in first-order logic, the language of Java DL is parametric in a signature which defines the primitives from which terms and formulae may be built.

Definition 2.5 (Java DL Signature). A *signature* is a tuple $\Sigma = (\mathcal{T}, \preceq, \mathrm{PV}, \mathrm{LgV}, \mathrm{Func}, \mathrm{Pred}, \alpha, Prg)$ consisting of (i) a finite set of *types* $\mathcal{T}$ such that *Any*, *Boolean*, *Int*, *Null*, *LocSet*, *Field*, *Heap*, *Object* $\in \mathcal{T}$, also containing all reference types of *Prg*, (ii) a partial order $\preceq \subseteq \mathcal{T} \times \mathcal{T}$ on $\mathcal{T}$, called the *subtype relation*, as depicted in Figure 2.2, (iii) an infinite set LgV of *logical variables,* (iv) an infinite set PV of *program variables* such that all local variables `a` of type T in *Prg* also appear as $\mathtt{a} \in \mathrm{PV}$ with type A, where $A = T$ if T is a reference type, $A = Boolean$ if $T =$ `boolean`, and $A = Int$ if $T \in \{$`byte`,`short`,`int`$\}$, (v) an infinite set Func of *function symbols,* (vi) an infinite set Pred of *predicate symbols,* (vii) a static *typing function* α such that $\alpha(v) \in \mathcal{T}$ for $v \in \mathrm{PV} \cup \mathrm{LgV}$, $\alpha(f) \in \mathcal{T}^* \times \mathcal{T}$ for $f \in \mathrm{Func}$, and $\alpha(p) \in \mathcal{T}^+$ for $p \in \mathrm{Pred}$, and (viii) a program *Prg* in the intersection between Java and Java Card, i.e. a set of Java classes and interfaces.

We require that the following symbols are present in every signature for each type $A \in \mathcal{T}$: $\mathtt{heap} \in \mathrm{PV}$, $cast_A \in \mathrm{Func}$, $select_A \in \mathrm{Func}$, $store \in \mathrm{Func}$, $create \in \mathrm{Func}$ and $created \in \mathrm{Func}$, with $\alpha(cast_A) = (Any, A)$, $\alpha(select_A) = ((Heap, Object, Field), A)$, $\alpha(store) = ((Heap, Object, Field, Any), Heap)$, $\alpha(create) = ((Heap, Object), Heap)$, $\alpha(created) = Field$. ◊

The distinction between rigid and non-rigid predicate and function symbols prevailing in [BHS07] is dropped. In this framework, rigid symbols had the same interpretation in all states, whereas non-rigid symbols could change their meaning between state transitions. Since general non-rigid functions and predicates became obsolete in the more recent framework of [Ben11], program variables are now contained in the dedicated set PV as the henceforth only non-rigid function symbols.

One of the main results of [Ben11] is the integration of an explicit model of the Java heap as the value of a special program variable `heap` into Java DL. A heap structure is logically represented as a term consisting of nested *store* expressions; for instance, the update $\mathtt{heap} := store(\mathtt{heap}, o, f, 2)$ changes the initial heap by setting the field $o.f$ to 2. The result is again a heap expression that can be the input of further store expressions.

In the following, if not otherwise specified, we assume an underlying signature $\Sigma = (\mathcal{T}, \preceq, \mathrm{PV}, \mathrm{LgV}, \mathrm{Func}, \mathrm{Pred}, \alpha, Prg)$ for the program of interest *Prg* as given.

$$\begin{tikzcd}
 & & \mathit{Any} & & \mathit{Field} & \mathit{Heap} & \dots \\
\mathit{Boolean} \arrow[urr] & \mathit{Int} \arrow[ur] & \mathit{Object} \arrow[u] & \mathit{LocSet} \arrow[ul] & & & \\
 & \dots \arrow[ur] & & \dots \arrow[ul] & & & \\
 & & \mathit{Null} \arrow[ul] \arrow[ur] & & & &
\end{tikzcd}$$

Figure 2.2: Structure of Java DL type hierarchies. Source: [Ben11]

From now on, we expect all Java / Java Card programs in Java DL to be "legal" fragments. Instead of giving a complete formal definition of legal syntax and semantics for Java programs, which would go beyond the scope of this thesis, we refer to the *Java Language Specification* [Gos+05]. For the complete definition, see [Ben11, Definition 5.2].

Subsequently, we define the syntax of Java DL terms, formulae and updates by a BNF specification.

Definition 2.6 (Java DL Syntax)**.** The sets Terms_Σ^A of *terms* of type A, Form_Σ of *formulae* and Upd_Σ of *updates* are defined by the following grammar:

$$\begin{aligned}
\mathrm{Terms}_\Sigma^A ::=\ & x \;\big|\; \mathtt{a} \;\big|\; f\left(\mathrm{Terms}_\Sigma^{B_1'}, \dots, \mathrm{Terms}_\Sigma^{B_n'}\right) \;\big|\\
& \mathit{if}\,(\mathrm{Form}_\Sigma)\,\mathit{then}\left(\mathrm{Terms}_\Sigma^A\right)\mathit{else}\left(\mathrm{Terms}_\Sigma^A\right) \;\big|\; \left\{\mathrm{Upd}_\Sigma\right\}\mathrm{Terms}_\Sigma^A\\
\mathrm{Form}_\Sigma ::=\ & \mathit{true} \;\big|\; \mathit{false} \;\big|\; p\left(\mathrm{Terms}_\Sigma^{B_1'}, \dots, \mathrm{Terms}_\Sigma^{B_n'}\right) \;\big|\; \neg\mathrm{Form}_\Sigma \;\big|\; \mathrm{Form}_\Sigma \wedge \mathrm{Form}_\Sigma \;\big|\\
& \mathrm{Form}_\Sigma \vee \mathrm{Form}_\Sigma \;\big|\; \mathrm{Form}_\Sigma \rightarrow \mathrm{Form}_\Sigma \;\big|\; \mathrm{Form}_\Sigma \leftrightarrow \mathrm{Form}_\Sigma \;\big|\\
& \forall A\, x; \mathrm{Form}_\Sigma \;\big|\; \exists A\, x; \mathrm{Form}_\Sigma \;\big|\; [p]\,\mathrm{Form}_\Sigma \;\big|\; \langle p\rangle\,\mathrm{Form}_\Sigma \;\big|\; \left\{\mathrm{Upd}_\Sigma\right\}\mathrm{Form}_\Sigma\\
\mathrm{Upd}_\Sigma ::=\ & \mathit{skip} \mid \mathtt{a} := \mathrm{Terms}_\Sigma^{A'} \;\big|\; \mathrm{Upd}_\Sigma \parallel \mathrm{Upd}_\Sigma \;\big|\; \left\{\mathrm{Upd}_\Sigma\right\}\mathrm{Upd}_\Sigma
\end{aligned}$$

for any variable $x \in \mathrm{LgV}$ s.th. $\alpha(x) = A$, any program variable $\mathtt{a} \in \mathrm{PV}$ s.th. $\alpha(\mathtt{a}) = A$, any function symbol $f \in \mathrm{Func}$ s.th. $\alpha(f) = ((B_1, \dots, B_n), A)$ and predicate symbol $p \in \mathrm{Pred}$ s.th. $\alpha(p) = (B_1, \dots, B_n)$, where $B_1' \preceq B_1, \dots, B_n' \preceq B_n$, any legal program fragment p in the context of *Prg*, and any type $A' \in \mathcal{T}$ with $A' \preceq A$. The set Terms_Σ of (arbitrarily typed) terms is defined as $\mathrm{Terms}_\Sigma := \bigcup_{A \in \mathcal{T}} \mathrm{Terms}_\Sigma^A$. As usual, we call a Java DL term, formula and update *closed* if it contains no free (unbound) logic variables. ◊

The terms of Java DL are similar to terms of first-order logic, except for program variables ("non-rigid constant symbols") and the if-then-else constructs, which are additions. Informally, a term $\mathit{if}\,(\varphi)\,\mathit{then}\,(t_1)\,\mathit{else}\,(t_2)$ evaluates to t_1 if φ holds and to t_2 otherwise. A further peculiarity of Java DL is the concept of *updates* that are employed to syntactically represent the effect of terminating program executions. The KeY system (symbolically) executes the statements of given Java programs and records the effect in an update, until the end of the program is reached ($\rightarrow$ Section 2.3). Intuitively, an *elementary* update $\mathtt{a} := t$ assigns the value of the term t to the program variable $\mathtt{a}$; a *parallel* update $U_1 \parallel U_2$ executes the updates U_1 and U_2 in parallel. Curly braces transform updates into update applications. Example 2.11 demonstrates the evaluation of updates.

Subsequently, we introduce a generalized notion of substitutions that allows, besides the usual substitution of free variables, also the substitution of constant symbols. Following [Rüm03], this concept can be characterized as *nullary f-substitution*. We need this extended definition to allow for the substitution of Skolem constants by quantifiable variables: In our join methods in Chapter 3, we sometimes introduce fresh Skolem constants in the spirit of a universally quantified variable; for faithfully using those as preconditions in sequents, we need to quantify over them.

Definition 2.7 (Substitution)**.** Let $\varphi \in \mathrm{Form}_\Sigma$ be a Java DL formula. By $\varphi\,[t'/t\,]$, where $t, t' \in \mathrm{Terms}_\Sigma^A$ for any type A, and t is *either a logic variable or a constant*, we denote the formula resulting from a substitution of t by t' in φ. If $t \in \mathrm{LgV}$ is a logic variable, we only substitute unbound instances of t, i.e. those that are not in the scope of an existential or universal quantifier for that variable. ◊

For convenience, we also introduce the following tuple notation.

Notation 2.8 (Tuple Notation). We abbreviate tuples of variables or constants $(x_1, x_2, \dots, x_n)$ by $\overline{x}$ and call n the *length* of the tuple. By $\forall/\exists\overline{x}\varphi$ we understand $\forall/\exists x_1; \dots \forall/\exists x_n; \varphi$; the notation $\varphi\left[\overline{t}/\overline{t'}\right]$ denotes $\varphi\left[t_1/t'_1\right]\left[\dots/\dots\right]\left[t_n/t'_n\right]$. ◊

2.2.2 Java DL Semantics

Java DL syntax elements are interpreted by Kripke structures which, simply speaking, allow for constructing transition systems with first-order models as vertices. A given Kripke structure assigns the same meaning to all function or predicate symbols; however, it may differ in the interpretation of the program variables. Whenever the value of a program variable is changed, the Kripke structure proceeds to another *state*. Together with variable assignments, Kripke structures interpret arbitrary Java DL formulae (of suitable signatures) of our language. The following definitions again originate from [Ben11]; some of those only occur in a shortened, simplified form subsequently.

Definition 2.9 (Java DL Kripke Structure). A *Java DL Kripke Structure* $K_{\Sigma_p} = (D, \delta, I, S, \rho)$ consists of (i) a set D of semantical values, called the *domain*, (ii) a dynamic typing function $\delta : D \to \mathcal{T}$, which gives rise to the *subdomains* $D^A = \{x \in D : \delta(x) \preceq A\}$ for all types $A \in \mathcal{T}$, (iii) an *interpretation function* I mapping every function symbol $f \in \text{Func}$ with $\alpha(f) = ((A_1, \dots, A_n), A)$ to a function $I(f) : D^{A_1}, \dots, D^{A_n} \to D^A$ and every predicate symbol $p \in \text{Pred}$ with $\alpha(p) = (A_1, \dots, A_n)$ to a relation $I(p) \subseteq D^{A_1} \times \dots \times D^{A_n}$, (iv) a set S of *states*, which are functions $\sigma \in S$ mapping every program variable $\mathtt{a} \in \text{PV}$ with $\alpha(\mathtt{a}) = A$ to a value $\sigma(\mathtt{a}) \in D^A$, and (v) a function ρ that associates with every program fragment p a transition relation $\rho(p) \subseteq S^2$ s.th. $(\sigma_1, \sigma_2) \in \rho(p)$ iff p, when started in σ_1, terminates normally (i.e., not by throwing an exception) in σ_2 [Gos+05]. We consider Java programs to be deterministic, so for all program fragments p and all $\sigma_1 \in S$, there is at most one σ_2 s.th. $(\sigma_1, \sigma_2) \in \rho(p)$. We require that every Kripke structure satisfies the following:

- S is the set of *all* functions mapping program variables to properly typed values (it is therefore completely determined by D and δ)
- $D^{Boolean} = \{tt, \mathit{ff}\}$, $D^{Int} = \mathbb{Z}$, $D^{Null} = \{null\}$, $D^{Heap} = D^{Object} \times D^{Field} \to D^{Any}$
- $I(true) = tt$, $I(false) = \mathit{ff}$
- $I(cast_A)(x) = \begin{cases} x & \text{if } x \in D^A \\ null & \text{if } x \notin D^A \text{ and } A \preceq Object \\ \emptyset & \text{if } x \notin D^A \text{ and } A = LocSet \\ \mathit{ff} & \text{if } x \notin D^A \text{ and } A = Boolean \end{cases}$
- $I(select_A)(h, o, f) = I(cast_A)(h(o, f))$ for all $h \in D^{Heap}$, $o \in D^{Object}$, $f \in D^{Field}$
- $I(store)(h, o, f, x)(o', f') = \begin{cases} x & \text{if } o = o',\ f = f' \text{ and } f \neq I(created) \\ h(o', f') & \text{otherwise} \end{cases}$

  for all $h \in D^{Heap}$, $o, o' \in D^{Object}$, $f, f' \in D^{Field}$, $d \in D^{Any}$
- $I(create)(h, o)(o', f) = \begin{cases} tt & \text{if } o = o',\ o \neq null \text{ and } f = I(created) \\ h(o', f) & \text{otherwise} \end{cases}$

  for all $h \in D^{Heap}$, $o, o' \in D^{Object}$, $f \in D^{Field}$. ◊

For the same reasons why we did not formalize the syntactical correctness of Java programs, we also omit a definition of the semantics of Java programs. Instead, a "black box" function ρ is utilized to capture the behavior of legal program fragments p. The symbolic execution rules of the Java DL calculus provide a formalization of the Java semantics. Listing all those rules would go beyond the scope of this thesis.

Kripke structures allow for the definition of the semantics of Java DL terms, formulae and updates based on a valuation function *val*.

Definition 2.10 (Java DL Semantics). Given a Kripke structure $K_\Sigma = (D, \delta, I, S, \rho)$, a state $\sigma \in S$ and a *variable assignment* $\beta : \text{LgV} \to D$ (where for $x \in \text{LgV}$ with $\alpha(x) = A$ we have $\beta(x) \in D^A$), we evaluate every term $t \in \text{Terms}^A_\Sigma$ to a value $val_{(K_\Sigma, \sigma, \beta)}(t) \in D^A$, every formula $\varphi \in \text{Form}_\Sigma$ to a truth value $val_{(K_\Sigma, \sigma, \beta)}(\varphi) \in \{tt, \mathit{ff}\}$, and every update $u \in \text{Upd}_\Sigma$ to a state transformer $val_{(K_\Sigma, \sigma, \beta)}(u) : S \to S$ as defined in [Ben11, Figure 5.2].

We write $(K_\Sigma, \sigma, \beta) \models \varphi$ for $val_{(K_\Sigma, \sigma, \beta)}(\varphi) = tt$. A formula $\varphi \in \text{Form}_\Sigma$ is called *logically valid*, in symbols $\models \varphi$, iff $(K_\Sigma, \sigma, \beta) \models \varphi$ for all Kripke structures K_Σ, all states $\sigma \in S$, and all variable assignments β. Furthermore, we write $(K_\Sigma, \sigma) \models \varphi$ if $(K_\Sigma, \sigma, \beta) \models \varphi$ holds for all variable assignments

β; in particular, we write $(K_\Sigma, \sigma) \models \varphi$ for *closed* formulae φ. For closed terms t without program variables we write $val_{K_\Sigma}(t)$. $\Diamond$

The following example illustrating the semantics of Java DL updates originates from [RRR13].

Example 2.11 (Update Semantics)**.** Consider the formula $\{\mathtt{i} := \mathtt{j}+1\}\mathtt{i} \geq \mathtt{j}$. Evaluating $\{\mathtt{i} := \mathtt{j}+1\}\mathtt{i} \geq \mathtt{j}$ in a state σ is identical to evaluating the subformula $\mathtt{i} \geq \mathtt{j}$ in a state σ' which coincides with σ except for the value of $\mathtt{i}$ that is evaluated to the value of $val_{(K_\Sigma,\sigma,\beta)}(\mathtt{j}+1)$. Evaluation of the parallel update $\mathtt{i} := \mathtt{j} \parallel \mathtt{j} := \mathtt{i}$ in a state σ leads to the successor state σ' that is identical to σ except that the values of $\mathtt{i}$ and $\mathtt{j}$ are swapped. The parallel update $\mathtt{i} := 3 \parallel \mathtt{i} := 4$ has a *conflict* as $\mathtt{i}$ is assigned different values. In such a case the last occurring assignment $\mathtt{i} := 4$ overrides all previous ones of the same location variable. Evaluation of the formula $\{\mathtt{i} := \mathtt{j}\}\{\mathtt{j} := \mathtt{i}\}\varphi$ in a state σ results in evaluating φ in a state where $\mathtt{i}$ has the value of $\mathtt{j}$ in σ, and $\mathtt{j}$ remains unchanged. The update *skip*, the empty update, does not change the interpreting state. $\Diamond$

Definition 2.12 (Update Normal Form)**.** An update is in *normal form* if it has the shape $U_1 \parallel \cdots \parallel U_n$, $n \geq 0$, where each U_i is an elementary update and there is no conflict between U_i and U_j for any $i \neq j$. $\Diamond$

2.3 Symbolic Execution

Symbolic Execution (SE), in contrast to concrete execution, treats program variables, in particular program inputs, as symbols – as long as they are not assigned concrete values. Whenever the execution depends on the concrete, but unknown, value of a variable (in an if statement, for instance), execution splits into subbranches. Thus, the result of the symbolic execution of a program is a Symbolic Execution Tree (SET) in which each node represents a symbolic execution state. An SET resembles an "unrolled" CFG and may in principle, for instance in the presence of loops, be infinite. SE states track changes made to program locations in course of the execution (the *symbolic state*), as well as the constraints on (symbolic) values that lead to the execution of the current path (the *path condition*). Those notions are defined subsequently, following [Kin76; Häh+10; JH14; HHB14].

Definition 2.13 (Symbolic Execution State)**.** A *Symbolic Execution State* is a triple (U, C, φ) consisting of (1) the *symbolic state* $U \in \mathrm{Upd}_\Sigma$, an update in normal form with only closed terms as right sides, tracking changes made to program variables, (2) a set of *closed* Java DL formulae $C \in 2^{\mathrm{Form}_\Sigma}$ encoding the current *path condition*, and (3) a Java DL formula $\varphi \in \mathrm{Form}_\Sigma$ usually containing a modality, which we call the *program counter*, representing the Java code that remains to be executed after that state. $\Diamond$

We denote the (underspecified) set of all symbolic execution states for a given program *Prg* by $SEStates_{Prg}$. Symbolic execution of a program yields an SET consisting of SE states in $SEStates_{Prg}$. Complete symbolic execution trees for a program p with desired post condition φ are finite acyclic trees whose root is labeled with the node $(U_0, C_0, [p]\,\varphi)$ or $(U_0, C_0, \langle p\rangle\,\varphi)$, and whose leaves only contain the empty program counter (i.e., the formula *true*). Every child node is generated from its parent according to the semantics of the symbolically executed program statement.

Example 2.14. Figure 2.3 shows the partial Symbolic Execution Tree for the Java program in Listing 2.1. Program counters are abstracted as line number pointing to the next statement to execute, where the special number -1 refers to the end of the program. The tree is infinite, since it is unknown whether the initial value of $\mathtt{z}$ is smaller than that of $\mathtt{y}$, and, e.g., whether $\mathtt{z}$ is initially greater or equal than 0 (in this case, $\mathtt{z}+\mathtt{y}<\mathtt{y}$ would be false). An alternative to the unwinding steps used in the tree are loop invariants, which would make the tree finite. $\Diamond$

Symbolic Execution in KeY

The KeY theorem prover is based upon a *sequent calculus*. A sequent is a pair of sets of formulae $\Gamma, \Delta \subseteq 2^{\mathrm{Form}_\Sigma}$, the antecedent and the succedent, of the form $\Gamma \Longrightarrow \Delta$. Its semantics is defined by $\bigwedge_{\varphi\in\Gamma}\varphi \rightarrow \bigvee_{\psi\in\Delta}\psi$. A *sequent calculus rule* has one conclusion and zero or more premises. It is applied to a sequent s by matching its conclusion against s. The instantiated premises are then added as children of s, thus generating a proof tree [RRR13]. The rules in the KeY calculus not concerning symbolic execution correspond to usual rules of a sequent calculus for first-order logic like the calculus LK by Gentzen [Gen64]. In addition to those, the calculus of KeY contains a large set of rules

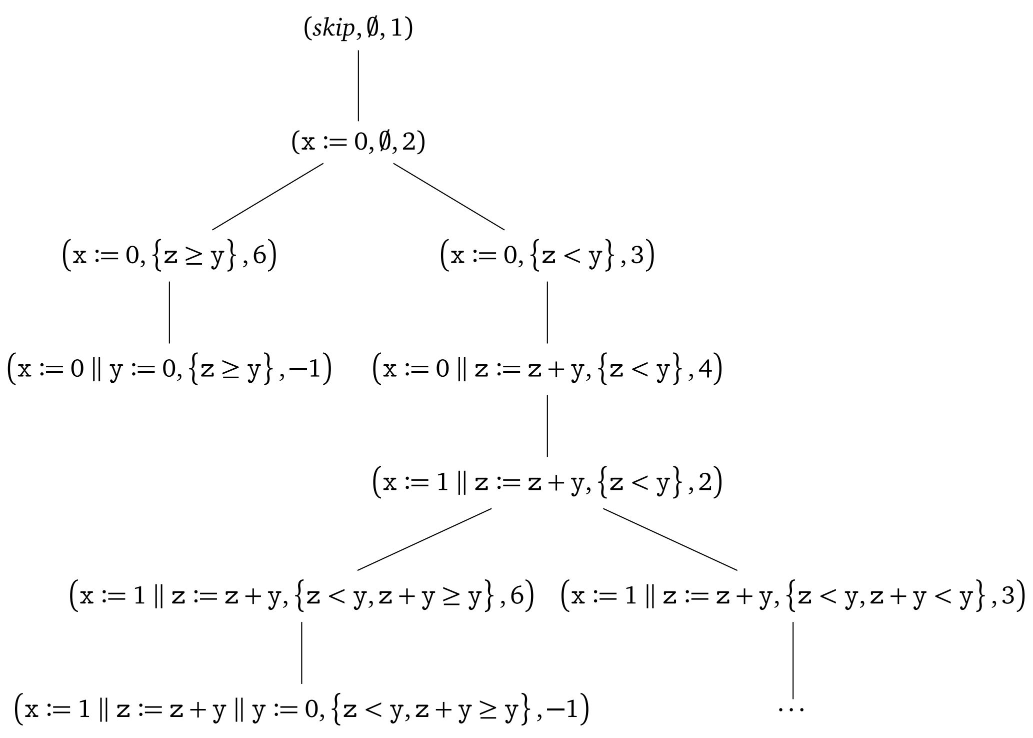

$(skip, \emptyset, 1)$
$(\mathrm{x} := 0, \emptyset, 2)$
$(\mathrm{x} := 0, \{\mathrm{z} \geq \mathrm{y}\}, 6)$
$(\mathrm{x} := 0, \{\mathrm{z} < \mathrm{y}\}, 3)$
$(\mathrm{x} := 0 \parallel \mathrm{y} := 0, \{\mathrm{z} \geq \mathrm{y}\}, -1)$
$(\mathrm{x} := 0 \parallel \mathrm{z} := \mathrm{z} + \mathrm{y}, \{\mathrm{z} < \mathrm{y}\}, 4)$
$(\mathrm{x} := 1 \parallel \mathrm{z} := \mathrm{z} + \mathrm{y}, \{\mathrm{z} < \mathrm{y}\}, 2)$
$(\mathrm{x} := 1 \parallel \mathrm{z} := \mathrm{z} + \mathrm{y}, \{\mathrm{z} < \mathrm{y}, \mathrm{z} + \mathrm{y} \geq \mathrm{y}\}, 6)$
$(\mathrm{x} := 1 \parallel \mathrm{z} := \mathrm{z} + \mathrm{y}, \{\mathrm{z} < \mathrm{y}, \mathrm{z} + \mathrm{y} < \mathrm{y}\}, 3)$
$(\mathrm{x} := 1 \parallel \mathrm{z} := \mathrm{z} + \mathrm{y} \parallel \mathrm{y} := 0, \{\mathrm{z} < \mathrm{y}, \mathrm{z} + \mathrm{y} \geq \mathrm{y}\}, -1)$
$\cdots$

Figure 2.3: Partial Symbolic Execution Tree for Listing 2.1.

dedicated to the treatment of modalities, that is of Java code. Ultimately, KeY transforms modalities into updates, and thus may be seen as a symbolic interpreter of Java code. Figure 2.4 shows some example calculus rules for symbolic execution. A sequent containing at least one modality corresponds to an SE state: Consider the sequent $\Gamma \implies \{U\}\,\varphi, \Delta$. We transform the sequent into the canonical form $\Gamma \cup \{\neg\psi : \psi \in \Delta\} \implies \{U\}\,\varphi$ by shifting all formulae of the succedent except for $\{U\}\,\varphi$ to the antecedent; the resulting sequent is logically equivalent to the original one. This sequent uniquely corresponds to the SE state $(U, \Gamma \cup \{\neg\psi : \psi \in \Delta\}, \varphi)$. Note that in principle, there could be other formulae in Δ that we could have chosen instead of $\{U\}\,\varphi$ as a source for the extraction of the symbolic state and program counter; $\neg\{U\}\,\varphi$ would then become a part of the path condition. Thus, a sequent may be mapped to several different SE states. However, for most cases resulting from correctness proofs of Java programs, the desired mapping can be uniquely determined, since there is only one formula in the succedent which contains a modality.

Notation 2.15. We write $\vdash\varphi$ to express that the sequent $\implies \varphi$ is provable within the sequent calculus of KeY. For sets of formulae Δ, $\vdash\Delta$ means that $\implies \bigwedge\Delta$ is provable, where

$$\bigwedge\{\varphi_1, \varphi_2, \dots, \varphi_n\} := \varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_n.$$

Equivalently, $\bigvee\Delta$ represents the formula resulting from a disjunction of the contained elements.

Furthermore, we write $\varphi \equiv \psi$ to express that φ is logically equivalent to ψ, i.e. φ is true in a model $(K_\Sigma, \sigma, \beta)$ iff ψ is true in $(K_\Sigma, \sigma, \beta)$. Again, $\Gamma \equiv \Delta$ means $\bigwedge\Gamma \equiv \bigwedge\Delta$ for sets of formulae Γ, Δ. $\Diamond$

$$\text{assignLocal}\ \frac{\Gamma \Longrightarrow \{U\}\{\mathtt{a} := t\}[\pi\ \omega]\,\varphi, \Delta}{\Gamma \Longrightarrow \{U\}[\pi\ \mathtt{a} = t;\ \omega]\,\varphi, \Delta}$$

$$\text{assignField}\ \frac{\Gamma \Longrightarrow \{U\}\left\{\mathtt{heap} := \mathit{store}\left(\mathtt{heap}, o, \mathtt{f}, t\right)\right\}[\pi\ \omega]\,\varphi, \Delta}{\Gamma \Longrightarrow \{U\}[\pi\ o.\mathtt{f} = t;\ \omega]\,\varphi, \Delta}$$

$$\text{conditional}\ \frac{\Gamma \Longrightarrow \{U\}\,\mathit{if}\left(\mathit{exp} \doteq \mathit{true}\right)\mathit{then}\,([\pi\ p_1\ \omega]\,\varphi)\,\mathit{else}\,([\pi\ p_2\ \omega]\,\varphi), \Delta}{\Gamma \Longrightarrow \{U\}[\pi\ \mathtt{if}\ (\mathit{exp})\ p_1\ \mathtt{else}\,p_2;\ \omega]\,\varphi, \Delta}$$

$$\text{unwindLoop}\ \frac{\Gamma \Longrightarrow \{U\}\left[\pi\ \mathtt{if}\,(\mathit{exp})\,\{\mathtt{p}';\mathtt{while}\ (\mathit{exp})\,p\}\ \omega\right]\varphi, \Delta}{\Gamma \Longrightarrow \{U\}[\pi\ \mathtt{while}\ (\mathit{exp})\,p;\ \omega]\,\varphi, \Delta}$$

$$\text{expandMethod}\ \frac{\begin{array}{l}\Gamma \Longrightarrow \{U\}[\pi\ \mathtt{method-frame(result} = \mathtt{r},\ \mathtt{this} = o)\ :\\ \qquad\qquad \{\ \mathit{body}\,(\mathtt{m}, A)\ \}\ \omega]\,\varphi, \Delta\\ \Gamma \Longrightarrow \{U\}\,\mathit{exactInstance}_A\,(o), \Delta\end{array}}{\Gamma \Longrightarrow \{U\}[\pi\ \mathtt{r}{=}o.\mathtt{m}();\ \omega]\,\varphi, \Delta}$$

$$\text{emptyModality}\ \frac{\Gamma \Longrightarrow \{U\}\,\varphi, \Delta}{\Gamma \Longrightarrow \{U\}[\,]\,\varphi, \Delta}$$

Figure 2.4: Selected rules of the KeY calculus for symbolic execution

3 A Lattice Model for Symbolic Execution

The problem to be solved in this thesis and particularly in this chapter is the merging of two branches in a symbolic execution tree, the last states of which have the same program counter. Figure 3.1 illustrates this situation, Figure 3.2 shows an example in KeY/Java DL syntax for joining two nodes after an if-statement. Our goal is to join SE nodes (U_1, C_1, φ) and (U_2, C_2, φ) with the same program counter φ to a new state (U^*, C^*, φ) that we call *join state*. This gives rise to two orthogonal questions:

(1) How and when during symbolic execution of a program can we detect suitable branches to join?
(2) What are the characteristics of sensible instantiations for U^* and C^*, and how can we construct them?

Question (1) addresses the integration of our techniques into the symbolic execution process; it would be desirable to automate the joining of branches such that a user presses "play" whereupon KeY outputs a DAG with suitable branches having been joined. A complete generation of the SET with subsequent pruning and joining steps is undesirable: amongst the disadvantages of this naive approach is the obvious performance overhead. Thus, branch joining should ideally be incorporated into the proof generation process. We refer to this question in Section 4.4. Question (2) concerns the actual joining of two branches, the computation of join states from two parents. In particular, we propose a general lattice framework for symbolic execution, with the property that join techniques conforming with our formal framework preserve the soundness of the KeY calculus. Subsequently, we fix the foundations of our framework by narrowing the gap between Symbolic Execution and Abstract Interpretation.

3.1 Concretization and Weakening

Symbolic Execution can be regarded, at least to some extent, as a case of Abstract Interpretation [CC77]. Each SE state describes a potentially infinite set of *concrete states*; only if all locations are set to concrete values, i.e. do not depend on symbolic input values, is the set of described concrete states a singleton set. However, abstract interpretation demands a complete semilattice with join operation, partial order, least and top element, which is usually not defined for SE states. Subsequently, we define a concretization function from SE states to concrete states, as well as a partial order relation between SE states. In Section 3.3, we furthermore define join operations on SE states, which allows us to stipulate lattice structures also for symbolic execution.

Definition 3.1 (Concrete Execution States)**.** A *concrete execution state* is a pair (σ, φ) consisting of (i) a Kripke state $\sigma : \mathrm{PV} \rightarrow D$ s.th. if $\sigma(\mathtt{a}) = x$ and $\alpha(\mathtt{a}) = A$, it holds that $\delta(x) = A$, and (ii) a Java DL formula φ, the program counter, usually containing a modality with the program that remains to be executed p. We denote the set of all concrete execution states for a program *Prg* by $\mathit{ConcrStates}_{\mathit{Prg}}$. ◊

We now can introduce a concretization function from SE states to concrete states based on the valuation function *val* (→ Definition 2.10).

Definition 3.2 (Concretization Function)**.** Let $s_{SE} = (U, C, \varphi) \in \mathit{SEStates}_{\mathit{Prg}}$. The *concretization function concr* maps s_{SE} to a set of concrete states in $2^{\mathit{ConcrStates}_{\mathit{Prg}}}$ where

$$\begin{aligned}\mathit{concr}(s_{SE}) :=\Big\{\big(\sigma', \varphi\big) : \sigma' = \mathit{val}_{(K_\Sigma,\sigma)}(U)(\sigma) \\ \wedge K_\Sigma = (D, \delta, I, S, \rho) \text{ is a Kripke structure} \wedge (K_\Sigma, \sigma) \models C\Big\}\end{aligned}$$

If the program counter φ is clear from the context, we also write $\sigma' \in \mathit{concr}(s_{SE})$ for $(\sigma', \varphi) \in \mathit{concr}(s_{SE})$. ◊

For each possible value that a term, that is the right side in the symbolic state for a program variable x, can attain in any Kripke structure under the given constraints C, the concretization as defined above contains an assignment function mapping x to exactly this value. Thus, the set $\mathit{concr}(s_{SE})$ contains exactly the concrete states that are described by the SE state s_{SE}.

Definition 3.2 facilitates the natural definition of a partial order relation between SE states: A *weakening relation* expressing that one state describes more concrete states than another one.

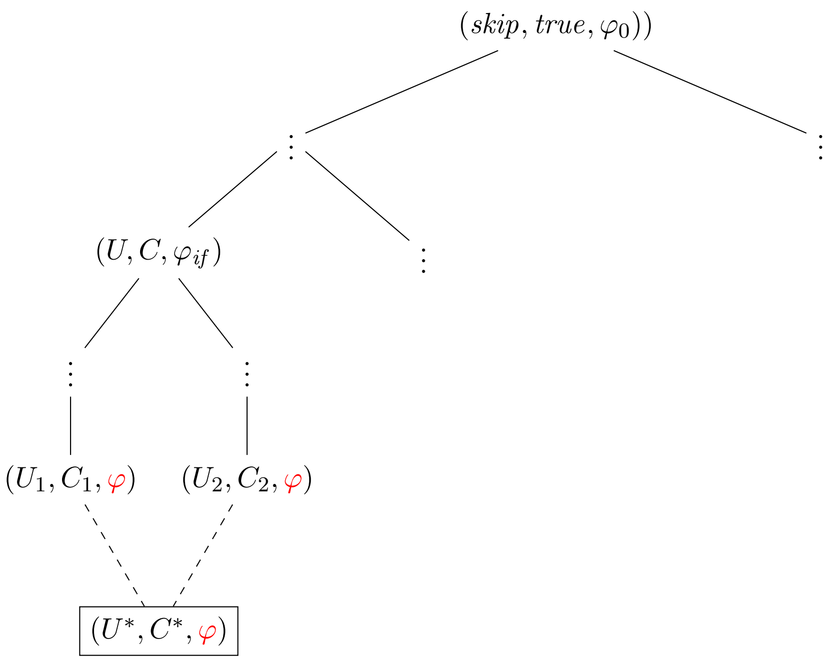


Figure 3.1: Joining two branches in an abstract SET after an `if` statement.

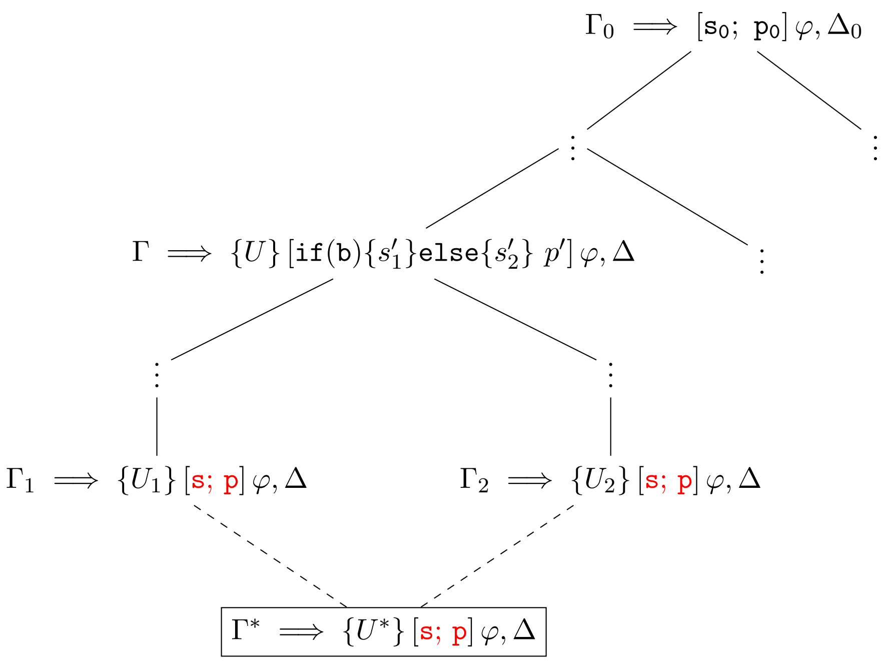

$\Gamma_0 \implies [\mathtt{s}_0;\ \mathtt{p}_0]\,\varphi, \Delta_0$
$\Gamma \implies \{U\}\,[\mathtt{if(b)}\{s'_1\}\mathtt{else}\{s'_2\}\ p']\,\varphi, \Delta$
$\Gamma_1 \implies \{U_1\}\,[\mathtt{s};\ \mathtt{p}]\,\varphi, \Delta$
$\Gamma_2 \implies \{U_2\}\,[\mathtt{s};\ \mathtt{p}]\,\varphi, \Delta$
$\Gamma^* \implies \{U^*\}\,[\mathtt{s};\ \mathtt{p}]\,\varphi, \Delta$

Figure 3.2: Joining two branches in a KeY proof tree after an `if` statement.

Definition 3.3 (Weakening Relation)**.** Let $s_1, s_2 \in SEStates_{Prg}$ be two SE states. We say that s_2 is *weaker* than, or a *weakening of*, s_1 and write $s_1 \lesssim s_2$ if and only if $concr(s_1) \subseteq concr(s_2)$. ◊

Note that Definition 3.3, along with Definition 3.2, implies that a state s_2 can only be a weakening of a state with *satisfiable path condition* s_1 if they have the same program counters. If the path condition of s_1 is unsatisfiable, $concr(s_1)$ equals the empty set; therefore, any state will be a weakening of a state with unsatisfiable path condition, in particular including states with a different program counter. Usually, we assume that all path conditions are satisfiable. The following example illustrates this concept.

Example 3.4. Consider the SE state $s_{SE} := \left(\mathtt{x} := \mathtt{y}, \mathtt{y} > 0, \varphi\right)$. The set of concretizations for s_{SE} is $concr(s) = \{(\sigma, \varphi) : \sigma(\mathtt{z}) > 0\}$. One intuitive weakening of s_{SE} is obtained by weakening the constraint on the program variable $\mathtt{y}$ such that the value 0 is also covered, resulting in $s'_{SE} = \left(\mathtt{x} := \mathtt{y}, \mathtt{y} \geq 0, \varphi\right)$. Indeed it holds for the set of concretizations of s'_{SE}, $concr\left(s'_{SE}\right) = \{(\sigma, p) : \sigma(\mathtt{z}) \geq 0\}$, that $concr\left(s'_{SE}\right) \supseteq concr(s_{SE})$, so s'_{SE} is a weakening of s_{SE}. Note that $s''_{SE} = \left(\mathtt{x} := \mathtt{y}, \mathtt{y} \geq 0, \varphi'\right)$, for $\varphi \neq \varphi'$, is not weaker than s_{SE}. ◊

Lemma 3.6 shows that the relation $\lesssim$ is actually a partial order relation. We slightly generalized the antisymmetry condition in the lemma, since strict syntactical equality on SE states is too strong. Assume that the symbolic state of a state s_1 contains the elementary update $\mathtt{x} := t$, whereas the symbolic state s_2 contains the elementary update $\mathtt{x} := if(true)\,then(t)\,else(t')$. s_1 and s_2 are obviously syntactically different, whereas all Kripke models map the program variable $\mathtt{x}$ to the value of the same term t. Therefore, we use the equality of the *concretizations* of the states, as defined below, rather than the syntactical equality.

Definition 3.5 (Equality of Concretizations)**.** The equivalence relation $\overset{concr}{=} \subseteq SEStates_{Prg} \times SEStates_{Prg}$ is defined by $s_1 \overset{concr}{=} s_2 \iff concr(s_1) = concr(s_2)$. ◊

It is obvious that $\overset{concr}{=}$ is an equivalence relation, since it employs the usual equality $=$ on sets in a straightforward manner. Using $\overset{concr}{=}$, we can formulate the subsequent lemma.

Lemma 3.6. *The relation $\lesssim \in SEStates_{Prg} \times SEStates_{Prg}$ is a partial order relation.* ◊

Proof. We have to show the following properties of $\lesssim$, for $s, s_1, s_2, s_3 \in SEStates_{Prg}$:

(1) reflexivity: $s \lesssim s$

(2) antisymmetry: $s_1 \lesssim s_2 \land s_2 \lesssim s_1 \rightarrow s_1 \overset{concr}{=} s_2$

(3) transitivity: $s_1 \lesssim s_2 \land s_2 \lesssim s_3 \rightarrow s_1 \lesssim s_3$

All properties follow from ordinary set theory. Property (1) follows from $concr(s) = concr(s)$ and, therefore, $concr(s) \subseteq concr(s)$. For (2), assume that $s_1 \lesssim s_2$ and $s_2 \lesssim s_1$, i.e. $concr(s_1) \subseteq concr(s_2)$ and $concr(s_2) \subseteq concr(s_1)$. From that, we obtain $concr(s_1) = concr(s_2)$. For (3), assume that $s_1 \lesssim s_2$ and $s_2 \lesssim s_3$, i.e. $concr(s_1) \subseteq concr(s_2)$ and $concr(s_2) \subseteq concr(s_3)$. Then $concr(s_1) \subseteq concr(s_3)$ follows from the transitivity of the subset relation. □

Having defined a *semantic* condition for two SE states being in the weakening relation, we now aim for a corresponding *logical representation*. This condition is then employed in the implementation part (→ Section 4.3) to facilitate automatic proofs of the weakening relation between two SE states with support of the KeY system.

Definition 3.7 (Logical Representation of Weakening)**.** Let $s_1 = (U_1, C_1, \varphi_1) \in SEStates_{Prg}$ and $s_2 = (U_2, C_2, \varphi_2) \in SEStates_{Prg}$ be two symbolic execution states. We say that s_2 is *logically weaker* than, or a *logical weakening* of, s_1 and write $s_1 \lesssim_{log} s_2$, if the following formula $\varphi_{\lesssim_{log}}$ is *provable* for s_1 and s_2, where $\bar{c}$ are the new constants introduced in s_2 (and not contained in s_1), $\bar{\mathtt{x}}$ are all program variables contained in U_1, U_2, C_1 and C_2, $\bar{v}$ is a tuple of fresh logical variables of the same length and types as $\bar{c}$, and P is a new predicate of suitable type:

$$\varphi_{\lesssim_{log}} := \begin{cases} \forall \bar{v}; \left(\left(\bigwedge C_2 \rightarrow \{U_2\}\, P(\bar{\mathtt{x}})\right)[\bar{v}/\bar{c}]\right) \rightarrow \left(\bigwedge C_1 \rightarrow \{U_1\}\, P(\bar{\mathtt{x}})\right) & \varphi_1 = \varphi_2 \\ false & \text{otherwise} \end{cases}$$ ◊

Note that we claim provability instead of the mere truth of the formula $\varphi_{\lesssim_{log}}$. Since logical weakening is meant to be used in the KeY system, it is important that the formula can actually be proven.

Subsequently, we establish two lemmas as well as a proposition following from those, which provide affirmations about the relation between semantic and logical weakening. As we will see, logical weakening is an equivalent of semantic weakening. The conditions on the path condition of the second SE state s_2 in the lemmas correspond to the property (SEL5) introduced later in Definition 3.12.

Lemma 3.8. *Let $s_1 = (U_1, C_1, \varphi_1)$ and $s_2 = (U_2, C \wedge C_{ax}, \varphi_2) \in SEStates_{Prg}$ be two SE states such that $\vdash \bigwedge C_1 \to C$ and $\vdash \exists \overline{v}; (C_{ax}[\overline{v}/\overline{c}])$ where $\overline{c}$ is a tuple of Skolem constants introduced in s_2 (and not present in s_1 and C). Then, semantic weakening implies logical weakening, i.e., it holds that*

$$s_1 \lesssim s_2 \implies s_1 \lesssim_{log} s_2 \qquad \Diamond$$

Proof. We have to show that $\varphi_{\lesssim_{log}}$ is provable. Consider the following proof:

$$\dfrac{\dfrac{\forall \overline{v}; ((C \wedge C_{ax} \to \{U_2\}\, P(\overline{\mathtt{x}}))[\overline{v}/\overline{c}]), \bigwedge C_1 \implies \{U_1\}\, P(\overline{\mathtt{x}})}{\forall \overline{v}; ((C \wedge C_{ax} \to \{U_2\}\, P(\overline{\mathtt{x}}))[\overline{v}/\overline{c}]) \implies \bigwedge C_1 \to \{U_1\}\, P(\overline{\mathtt{x}})}\, {\to}\mathrm{R}}{\implies \forall \overline{v}; ((C \wedge C_{ax} \to \{U_2\}\, P(\overline{\mathtt{x}}))[\overline{v}/\overline{c}]) \to \left(\bigwedge C_1 \to \{U_1\}\, P(\overline{\mathtt{x}})\right)}\, {\to}\mathrm{R}$$

To close the proof under the assumption that C_1 is satisfiable, we have to find suitable instantiations for the variables $\overline{v}$. Let $\Sigma_1 \subseteq \Sigma_2$ be two signatures, where Σ_2 results from Σ_1 by adding the constants $\overline{c}$, and Σ_1 is a suitable signature for s_1 without the constants $\overline{c}$. Furthermore, let $\left(K_{\Sigma_1}, \sigma'\right)$ be an arbitrary Java DL model satisfying C_1. Obviously, $\sigma = val_{\left(K_{\Sigma_1}, \sigma'\right)}(U_1)(\sigma') \in concr(s_1)$, and by $s_1 \lesssim s_2$, $\sigma \in concr(s_2)$. We expand K_{Σ_1} to a structure K_{Σ_2} in the signature Σ_2 by choosing the interpretation of the constants $\overline{c}$ in K_{Σ_2} such that $val_{\left(K_{\Sigma_2}, \sigma'\right)}(U_2)(\sigma') = \sigma$ and $\left(K_{\Sigma_2}, \sigma'\right) \models C_{ax}$. Satisfying the condition $\left(K_{\Sigma_2}, \sigma'\right) \models C_{ax}$ is possible due to $\vdash \exists \overline{v}; (C_{ax}[\overline{v}/\overline{c}])$. The condition $val_{\left(K_{\Sigma_2}, \sigma'\right)}(U_2)(\sigma') = \sigma$ can be satisfied since otherwise, $s_1 \lesssim s_2$ would not hold in general. Now let $\overline{d} := I_{K_{\Sigma_2}}(\overline{c})$. Then there are closed terms $\overline{t}$ of the signature Σ_1 such that $val_{K_{\Sigma_2}}\left(\overline{t}\right) = \overline{d}$ due to $val_{\left(K_{\Sigma_2}, \sigma'\right)}(U_2)(\sigma') = \sigma$. We now continue our proof (we omit $\forall \overline{v}; \left(\left(\bigwedge C_2 \to \{U_2\}\, P(\overline{\mathtt{x}})\right)[\overline{v}/\overline{c}]\right)$ after the $\exists$L application as well as $\{U_1\}\, P(\overline{\mathtt{x}})$ in the first and second leaf branch, and implicitly eliminate the conjunction from $\bigwedge C_1$ in the antecedent):

$$\dfrac{\dfrac{\dfrac{C_1 \implies C \qquad C_1 \implies C_{ax}\left[\overline{t}/\overline{c}\right]}{C_1 \implies (C \wedge C_{ax})\left[\overline{t}/\overline{c}\right], \{U_1\}\, P(\overline{\mathtt{x}})}\, \wedge\mathrm{R} \qquad C_1, (\{U_2\}\, P(\overline{\mathtt{x}}))\left[\overline{t}/\overline{c}\right] \implies \{U_1\}\, P(\overline{\mathtt{x}})}{(C \wedge C_{ax} \to \{U_2\}\, P(\overline{\mathtt{x}}))\left[\overline{t}/\overline{c}\right], \bigwedge C_1 \implies \{U_1\}\, P(\overline{\mathtt{x}})}\, {\to}\mathrm{L}}{\forall \overline{v}; ((C \wedge C_{ax} \to \{U_2\}\, P(\overline{\mathtt{x}}))[\overline{v}/\overline{c}]), \bigwedge C_1 \implies \{U_1\}\, P(\overline{\mathtt{x}})}\, \exists\mathrm{L}$$
$$\vdots$$

The branch $C_1 \implies C$ can be closed since $\vdash \bigwedge C_1 \to C$. Furthermore, we can close the second branch $C_1 \implies C_{ax}\left[\overline{t}/\overline{c}\right]$ since, due to our reasoning above, any model satisfying C_1 also satisfies $C_{ax}\left[\overline{t}/\overline{c}\right]$ for the chosen terms $\overline{t}$. For the remaining branch, let us assume that the model $\left(K_{\Sigma_2}, \sigma'\right)$ interprets the predicate P such that $(\{U_2\}\, P(\overline{\mathtt{x}}))\left[\overline{t}/\overline{c}\right]$ (otherwise, we could close the branch), i.e.

$$\left(K_{\Sigma_2}, val_{\left(K_{\Sigma_2}, \sigma'\right)}(U_2)\left(\sigma'\right)\right) \models P(\overline{x}).$$

Since we chose $val_{\left(K_{\Sigma_2}, \sigma'\right)}(U_2)(\sigma') = \sigma$, it holds that

$$\begin{aligned} & \left(K_{\Sigma_2}, val_{\left(K_{\Sigma_1}, \sigma'\right)}(U_1)\left(\sigma'\right)\right) \models P(\overline{x}) \\ \iff & \left(K_{\Sigma_2}, \sigma'\right) \models \{U_1\}\, P(\overline{x}) \end{aligned}$$

which is furthermore, since U_1 is in the old language Σ_1, equivalent to $\left(K_{\Sigma_1}, \sigma'\right) \models \{U_1\}\, P(\overline{x})$. Therefore, we can also close the third branch of the proof.

□

In the above proof, there are some potential incompleteness issues since we sometimes argue for the provability of a sequent using its truth. At these places, we make use of the *relative* completeness of the KeY calculus. In a complete calculus, all formulae that are *valid* can actually be *proven* within the calculus. Since the logic of KeY supports arithmetic, the calculus is inherently incomplete by Gödel's first incompleteness theorem. However, by the rules for arithmetic implemented in KeY, it is possible to prove almost any arithmetical statement that occurs *in practice* [BHS07].

Lemma 3.9. *Let $s_1 = (U_1, C_1, \varphi_1)$ and $s_2 = (U_2, C \land C_{ax}, \varphi_2) \in SEStates_{Prg}$ be two SE states such that $\vdash \bigwedge C_1 \to C$ and $\vdash \exists \overline{v}; (C_{ax}[\overline{v}/\overline{c}])$ where $\overline{c}$ is a tuple of Skolem constants introduced in s_2 (and not present in s_1 and C). Then, logical weakening implies semantic weakening, i.e. it holds that*

$$s_1 \lesssim_{log} s_2 \implies s_1 \lesssim s_2. \qquad \Diamond$$

Proof. We assume $s_1 \lesssim_{log} s_2$, i.e., the formula $\varphi_{\lesssim_{log}}$ is provable for s_1, s_2 according to Definition 3.7. Let $\sigma \in concr(s_1)$; we have to show that $\sigma \in concr(s_2)$. Since $\sigma \in concr(s_1)$, there is a model $(K_\Sigma, \sigma') \models C_1$. Furthermore, $\{U_1\} P(\overline{x})$ is satisfiable, but not valid. Any proof of $\varphi_{\lesssim_{log}}$ must therefore find instantiations $\overline{t}$ for $\overline{v}$ such that, after an application of the implication left rule, the proof

$$\to L \; \frac{C_1 \implies (\bigwedge C_2)[\overline{t}/\overline{c}], \{U_1\} P(\overline{x}) \qquad C_1, (\{U_2\} P(\overline{x}))[\overline{t}/\overline{c}] \implies \{U_1\} P(\overline{x})}{C_1, (\bigwedge C_2)[\overline{t}/\overline{c}] \to (\{U_2\} P(\overline{x}))[\overline{t}/\overline{c}] \implies \{U_1\} P(\overline{x})}$$

can be closed. Consider the right branch

$$C_1, (\{U_2\} P(\overline{x}))[\overline{t}/\overline{c}] \implies \{U_1\} P(\overline{x})$$

or equivalently

$$(\{U_2\} P(\overline{x}))[\overline{t}/\overline{c}] \implies \bigwedge C_1 \to \{U_1\} P(\overline{x})$$

Since the succedent is not valid, it must hold that for *any model* satisfying C_1, there are instantiations $\overline{t}$ for the new constants such that the right sides of U_1, under the constraints C_1, are instances of the right sides of U_2, namely those resulting from the substitution $U_2[\overline{t}/\overline{c}]$. In particular, $\sigma = val_{(K_\Sigma, \sigma')}(U_1)(\sigma')$ must also be contained in $concr(s_2)$. □

Remark 3.10. Consider the left branch $\to L$ application in Lemma 3.9:

$$C_1 \implies (\bigwedge C_2)[\overline{t}/\overline{c}], \{U_1\} P(\overline{x}).$$

Since C_1 is satisfiable, $\{U_1\} P(\overline{x})$ is not valid, and $\bigwedge C_2$ is equivalent to a formula $C \land C_{ax}$ such that $C_1 \to C$ and $\vdash \exists \overline{v}; (C_{ax}[\overline{t}/\overline{c}])$, the formula $\{U_1\} P(\overline{x})$ is irrelevant for a proof of the sequent and we obtain, since the $\overline{c}$ do not occur in C_1, the w.r.t. provability equivalent sequent

$$C_1 \implies C \land (C_{ax}[\overline{t}/\overline{c}]).$$

Therefore, either $(C_{ax})[\overline{t}/\overline{c}]$ must be provable or C_1 must imply $(C_{ax})[\overline{t}/\overline{c}]$. This characterizes both situations occurring in our join operations introducing new constants: One of those introduces axioms like $c \geq 0$ which are, after the substitution and instantiation, provable without C_1. The other one constructs C_{ax} as a conjunction of formulae of the form

$$(\bigwedge C_1 \to c_i \doteq \{U_1\} \mathrm{x}_i) \land (\bigwedge C' \to c_i \doteq \{U'\} \mathrm{x}_i)$$

that are only provable based on the knowledge of C_1 (and C'). $\Diamond$

The following proposition, an easy conclusion from the Lemmas 3.8 and 3.9, provides an equivalence result for semantic and logical weakening.

Proposition 3.11 (Weak Equivalence of Semantic and Logical Weakening)**.** *Let $s_1 = (U_1, C_1, \varphi_1)$ and $s_2 = (U_2, C \land C_{ax}, \varphi_2) \in SEStates_{Prg}$ be two SE states such that $\vdash \bigwedge C_1 \to C$ and $\vdash \exists \overline{v}; (C_{ax}[\overline{v}/\overline{c}])$ where $\overline{c}$ is a tuple of Skolem constants introduced in s_2 (and not present in s_1 and C). Then, semantic and logical weakening coincide, i.e. it holds that*

$$s_1 \lesssim s_2 \iff s_1 \lesssim_{log} s_2. \qquad \Diamond$$

Proof. The proposition follows immediately from Lemmas 3.8 and 3.9. □

Proposition 3.11 facilitates the implementation of checks in the KeY system verifying the soundness of the implemented join rules for every application. The concrete details of the implementation are outlined in Section 4.3.

3.2 The General Lattice Model

The core of our formal framework for embedding join operations into symbolic execution is a family of join-semilattices parametric in a join operation. The partial order induced by the join operation is constrained by the semantic weakening relation (→ Definition 3.3).

Definition 3.12 (Induced Join-Semilattices of SE States)**.** Let $\dot{\sqcup} : \mathit{SEStates}_{Prg} \times \mathit{SEStates}_{Prg} \to \mathit{SEStates}_{Prg}$ be an operation on SE states. Then, we call the family of structures

$$\mathfrak{L}_{Prg} := \left\{ \left(S_\varphi, \dot{\sqcup}\right)_\varphi \middle| \left(U, C, \varphi'\right) \in S_\varphi \iff \left(\left(U, C, \varphi'\right) \in \mathit{SEStates}_{Prg} \text{ and } \varphi' = \varphi\right) \right\}$$

the *induced family of join-semilattices for* $\dot{\sqcup}$, each structure $(S, \dot{\sqcup})_\varphi \in \mathfrak{L}_{Prg}$ the *induced join-semilattice for* $\dot{\sqcup}$ *and* φ, and the relation

$$\begin{aligned} &\preceq \;\subseteq \mathit{SEStates}_{Prg} \times \mathit{SEStates}_{Prg} \text{ where} \\ &a \preceq b \iff \begin{cases} a \,\dot{\sqcup}\, b \overset{concr}{=} b \text{ and} \\ \quad a, b \text{ have the same program counter} \end{cases} \end{aligned}$$

the *induced partial order relation* for $\dot{\sqcup}$, if the properties (SEL1) to (SEL5) are satisfied for $a = (U_1, C_1, \varphi),\ b = (U_2, C_2, \varphi),\ c = (U_3, C_3, \varphi) \in \mathit{SEStates}_{Prg}$:

(SEL1)	*Idempotency:*	$a \,\dot{\sqcup}\, a \overset{concr}{=} a$.
(SEL2)	*Commutativity:*	$a \,\dot{\sqcup}\, b \overset{concr}{=} b \,\dot{\sqcup}\, a$
(SEL3)	*Associativity:*	$(a \,\dot{\sqcup}\, b) \,\dot{\sqcup}\, c \overset{concr}{=} a \,\dot{\sqcup}\, (b \,\dot{\sqcup}\, c)$
(SEL4)	*Correctness:*	$a \preceq b$ implies $a \lesssim b$
(SEL5)	*Conservativity:*	$a \preceq b$ implies that C_2 is logically equivalent to a formula $C \wedge C_{ax}$, where $\vdash \bigwedge C_1 \to C$, C does not contain uninterpreted Skolem constants not occurring in a and, if $\overline{c}$ are all uninterpreted Skolem constants in C_{ax} not contained in a, the formula $\exists \overline{v}; (C_{ax}[\overline{v}/\overline{c}])$ is provable. ◊

Note that we could replace $\lesssim$ in (SEL4) by $\lesssim_{log}$ by Proposition 3.11. The properties (SEL1) to (SEL3) are obviously desirable for any join operation. If we join two states, we usually do not want lose precision by strictly weakening program variables that evaluate to the same value in both states; this motivates the idempotency property. Likewise, we do not want the order of the join to influence the result, which motivates the commutativity and associativity properties. However, these properties are also standards for the definition of a semilattice (→ Definition 2.3). Property (SEL4) makes use of the semantic weakening relation (→ Definition 3.3). We call this property "correctness" since it allows, along with (SEL5), for proving the correctness Theorem 3.16.

The subsequent lemma that is employed in the proof of the correctness theorem concerns the interplay between a join operation and its corresponding induced partial order relation.

Lemma 3.13. *Let $a, b \in \mathit{SEStates}_{Prg}$ be two SE states with the same program counter, and let $\dot{\sqcup}$ be an idempotent and associative join operation with the induced partial order relation $\preceq$. Then, for any $a, b \in \mathit{SEStates}_{Prg}$ with the same program counter, it holds that $a \preceq b$ if and only if there is a state c with the same program counter as a, b such that $b = a \,\dot{\sqcup}\, c$.* ◊

Proof. The direction "$\Longrightarrow$" follows from a choice of $c := b$. By definition of $\preceq$, it holds that $a \preceq b$ iff $a \,\dot{\sqcup}\, b \overset{concr}{=} b$. "$\Longleftarrow$": By the reflexivity of the equivalence relation $\overset{concr}{=}$, we have $a \,\dot{\sqcup}\, c \overset{concr}{=} a \,\dot{\sqcup}\, c$. Then by idempotency of $\dot{\sqcup}$, we obtain $(a \,\dot{\sqcup}\, a) \,\dot{\sqcup}\, c \overset{concr}{=} a \,\dot{\sqcup}\, c$, and then by associativity $a \,\dot{\sqcup}\, (a \,\dot{\sqcup}\, c) \overset{concr}{=} a \,\dot{\sqcup}\, c$. Since $a \,\dot{\sqcup}\, c = b$, we have $a \,\dot{\sqcup}\, b \overset{concr}{=} b$, and therefore $a \preceq b$. □

Note that in the above proof, we made use of the idempotency and associativity properties of the induced join-semilattices. So, besides the intuitive motivation why those properties are desirable, those standard lattice properties also play a vital role for the proof of the central soundness theorem. The following lemma is needed in the proof of the soundness theorem.

Lemma 3.14. *Let $\Sigma_1 \subseteq \Sigma_2$ be two signatures, where Σ_2 results from Σ_1 by the addition of new constants $\overline{c}$. Let $\varphi \in \mathrm{Form}_{\Sigma_1}$ be a Java DL formula that is true in all models of the set*

$$\mathcal{M}_1 := \left\{K_{\Sigma_2} : \left(K_{\Sigma_2}, s\right) \models C \wedge C_{ax}\right\} \times S,$$

where $C \in \mathrm{Form}_{\Sigma_1}$, $C_{ax} \in \mathrm{Form}_{\Sigma_2}$ such that $\vdash \exists\overline{v}; (C_{ax}[\overline{v} / \overline{c}])$ and S is a set of Kripke states in the signature Σ_1. Then φ is also true in all models in the set

$$\mathcal{M}_2 := \left\{K_{\Sigma_1} : \left(K_{\Sigma_1}, s\right) \models C\right\} \times S \qquad \Diamond$$

Proof. Assume that φ does hold in the set $\mathcal{M}_1$, but does not hold for all formulae in $\mathcal{M}_2$. Let $\left(K_{\Sigma_1}, \sigma\right) \in \mathcal{M}_2$ be a model such that $\left(K_{\Sigma_1}, \sigma\right) \not\models \varphi$. From a proof of $\exists\overline{v}; (C_{ax}[\overline{v} / \overline{c}])$, we obtain a tuple $\overline{t}$ as instantiations of $\overline{v}$ for which $C_{ax}\left[\overline{t} / \overline{c}\right]$ holds in all models in Σ_1. Then we can expand K_{Σ_1} to a structure K_{Σ_2} in the signature Σ_2 where the constants $\overline{c}$ are interpreted according to the terms $\overline{t}$. Since $\left(K_{\Sigma_2}, \sigma\right)$ still satisfies C, it holds that $\left(K_{\Sigma_2}, \sigma\right) \models C \wedge C_{ax}$; since however φ is in the old language of Σ_1, and therefore its interpretation in $\left(K_{\Sigma_2}, \sigma\right)$ equals that of $\left(K_{\Sigma_1}, \sigma\right)$, we still have $\left(K_{\Sigma_2}, \sigma\right) \not\models \varphi$, which is a contradiction to $\left(K_{\Sigma_2}, \sigma\right) \in \mathcal{M}_1$. $\square$

Our proposed join rule, having one premise and more than one conclusion, is a *defocusing rule* in the sense of [CS00; Fin05]. Its application on two sequents therefore effectively renders a proof tree into a DAG and thereby realizes the main goal of this thesis. Subsequently, before introducing the main theorem, we provide a definition for defocusing rules.

Definition 3.15 (Defocusing Rule)**.** Let $S, S_1, \ldots, S_n$ be sequents in the language of Java DL. A *defocusing rule* is a rule of the form

$$\frac{S}{S_1 \qquad S_2 \qquad \ldots \qquad S_n}$$

We designate defocusing rules by the double bar separating the premise from the conclusions. $\Diamond$

Theorem 3.16 (Correctness of Joins with Induced Join-Semilattices)**.** *Let $\mathfrak{L}_{Prg}$ be an induced family of join-semilattices for a join operation $\dot\sqcup$ with the induced partial order relation $\preceq$. The (defocusing)* join rule

$$\textit{join by } \dot\sqcup \; \frac{C^* \implies \{U^*\}\,\varphi}{\Gamma_1 \implies \{U_1\}\,\varphi, \Delta_1 \qquad \Gamma_2 \implies \{U_2\}\,\varphi, \Delta_2} \; ((1), (2) \textit{ hold})$$

where
(1) $(U^*, C^*, \varphi) = s_1 \,\dot\sqcup\, s_2$
(2) $s_1 = (U_1, \Gamma_1 \cup \{\neg\psi : \psi \in \Delta_1\}, \varphi)$, $s_2 = (U_2, \Gamma_2 \cup \{\neg\psi : \psi \in \Delta_2\}, \varphi) \in \mathit{SEStates}_{Prg}$
is sound*, i.e. if $C^* \implies \{U^*\}\,\varphi$ is valid, then both $\Gamma_1 \implies \{U_1\}\,\varphi, \Delta_1$ and $\Gamma_2 \implies \{U_2\}\,\varphi, \Delta_2$ are valid.* $\Diamond$

Proof. Under the assumption of the validity of $C^* \implies \{U^*\}\,\varphi$, we have to show the validity of $\Gamma_1 \implies \{U_1\}\,\varphi, \Delta_1$ and $\Gamma_2 \implies \{U_2\}\,\varphi, \Delta_2$, respectively (we consider from now on only the first conclusion of the rule; the proof for the second one is analogous). From

$$s^* := (U^*, C^*, \varphi) = s_1 \,\dot\sqcup\, s_2$$

we obtain $s_1 \preceq s^*$ by Lemma 3.13. Therefore, by property (SEL4) of Definition 3.12, we also obtain $s_1 \lesssim s^*$, i.e. $\mathit{concr}(s_1) \subseteq \mathit{concr}(s^*)$. Let (K_Σ, σ) be an arbitrary model, and let

$$C_1 := \Gamma_1 \cup \{\neg\psi : \psi \in \Delta_1\}.$$

We assume that $(K_\Sigma, \sigma) \models C_1$; otherwise, we are done. We therefore need to show that

$$(K_\Sigma, \sigma) \models \{U_1\}\,\varphi.$$

Due to the validity of $C^* \implies \{U^*\}\,\varphi$, we know that φ holds for all models in

$$\left\{\left(K'_{\Sigma'},\sigma''\right):\sigma''=val_{\left(K'_{\Sigma'},\sigma'\right)}\left(U^*\right)\left(\sigma'\right)\wedge\left(K'_{\Sigma'},\sigma'\right)\models C^*\right\}= \\ \left\{K'_{\Sigma'}:\left(K'_{\Sigma'},\sigma'\right)\models C^*\right\}\times proj_1\left(concr\left(s^*\right)\right)$$

where $\Sigma'\supseteq\Sigma$ results from Σ by adding the new constants introduced in s^*. Since $concr(s_1)\subseteq concr(s^*)$, we particularly know that φ is true in all models contained in the set

$$\left\{K'_{\Sigma'}:\left(K'_{\Sigma'},\sigma'\right)\models C^*\right\}\times proj_1\left(concr\left(s_1\right)\right).$$

Due to (SEL5), C^* is logically equivalent to a formula $C\wedge C_{ax}$, where $C_1\to C$. Therefore,

$$\left\{K'_{\Sigma'}:\left(K'_{\Sigma'},\sigma'\right)\models C^*\right\}=\left\{K'_{\Sigma'}:\left(K'_{\Sigma'},\sigma'\right)\models C\wedge C_{ax}\right\}\supseteq\left\{K'_{\Sigma'}:\left(K'_{\Sigma'},\sigma'\right)\models C_1\wedge C_{ax}\right\},$$

so φ is true in all models of the set

$$\left\{K'_{\Sigma'}:\left(K'_{\Sigma'},\sigma'\right)\models C_1\wedge C_{ax}\right\}\times proj_1\left(concr\left(s_1\right)\right)$$

From Lemma 3.14, we obtain that φ is also modeled by the elements of the seemingly bigger set

$$\left\{K'_{\Sigma}:\left(K'_{\Sigma},\sigma'\right)\models C_1\right\}\times proj_1\left(concr\left(s_1\right)\right)=:\mathcal{M}$$

Obviously, it holds that $\left(K_\Sigma,val_{(K_\Sigma,\sigma)}\left(U_1\right)(\sigma)\right)\in\mathcal{M}$. Therefore,

$$\left(K_\Sigma,val_{(K_\Sigma,\sigma)}\left(U_1\right)(\sigma)\right)\models\varphi$$

which is equivalent to $(K_\Sigma,\sigma)\models\{U_1\}\,\varphi$. □

Remark 3.17 (Considerations on the Join Rule). An integration of the join rule defined in Theorem 3.16 into an existing sequent calculus bears the problem that existing soundness proofs would no longer be valid, considering that they are based on the assumption of rules having only one conclusion. Usually, this only affects the top level part of the soundness proof, since the proofs for individual rules are constructed independently. Still, we circumvent this complication by proposing the following *weakening rule* "weaken by join" with only one conclusion:

$$\text{weaken by join}\ \frac{C^* \implies \{U^*\}\,\varphi}{\Gamma_1 \implies \{U_1\}\,\varphi,\Delta_1}$$

where C^* and U^* are computed including a sequent $\Gamma_2 \implies \{U_2\}\,\varphi,\Delta_2$ as in Theorem 3.16. The "weaken by join" rule is accordingly also sound. When the rule is applied to two leaves $\Gamma_1 \implies \{U_1\}\,\varphi,\Delta_1$ and $\Gamma_2 \implies \{U_2\}\,\varphi,\Delta_2$ of a proof tree, we can append the subtree T following $C^* \implies \{U^*\}\,\varphi$ in the first branch unchanged to the second branch. Figure 3.3 visualizes this scenario. The effect is similar to an employment of the actual defocusing join rule: We only have to *construct* one proof subtree for the such "joined" branches, although we use this tree at two places. In the implementation of our framework, we actually close the second branch to avoid redundancy ($\to$ Section 4.3). ◊

To simplify proofs of property (SEL4) in the subsequent section, we introduce the following simple lemma, which allows us to show property (SEL4) by proving the implication $a\mathbin{\dot\sqcup}b=c\implies a\lesssim c$.

Lemma 3.18. *Let $a,b,c\in SEStates_{Prg}$ be two SE states with the same program counter, and let $\dot\sqcup$ be an idempotent (SEL1) and associative (SEL3) join operation with the corresponding partial order relation $\preceq$. Then, if $a\mathbin{\dot\sqcup}b=c\implies a\lesssim c$, it holds that $a\preceq c\implies a\lesssim c$.* ◊

Proof. We obtain the equivalence $a\mathbin{\dot\sqcup}b=c\iff a\preceq c$ from Lemma 3.13. Therefore, it follows from $a\mathbin{\dot\sqcup}b=c\implies a\lesssim c$ that $a\preceq c\implies a\lesssim c$. □

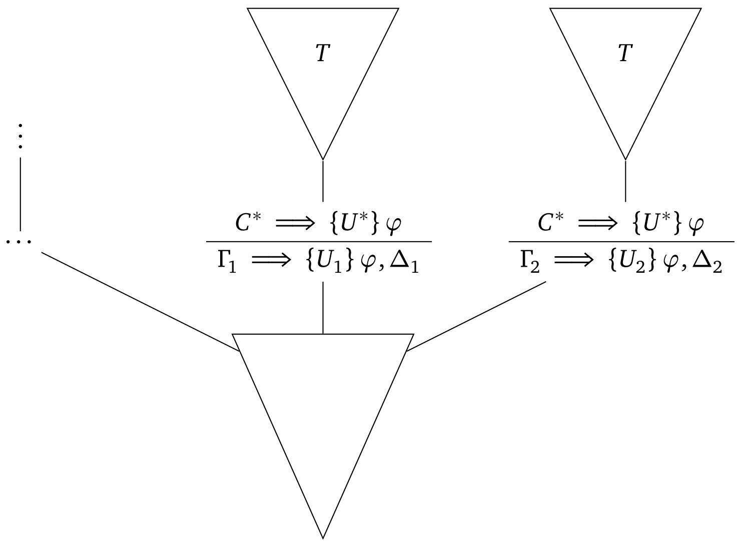

T
T
$C^* \Longrightarrow \{U^*\}\,\varphi$
$\Gamma_1 \Longrightarrow \{U_1\}\,\varphi, \Delta_1$
$C^* \Longrightarrow \{U^*\}\,\varphi$
$\Gamma_2 \Longrightarrow \{U_2\}\,\varphi, \Delta_2$
...

Figure 3.3: Double application of the weakening rule

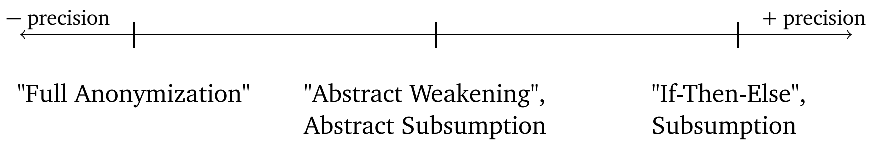

− precision
+ precision
"Full Anonymization"
"Abstract Weakening", Abstract Subsumption
"If-Then-Else", Subsumption

Figure 3.4: Design space for join operators

3.3 Constructing Join Nodes

In this section, we study instantiations of operators $\dot{\sqcup} : \mathit{SEStates}_{\mathit{Prg}} \times \mathit{SEStates}_{\mathit{Prg}} \to \mathit{SEStates}_{\mathit{Prg}}$ inducing families of join-semilattices.

Figure 3.4 illustrates the design space for the operators $\dot{\sqcup}$. The mentioned subsumption techniques are referred to in Section 6.1. Subsequently, we discuss techniques ranging from zero precision (forgetting the symbolic states) to full precision (remembering all values in the symbolic states), also comprising methods in between those border cases.

Remark 3.19. For the join techniques presented subsequently, we assume that the considered program variables are deterministically ordered according to some fixed, e.g. alphabetic, total order. Furthermore, we also assume that the generation of fresh constants for given program variables is deterministic. This may be accomplished by appending a numeric postfix to the name of the given variable and incrementing this number until the such generated symbol does not occur in the SE states that are joined. This property ensures that the names of fresh constants do not depend on the order of the input states, and thus simplifies some of our proofs. $\Diamond$

3.3.1 Full Anonymization

The most radical way of constructing a join node inducing a family of join-semilattices of SE states is to completely “forget” the symbolic state of the parent nodes that are to be joined for variables that are assigned different values. In this technique, the value of a program location is set to a fixed, but unknown value. Technically, this is realized by the introduction of a fresh constant (of suitable type). The technique marks the zero-precision border of the design space for join operations.

Technique 3.20 (Full Anonymization Join Method). *Given (U_1, C_1, φ) and $(U_2, C_2, \varphi) \in \mathit{SEStates}_{\mathit{Prg}}$ for any program Prg with program variables $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \in \mathrm{PV}$, the join operation $\dot{\sqcup}_{\mathit{fresh}} : \mathit{SEStates}_{\mathit{Prg}} \times \mathit{SEStates}_{\mathit{Prg}} \to \mathit{SEStates}_{\mathit{Prg}}$ is defined by*

$$(U_1, C_1, \varphi)\,\dot{\sqcup}_{\mathit{fresh}}\,(U_2, C_2, \varphi) := \big((U_1, C_1) \oplus_{\mathit{fresh}} (U_2, C_2), (U_1, C_1) \heartsuit_{\mathit{fresh}} (U_2, C_2), \varphi\big)$$

where $(U_1, C_1) \circledvee_{\mathit{fresh}} (U_2, C_2) := C_1 \lor C_2$ *and* $(U_1, C_1) \oplus_{\mathit{fresh}} (U_2, C_2) := \{\mathtt{x}_1 := t_1 \parallel \mathtt{x}_2 := t_2 \parallel \cdots \parallel \mathtt{x}_n := t_n\}$ *with*

$$t_i := \begin{cases} \{U_1\}\,\mathtt{x}_i & \textit{if } \vdash (C_1 \to \{U_1\}\,P(\mathtt{x}_i)) \leftrightarrow (C_2 \to \{U_2\}\,P(\mathtt{x}_i)) \quad (\star) \\ c_i & \textit{otherwise} \end{cases}$$

for fresh constant symbols $c_i \in \mathrm{Func}$ *of suitable types that are not contained in* $U_1, U_2, C_1, C_2, \varphi$*, and a fresh, uninterpreted predicate symbol* P. ◊

The following proposition assures that joins using Technique 3.20 are sound in the KeY calculus by Theorem 3.16.

Proposition 3.21. *Technique 3.20 induces a family of join-semilattices of SE states, i.e. the operation* $\dot\sqcup_{\mathit{fresh}}$ *and its associated partial order relation* $\preceq$ *satisfy the axioms (SEL1) to (SEL5) of Definition 3.12.* ◊

Proof. (SEL1) Let $a = (U, C, \varphi) \in \mathit{SEStates}_{\mathit{Prg}}$. Then, $a \dot\sqcup_{\mathit{fresh}} a = (U^*, C \lor C, \varphi)$. In U^*, each program variable $\mathtt{x}_i$ is set to the term to which it evaluates in U, since $(C \to \{U\}\,P(\mathtt{x}_i)) \leftrightarrow (C \to \{U\}\,P(\mathtt{x}_i))$ is always provable. Because of this and idempotency of disjunction, we have $a \dot\sqcup_{\mathit{fresh}} a \overset{concr}{=} a$ in both directions of $\overset{concr}{=}$.

(SEL2) Let $a = (U_1, C_1, \varphi),\ b = (U_2, C_2, \varphi) \in \mathit{SEStates}_{\mathit{Prg}}$. We prove

$$a \dot\sqcup_{\mathit{fresh}} b =: \left(U_1^*, C_1^*, \varphi\right) \overset{concr}{=} \left(U_2^*, C_2^*, \varphi\right) := b \dot\sqcup_{\mathit{fresh}} a$$

by showing

$$\sigma \in \mathit{concr}\left(a \dot\sqcup_{\mathit{fresh}} b\right) \iff \sigma \in \mathit{concr}\left(b \dot\sqcup_{\mathit{fresh}} a\right).$$

It holds that

$$\begin{aligned} & \sigma \in \mathit{concr}\left(a \dot\sqcup_{\mathit{fresh}} b\right) \\ \iff & \text{there is } \left(K_\Sigma, \sigma'\right), \left(K_\Sigma, \sigma'\right) \models C_1 \lor C_2 \text{ and } \sigma = \mathit{val}_{(K_\Sigma, \sigma')}\left(U_1^*\right)\left(\sigma'\right) \\ \iff & \left(K_\Sigma, \sigma'\right) \models C_2 \lor C_1 \\ \iff & \sigma'' := \mathit{val}_{(K_\Sigma, \sigma')}\left(U_2^*\right)\left(s'\right) \in \mathit{concr}\left(b \dot\sqcup_{\mathit{fresh}} a\right). \end{aligned}$$

Therefore it suffices to show that for every $\mathtt{x}_i$:

$$\left(K_\Sigma, \sigma'\right) \models \left\{U_1^*\right\} P(\mathtt{x}_i) \leftrightarrow \left\{U_2^*\right\} P(\mathtt{x}_i),$$

for a suitable new predicate P, i.e. that the right sides for $\mathtt{x}_i$ are equivalent in (K_Σ, σ'). This is trivially the case for equal new constants c_i as right sides. Note that a right side in U_1^* is a fresh constant iff the corresponding right side in U_2^* is a fresh constant, since the bi-implication in $(\star)$ is commutative; furthermore, new constants are assumed to be deterministically generated ($\to$ Remark 3.19). In the other case, that is if $(\star)$ is provable, the equivalence obviously also holds.

(SEL3) Let $a = (U_1, C_1, \varphi),\ b = (U_2, C_2, \varphi),\ c = (U_3, C_3, \varphi) \in \mathit{SEStates}_{\mathit{Prg}}$. We have to show that

$$\left(a \dot\sqcup_{\mathit{fresh}} b\right) \dot\sqcup_{\mathit{fresh}} c \overset{concr}{=} a \dot\sqcup_{\mathit{fresh}} \left(b \dot\sqcup_{\mathit{fresh}} c\right),$$

that is

$$\sigma \in \mathit{concr}\left(\left(a \dot\sqcup_{\mathit{fresh}} b\right) \dot\sqcup_{\mathit{fresh}} c\right) \iff \sigma \in \mathit{concr}\left(a \dot\sqcup_{\mathit{fresh}} \left(b \dot\sqcup_{\mathit{fresh}} c\right)\right).$$

Let

$$\begin{aligned} a \dot\sqcup_{\mathit{fresh}} b &:= \left(U_1^*, C_1 \lor C_2, \varphi\right) \\ b \dot\sqcup_{\mathit{fresh}} c &:= \left(U_2^*, C_2 \lor C_3, \varphi\right) \\ \left(a \dot\sqcup_{\mathit{fresh}} b\right) \dot\sqcup_{\mathit{fresh}} c &:= \left(U_1^{**}, (C_1 \lor C_2) \lor C_3, \varphi\right) \\ a \dot\sqcup_{\mathit{fresh}} \left(b \dot\sqcup_{\mathit{fresh}} c\right) &:= \left(U_2^{**}, C_1 \lor (C_2 \lor C_3), \varphi\right) \end{aligned}$$

and $\sigma \in concr\big((a \mathbin{\dot\sqcup_{fresh}} b) \mathbin{\dot\sqcup_{fresh}} c\big)$. Therefore, there is a model (K_Σ, σ') such that $(K_\Sigma, \sigma') \models (C_1 \vee C_2) \vee C_3$ and $\sigma = val_{(K_\Sigma,\sigma')}\big(U_1^{**}\big)(\sigma')$. Obviously, it holds that $(K_\Sigma, \sigma') \models C_1 \vee (C_2 \vee C_3)$ due to associativity of disjunction, thus it holds that

$$val_{(K_\Sigma,\sigma')}\big(U_2^{**}\big)\big(\sigma'\big) \in concr\big(a \mathbin{\dot\sqcup_{fresh}} \big(b \mathbin{\dot\sqcup_{fresh}} c\big)\big).$$

It remains to show that $val_{(K_\Sigma,\sigma')}\big(U_2^{**}\big)(\sigma') = val_{(K_\Sigma,\sigma')}\big(U_1^{**}\big)(\sigma')$. Let x be in $\mathrm{x}_1, \mathrm{x}_2, \ldots, \mathrm{x}_n$. Consider two cases: (i) The right side for x in U_1^{**} is $\big\{U_1^*\big\}\,\mathrm{x}$. Then, the terms $\{U_1\}\,\mathrm{x}$, $\{U_2\}\,\mathrm{x}$, $\{U_3\}\,\mathrm{x}$ and also $\big\{U_1^*\big\}\,\mathrm{x}$ must be equivalent in every model due to the definition of t_i. The right side in U_1^{**} is therefore $\{U_1\}\,\mathrm{x}$ which we know is equivalent to $\big\{U_1^*\big\}\,\mathrm{x}$. (ii) The right side for x in U_1^{**} is a Skolem constant. Then, by $(\star)$, $\big\{U_1^*\big\}\,\mathrm{x}$ and $\{U_3\}\,\mathrm{x}$ are not logically equivalent terms. Thus, either $\{U_1\}\,\mathrm{x}$ and $\{U_2\}\,\mathrm{x}$ are not equivalent, or both are not equivalent to $\{U_3\}\,\mathrm{x}$. In both of these cases, $\{U_1\}\,\mathrm{x}$ and $\big\{U_2^*\big\}\,\mathrm{x}$ cannot be equivalent, and therefore the right side for x in U_2^{**} is also the same constant.

(SEL4) Since we already proved idempotency and associativity, it suffices to show

$$a \mathbin{\dot\sqcup_{fresh}} b = c \implies a \lesssim c$$

for any $a = (U_1, C_1, \varphi)$, $b = (U_2, C_2, \varphi)$, $c = (U^*, C_1 \vee C_2, \varphi) \in SEStates_{Prg}$ by Lemma 3.18. Assuming $a \mathbin{\dot\sqcup_{fresh}} b = c$, we thus have to show that $concr(a) \subseteq concr(c)$. Let $\sigma \in concr(a)$, i.e. there is a model $(K_\Sigma, \sigma') \models C_1$ such that $\sigma = val_{(K_\Sigma,\sigma')}(U_1)(\sigma')$. Let K'_Σ be an expansion of K_Σ by the new constants c_i such that $I_{K'_\Sigma}(c_i) = \sigma(\mathrm{x}_i)$. Since C_1 is in the old language, $\big(K'_\Sigma, \sigma'\big) \models C_1$ and thus $\big(K'_\Sigma, \sigma'\big) \models C_1 \vee C_2$. Therefore, we have

$$val_{\left(K'_\Sigma,\sigma'\right)}(U^*)\big(\sigma'\big) \in concr(c).$$

It suffices to show that, for all $i = 1, 2, \ldots, n$:

$$val_{\left(K'_\Sigma,\sigma'\right)}(U^*)\big(\sigma'\big)(\mathrm{x}_i) = val_{(K_\Sigma,\sigma')}(U_1)\big(\sigma'\big)(\mathrm{x}_i).$$

We distinguish the following two cases:

Case "$(\star)$ is provable". In this case,

$$\begin{aligned}
val_{\left(K'_\Sigma,\sigma'\right)}(U^*)\big(\sigma'\big)(\mathrm{x}_i) &= val_{\left(K'_\Sigma,\sigma'\right)}(\mathrm{x}_i := \{U_1\}\,\mathrm{x}_i)\big(\sigma'\big)(\mathrm{x}_i) \\
&= \left(\mathrm{x} \mapsto \begin{cases} \sigma'(\mathrm{x}) & \text{if } \mathrm{x} \neq \mathrm{x}_i \\ val_{\left(K'_\Sigma,\sigma'\right)}(\{U_1\}\,\mathrm{x}_i) & \text{otherwise} \end{cases}\right)(\mathrm{x}_i) \\
&= val_{\left(K'_\Sigma,\sigma'\right)}(\{U_1\}\,\mathrm{x}_i) \\
&= val_{\left(K'_\Sigma,\sigma'\right)}(U_1)\big(\sigma'\big)(\mathrm{x}_i) \\
&= val_{(K_\Sigma,\sigma')}(U_1)\big(\sigma'\big)(\mathrm{x}_i).
\end{aligned}$$

where the last equation is true since U_1 is in the language without the constants.

Case "otherwise". Then,

$$\begin{aligned}
val_{\left(K'_\Sigma,\sigma'\right)}(U^*)\big(\sigma'\big)(\mathrm{x}_i) &= val_{\left(K'_\Sigma,\sigma'\right)}(\mathrm{x}_i := c_i)\big(\sigma'\big)(\mathrm{x}_i) \\
&= \left(\mathrm{x} \mapsto \begin{cases} \sigma'(\mathrm{x}) & \text{if } \mathrm{x} \neq \mathrm{x}_i \\ val_{\left(K'_\Sigma,\sigma'\right)}(c_i) & \text{otherwise} \end{cases}\right)(\mathrm{x}_i) \\
&= val_{\left(K'_\Sigma,\sigma'\right)}(c_i) = I_{K'_\Sigma}(c_i) \overset{\text{def.}}{=} \sigma(\mathrm{x}_i) \\
&= val_{(K_\Sigma,\sigma')}(U_1)\big(\sigma'\big)(\mathrm{x}_i).
\end{aligned}$$

(SEL5) We have to show that $C^* \leftrightarrow (C \wedge C_{ax})$, where $\bigwedge C_1 \rightarrow C$ and $\exists \overline{v}; (C_{ax}[\overline{v}/\overline{c}])$, for $\overline{c} = c_1, c_2, \ldots, c_n$ being the constants introduced in c and $\overline{v}$ being a tuple of suitable fresh logical variables, is provable. By definition, $C^* = \big(\bigwedge C_1 \vee \bigwedge C_2\big)$. We choose $C := \bigwedge C_1 \vee \bigwedge C_2$ and $C_{ax} := true$. Obviously, $\bigwedge C_1 \rightarrow \big(\bigwedge C_1 \vee \bigwedge C_2\big)$, and $true$ is trivially provable.

□

3.3.2 If-Then-Else

The "If-Then-Else" technique is a "classic" of state joining for symbolic execution (see, e.g., [HSS09; Kuz+12; Sen+14]) that retains the full precision of the analysis. We realize the technique by employing the if-the-else construct of Java DL (→ Definition 2.6). An alternative way is the introduction of fresh Skolem constants for the symbolic states that are constrained by implications in the path condition. We discuss this approach in the subsequent subsection.

Technique 3.22 (If-Then-Else Join Method). *Given (U_1, C_1, φ) and $(U_2, C_2, \varphi) \in SEStates_{Prg}$ for any program Prg with program variables* $\mathtt{x}_1, \mathtt{x}_2, \ldots, \mathtt{x}_n \in \mathrm{PV}$, *the join operation $\dot{\sqcup}_{ite} : SEStates_{Prg} \times SEStates_{Prg} \to SEStates_{Prg}$ is defined by*

$$(U_1, C_1, \varphi) \dot{\sqcup}_{ite} (U_2, C_2, \varphi) := ((U_1, C_1) \oplus_{ite} (U_2, C_2), (U_1, C_1) \circledvee_{ite} (U_2, C_2), \varphi)$$

where $(U_1, C_1) \circledvee_{ite} (U_2, C_2) := \left(\bigwedge C_1\right) \vee \left(\bigwedge C_2\right)$ *and* $(U_1, C_1) \oplus_{ite} (U_2, C_2) := \{\mathtt{x}_1 := t_1 \parallel \cdots \parallel \mathtt{x}_n := t_n\}$ *with*

$$t_i := \begin{cases} \{U_1\}\,\mathtt{x}_i & \textit{if } \vdash (C_1 \to \{U_1\}\,P(\mathtt{x}_i)) \leftrightarrow (C_2 \to \{U_2\}\,P(\mathtt{x}_i)) \quad (\star) \\ \textit{if}\left(\bigwedge C_1\right) \textit{then}\,(\{U_1\}\,\mathtt{x}_i)\,\textit{else}\,(\{U_2\}\,\mathtt{x}_i) & \textit{otherwise} \end{cases}$$

for a fresh, uninterpreted predicate symbol P. ◊

Remark 3.23 (Incompatible Path Conditions). For our join techniques, and in particular for the techniques 3.22 and 3.25, we assume that two path conditions C_1 and C_2 of different branches in a tree are generally incompatible. This assumption is reasonable since splits in symbolic execution should only occur when the execution depends on the concrete value of a variable; in this case, a case distinction takes place. If C_1 and C_2 are the path conditions for those case distinction branches, it holds that $\vdash \bigwedge C_1 \to \neg \bigwedge C_2$, as well as the contraposition $\vdash \bigwedge C_2 \to \neg \bigwedge C_1$. The assumption *could* be problematic when If-Then-Else joins are applied to an SET where other branches are joined with rules violating these constraints. Since our proposed join rules either use the canonical disjunction of path conditions for the path conditions of join nodes, or a strengthening of that (in particular, they respect the property (SEL5)), we are confident that this issue can be excluded. ◊

Proposition 3.24. *Technique 3.22 induces a family of join-semilattices of SE states, i.e. the operation $\dot{\sqcup}_{ite}$ and its associated partial order relation $\preceq$ satisfy the axioms (SEL1) to (SEL5) of Definition 3.12.* ◊

Proof. (SEL1) The proof for (SEL1) is similar to the corresponding case in the proof of Proposition 3.21.

(SEL2) Under the assumption $\vdash \bigwedge C_1 \to \neg \bigwedge C_2$, the terms $\textit{if}\left(\bigwedge C_1\right)\textit{then}\,(\{U_1\}\,\mathtt{x})\,\textit{else}\,(\{U_2\}\,\mathtt{x})$ and $\textit{if}\left(\bigwedge C_2\right)\textit{then}\,(\{U_2\}\,\mathtt{x})\,\textit{else}\,(\{U_1\}\,\mathtt{x})$ are logically equivalent. Based on this observation, the proof of (SEL2) is similar to the corresponding case in the proof of Proposition 3.21.

(SEL3) Under the assumption $\vdash \bigwedge C_1 \to \neg \bigwedge C_2$, the proof of (SEL3) is similar to the corresponding case in the proof of Proposition 3.21.

(SEL4) Since we already proved idempotency and associativity, it suffices to show

$$a \,\dot{\sqcup}_{ite}\, b = c \implies a \lesssim c$$

for any $a = (U_1, C_1, \varphi)$, $b = (U_2, C_2, \varphi)$, $c = (U^*, C_1 \vee C_2, \varphi) \in SEStates_{Prg}$ by Lemma 3.18. Assuming $a \,\dot{\sqcup}_{ite}\, b = c$, we thus have to show that $concr(a) \subseteq concr(c)$. Let $\sigma \in concr(a)$, i.e. there is a model $(K_\Sigma, \sigma') \models C_1$ such that $\sigma = val_{(K_\Sigma,\sigma')}(U_1)(\sigma')$. Since furthermore, $(K_\Sigma, \sigma') \models C_1 \vee C_2$, it holds that

$$val_{(K_\Sigma,\sigma')}(U^*)\left(\sigma'\right) \in concr(c)\,.$$

It suffices to show that, for all $i = 1, 2, \ldots, n$:

$$val_{(K_\Sigma,\sigma')}(U^*)\left(\sigma'\right)(\mathtt{x}_i) = val_{(K_\Sigma,\sigma')}(U_1)\left(\sigma'\right)(\mathtt{x}_i)\,.$$

We distinguish the following two cases:

Case "($\star$) is provable". In this case,

$$\begin{aligned}
val_{(K_\Sigma,\sigma')}(U^*)\big(\sigma'\big)(\mathrm{x}_i) &= val_{(K_\Sigma,\sigma')}(\mathrm{x}_i := \{U_1\}\,\mathrm{x}_i)\big(\sigma'\big)(\mathrm{x}_i)\\
&= \left(\mathrm{x} \mapsto \begin{cases}\sigma'(\mathrm{x}) & \text{if } \mathrm{x} \neq \mathrm{x}_i\\ val_{(K_\Sigma,\sigma')}(\{U_1\}\,\mathrm{x}_i) & \text{otherwise}\end{cases}\right)(\mathrm{x}_i)\\
&= val_{(K_\Sigma,\sigma')}(\{U_1\}\,\mathrm{x}_i)\\
&= val_{(K_\Sigma,\sigma')}(U_1)\big(\sigma'\big)(\mathrm{x}_i)
\end{aligned}$$

Case "otherwise". Then,

$$\begin{aligned}
&val_{(K_\Sigma,\sigma')}(U^*)\big(\sigma'\big)(\mathrm{x}_i) =\\
&val_{(K_\Sigma,\sigma')}\big(\mathrm{x}_i := \mathit{if}\,\big(\textstyle\bigwedge C_1\big)\,\mathit{then}\,(\{U_1\}\,\mathrm{x}_i)\,\mathit{else}\,(\{U_2\}\,\mathrm{x}_i)\big)\big(\sigma'\big)(\mathrm{x}_i) =\\
&\left(\mathrm{x} \mapsto \begin{cases}\sigma'(\mathrm{x}) & \text{if } \mathrm{x} \neq \mathrm{x}_i\\ val_{(K_\Sigma,\sigma')}\big(\mathit{if}\,\big(\bigwedge C_1\big)\,\mathit{then}\,(\{U_1\}\,\mathrm{x}_i)\,\mathit{else}\,(\{U_2\}\,\mathrm{x}_i)\big) & \text{otherwise}\end{cases}\right)(\mathrm{x}_i) =\\
&\left(\mathrm{x} \mapsto \begin{cases}\sigma'(\mathrm{x}) & \text{if } \mathrm{x} \neq \mathrm{x}_i\\ val_{(K_\Sigma,\sigma')}(\{U_1\}\,\mathrm{x}_i) & \mathrm{x} = \mathrm{x}_i \land val_{(K_\Sigma,\sigma')}\big(\bigwedge C_1\big) = \mathit{tt}\\ val_{(K_\Sigma,\sigma')}(\{U_2\}\,\mathrm{x}_i) & \text{otherwise}\end{cases}\right)(\mathrm{x}_i) =\\
&val_{(K_\Sigma,\sigma')}(\{U_1\}\,\mathrm{x}_i) =\\
&val_{(K_\Sigma,\sigma')}(U_1)\big(\sigma'\big)(\mathrm{x}_i).
\end{aligned}$$

(SEL5) We have to show that $C^* \leftrightarrow (C \land C_{ax})$, where $\bigwedge C_1 \to C$ and $\exists \overline{v}; (C_{ax}[\overline{v}/\overline{c}])$, is provable, which is equivalent to C_{ax} being provable since $\dot\sqcup_{ite}$ does not introduce new constants. By definition, $C^* = \big(\bigwedge C_1 \lor \bigwedge C_2\big)$. We choose $C := \bigwedge C_1 \lor \bigwedge C_2$ and $C_{ax} := \mathit{true}$. Obviously, $\bigwedge C_1 \to \big(\bigwedge C_1 \lor \bigwedge C_2\big)$, and *true* is trivially provable.

□

3.3.3 If-Then-Else by additional Path Condition Constraints

An alternative to the If-Then-Else approach discussed in Section 3.3.2 is the realization of constraints of program variables by the introduction of fresh Skolem constants along with additional constraints in the path conditions of join states. While being logically equivalent (under the assumption $\vdash \bigwedge C_1 \to \neg \bigwedge C_2$ of Remark 3.23), this technique has a different runtime behavior when implemented in KeY ($\to$ Section 5.1). Since the path condition of a state is usually contained in the antecedent of the corresponding sequent in the SET, we also refer to this technique as "If-Then-Else-Antecedent".

Technique 3.25 (If-Then-Else with PC Constraints Join Method). *Given (U_1, C_1, φ) and $(U_2, C_2, \varphi) \in SEStates_{Prg}$ for any program Prg with program variables $\mathrm{x}_1, \mathrm{x}_2, \ldots, \mathrm{x}_n \in$ PV, the join operation $\dot\sqcup_{ite2}$: $SEStates_{Prg} \times SEStates_{Prg} \to SEStates_{Prg}$ is defined by*

$$(U_1, C_1, \varphi)\,\dot\sqcup_{ite2}\,(U_2, C_2, \varphi) := ((U_1, C_1) \oplus_{ite2} (U_2, C_2), (U_1, C_1) \circledvee_{ite2} (U_2, C_2), \varphi)$$

where $(U_1, C_1) \oplus_{ite2} (U_2, C_2) := \{\mathrm{x}_1 := t_1 \parallel \mathrm{x}_2 := t_2 \parallel \cdots \parallel \mathrm{x}_n := t_n\}$ with

$$t_i := \begin{cases}\{U_1\}\,\mathrm{x}_i & \text{if } \vdash (C_1 \to \{U_1\}\,P(\mathrm{x}_i)) \leftrightarrow (C_2 \to \{U_2\}\,P(\mathrm{x}_i))\\ c_i & \text{otherwise}\end{cases}$$

and $(U_1, C_1) \circledvee_{ite2} (U_2, C_2) := \big(\bigwedge C_1 \lor \bigwedge C_2\big) \land \bigwedge C_i^{ite}$ where

$$C_i^{ite} := \begin{cases}\mathit{true} & \text{if } \vdash (C_1 \to \{U_1\}\,P(\mathrm{x}_i)) \leftrightarrow (C_2 \to \{U_2\}\,P(\mathrm{x}_i))\\ \big(\bigwedge C_1 \to c_i \doteq \{U_1\}\,\mathrm{x}_i\big) \land & \text{otherwise}\\ \qquad \big(\bigwedge C_2 \to c_i \doteq \{U_2\}\,\mathrm{x}_i\big) & \end{cases}$$

for a fresh, uninterpreted predicate symbol P and fresh Skolem constants $c_1, c_2, \ldots, c_n$ of appropriate types that are not contained in $U_1, U_2, C_1, C_2, \varphi$. For the "otherwise" case, we assume $\vdash \bigwedge C_1 \to \neg \bigwedge C_2$ ($\to$ Remark 3.23). ◊

Proposition 3.26. *Technique 3.25 induces a family of join-semilattices of SE states, i.e. the operation $\dot{\sqcup}_{ite2}$ and its associated partial order relation $\preceq$ satisfy the axioms (SEL1) to (SEL5) of Definition 3.12.* $\Diamond$

Proof. We omit the proofs of (SEL1) to (SEL3) which are basically obvious when following the arguments in the corresponding proofs of the techniques discussed above.

For (SEL4), it suffices to show

$$a \mathbin{\dot{\sqcup}_{ite2}} b = c \implies a \lesssim c$$

for any $a = (U_1, C_1, \varphi)$, $b = (U_2, C_2, \varphi)$, $c = (U^*, C_1 \vee C_2, \varphi) \in SEStates_{Prg}$ by Lemma 3.18. Assuming $a \mathbin{\dot{\sqcup}_{ite2}} b = c$, we thus have to show that $concr(a) \subseteq concr(c)$. We again assume that $\vdash \bigwedge C_1 \to \neg \bigwedge C_2$. Let $\sigma \in concr(a)$, i.e. there is a model $(K_\Sigma, \sigma') \models C_1$ such that $\sigma = val_{(K_\Sigma,\sigma')}(U_1)(\sigma')$. Let K'_Σ be an expansion of K_Σ by the new constants c_i such that $I_{K'_\Sigma}(c_i) = \sigma(\mathrm{x}_i)$. Since C_1 is in the old language, $\left(K'_\Sigma, \sigma'\right) \models C_1$ (which is a direct consequence of Herbrand's theorem), $\left(K'_\Sigma, \sigma'\right) \models C_1 \vee C_2$ and $\left(K'_\Sigma, \sigma'\right) \models \bigwedge C_i^{ite}$. Therefore, we have

$$val_{\left(K'_\Sigma,\sigma'\right)}(U^*)\left(\sigma'\right) \in concr(c).$$

It suffices to show that, for all $i = 1, 2, \ldots, n$:

$$val_{\left(K'_\Sigma,\sigma'\right)}(U^*)\left(\sigma'\right)(\mathrm{x}_i) = val_{(K_\Sigma,\sigma')}(U_1)\left(\sigma'\right)(\mathrm{x}_i).$$

We distinguish the following two cases:

Case "$(\star)$ is provable". In this case,

$$\begin{aligned}
val_{\left(K'_\Sigma,\sigma'\right)}(U^*)\left(\sigma'\right)(\mathrm{x}_i) &= val_{\left(K'_\Sigma,\sigma'\right)}(\mathrm{x}_i := \{U_1\}\,\mathrm{x}_i)\left(\sigma'\right)(\mathrm{x}_i) \\
&= \left(\mathrm{x} \mapsto \begin{cases} \sigma'(\mathrm{x}) & \text{if } \mathrm{x} \neq \mathrm{x}_i \\ val_{\left(K'_\Sigma,\sigma'\right)}(\{U_1\}\,\mathrm{x}_i) & \text{otherwise} \end{cases}\right)(\mathrm{x}_i) \\
&= val_{\left(K'_\Sigma,\sigma'\right)}(\{U_1\}\,\mathrm{x}_i) \\
&= val_{\left(K'_\Sigma,\sigma'\right)}(U_1)\left(\sigma'\right)(\mathrm{x}_i) \\
&\overset{(\dagger)}{=} val_{(K_\Sigma,\sigma')}(U_1)\left(\sigma'\right)(\mathrm{x}_i).
\end{aligned}$$

where $(\dagger)$ is true since U_1 is in the language without the constants.

Case "otherwise". Then,

$$\begin{aligned}
val_{\left(K'_\Sigma,\sigma'\right)}(U^*)\left(\sigma'\right)(\mathrm{x}_i) &= val_{\left(K'_\Sigma,\sigma'\right)}(\mathrm{x}_i := c_i)\left(\sigma'\right)(\mathrm{x}_i) \\
&= \left(\mathrm{x} \mapsto \begin{cases} \sigma'(\mathrm{x}) & \text{if } \mathrm{x} \neq \mathrm{x}_i \\ val_{\left(K'_\Sigma,\sigma'\right)}(c_i) & \text{otherwise} \end{cases}\right)(\mathrm{x}_i) \\
&= val_{\left(K'_\Sigma,\sigma'\right)}(c_i) = I_{K'_\Sigma}(c_i) \overset{\text{def.}}{=} \sigma(\mathrm{x}_i) \\
&= val_{(K_\Sigma,\sigma')}(U_1)\left(\sigma'\right)(\mathrm{x}_i).
\end{aligned}$$

For (SEL5), we have to show that $C^* \leftrightarrow (C \wedge C_{ax})$, where $\bigwedge C_1 \to C$ and $\exists \overline{v}; (C_{ax}[\overline{v}/\overline{c}])$, for $\overline{c} = c_1, c_2, \ldots, c_n$ and $\overline{v}$ is a tuple of suitable fresh logical variables, is provable. By definition, $C^* = \left(\bigwedge C_1 \vee \bigwedge C_2\right) \wedge \bigwedge C_i^{ite}$. We choose $C := \bigwedge C_1 \vee \bigwedge C_2$ and $C_{ax} := \bigwedge C_i^{ite}$. Obviously, $\bigwedge C_1 \to \left(\bigwedge C_1 \vee \bigwedge C_2\right)$. $\exists \overline{v}; (C_{ax}[\overline{v}/\overline{c}])$ equals

$$\exists \overline{v}; \left(\bigwedge \left\{\left(\bigwedge C_1 \to v_i \doteq \{U_1\}\,\mathrm{x}_i\right) \wedge \left(\bigwedge C_2 \to v_i \doteq \{U_2\}\,\mathrm{x}_i\right)\right\}\right).$$

This formula is easily provable for the instantiation

$$v_i := \begin{cases} \{U_1\}\,\mathrm{x}_i & \text{if } C_1 \text{ is provable} \\ \{U_2\}\,\mathrm{x}_i & \text{otherwise} \end{cases}$$

$\square$

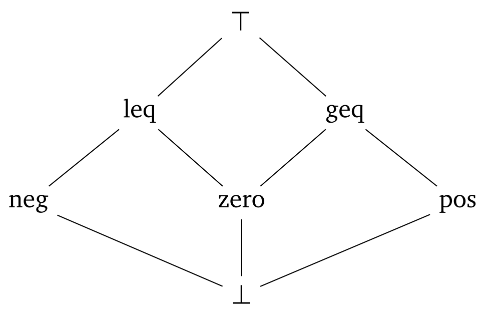


Figure 3.5: Abstract domain for sign analysis

3.3.4 Abstract Weakening

Abstract Interpretation [CC77] is a static analysis technique that allows for the construction of fully automatic proof methods [Cou+05]. Our General Lattice Framework, along with the technique proposed subsequently, at least partly closes the gap between symbolic execution and abstract interpretation by facilitating joins based on abstract domain lattices. We first define the notion of abstract domain elements.

Definition 3.27 (Abstract Domain Element)**.** An *Abstract Domain Element* is a function *defAx* : $\mathrm{Terms}_\Sigma \to \mathrm{Form}_\Sigma$ mapping Java DL terms of appropriate types to *closed* formulae. ◊

Intuitively, an abstract domain element models an infinite set of *defining axioms* for Java DL terms. If an axiom is true for a given term, then this term is described by the corresponding abstract domain element. This rather technical, syntactical definition is beneficial for the application in branch joining. Note that we restrict abstract elements / domains to those that can be characterized in Java DL.

Definition 3.28 (Abstract Domain Lattice)**.** An *Abstract Domain Lattice* is a join-semilattice $\mathcal{A}_T = (A_T, \sqcup)$ with the induced partial order relation $\sqsubseteq$ for a countable set A_T of abstract domain elements accepting terms of some fixed type $T \in \mathcal{T}$ as arguments. We impose the following requirements on A_T and $\sqsubseteq$:

(1) A_T includes two elements $\bot(t) = \mathit{false}$ and $\top(t) = \mathit{true}$
(2) For $a, b \in A_T$ with $a \sqsubseteq b$, it holds that $\vdash a(t) \to b(t)$ for any term t of type T.
(3) For all $a \in A_T$ except for $\bot$, it holds that $\vdash \exists v; a(v)$. ◊

Example 3.29. A well-known example for an abstract domain lattice also used in [BHW09] is the lattice $\mathcal{A}^{sgn}_{\mathtt{int}} = \left(A^{sgn}_{\mathtt{int}}, \sqcup\right)$ for sign analysis visualized in the Hasse diagram of Figure 3.5. $A^{sgn}_{\mathtt{int}}$ is defined as $A^{sgn}_{\mathtt{int}} = \{\bot, \mathrm{neg}, \mathrm{zero}, \mathrm{pos}, \mathrm{leq}, \mathrm{geq}, \top\}$ where $\bot(t) := \mathit{false}$, $\mathrm{neg}(t) := t < 0$, $\mathrm{zero}(t) := t \doteq 0$, $\mathrm{pos}(t) := t > 0$, $\mathrm{leq}(t) := t \leq 0$, $\mathrm{geq}(t) := t \geq 0$ and $\top(t) := \mathit{true}$. The join operator $\sqcup$ and its induced partial order relation $\sqsubseteq$ are defined like depicted in the Hasse diagram. Intuitively, geq, for instance, represents the positive integers including zero, while pos represents the positive integers without zero; $\top$ represents all integers, whilst $\bot$ does not represent any integer. ◊

For the definition of the following technique, we assume that there are only program variables of the same type $T \in \mathcal{T}$. However, the technique is easily generalized to multiple types by demanding the existence of one abstract domain lattice per type; moreover, it is possible to define "hybrid" approaches using, for instance, the "If-Then-Else" join technique whenever there is no abstract domain lattice for a given type.

Technique 3.30 (Abstract Weakening Join Method)**.** *Let $\mathcal{A}_T = (A_T, \sqcup)$ be an abstract domain lattice. Given (U_1, C_1, φ) and $(U_2, C_2, \varphi) \in \mathit{SEStates}_{Prg}$ for any program Prg with program variables* $\mathrm{x}_1, \mathrm{x}_2, \ldots, \mathrm{x}_n \in$ PV *of type T, the join operation $\dot\sqcup_{abstr} : \mathit{SEStates}_{Prg} \times \mathit{SEStates}_{Prg} \to \mathit{SEStates}_{Prg}$ is defined by*

$$(U_1, C_1, \varphi) \dot\sqcup_{abstr} (U_2, C_2, \varphi) := ((U_1, C_1) \oplus_{abstr} (U_2, C_2), (U_1, C_1) \odot_{abstr} (U_2, C_2), \varphi)$$

where $(U_1, C_1) \oplus_{abstr} (U_2, C_2) := \{\mathrm{x}_1 := t_1 \parallel \mathrm{x}_2 := t_2 \parallel \cdots \parallel \mathrm{x}_n := t_n\}$ with, for a fresh, uninterpreted predicate symbol P and fresh Skolem constants $c_1, c_2, \ldots, c_n$ of appropriate types that are not contained in $U_1, U_2, C_1, C_2, \varphi$,

$$t_i := \begin{cases} \{U_1\}\,\mathrm{x}_i & \text{if } \vdash (C_1 \to \{U_1\}\,P(\mathrm{x}_i)) \leftrightarrow (C_2 \to \{U_2\}\,P(\mathrm{x}_i)) \\ c_i & \text{otherwise} \end{cases}$$

and $(U_1, C_1) \heartsuit_{abstr} (U_2, C_2) := \left(\bigwedge C_1 \vee \bigwedge C_2\right) \wedge \bigwedge C_i^{abs}$ *where*

$$C_i^{abs} := \begin{cases} true & \text{if } \vdash (C_1 \to \{U_1\} P(\mathrm{x}_i)) \leftrightarrow (C_2 \to \{U_2\} P(\mathrm{x}_i)) \\ \left(defAx_1 \sqcup defAx_2\right)(c_i) & \text{otherwise} \end{cases}$$

and, for $k \in 1, 2$, $defAx_k \in A_T$ *is an abstract domain element such that* $\vdash C_k \to defAx_k(\{U_k\}\,\mathrm{x}_i)$ *is provable and there is no element* $defAx_k' \in A_T$ *with* $defAx_k' \neq defAx_k$ *and* $defAx_k' \sqsubseteq defAx_k$. ◊

The constraints on $defAx_k$ state that those elements must be least ones in the abstract domain lattice. Note that there does not necessarily exist a *unique* such element; in this case, any element for which there is no suitable strictly smaller one suffices. For countable lattices with a given enumeration $e_1, e_2, \ldots, e_n, \ldots$ of the domain elements such that for $0 \leq i < l$, it does not hold that $e_l \sqsubseteq e_i$, $defAx_k$ is computable. Such an enumeration can always be constructed for finite domains by a BFS traversal of the lattice starting from the smallest element. For the sign analysis domain, the enumeration is $\bot, \mathrm{neg}, \mathrm{zero}, \mathrm{pos}, \mathrm{leq}, \mathrm{geq}, \top$. Generally, infinite domains should support *widening* [CC77] to ensure that suitable abstractions can be computed.

Proposition 3.31. *Technique 3.30 induces a family of join-semilattices of SE states, i.e. the operation* $\dot\sqcup_{abstr}$ *and its associated partial order relation* $\preceq$ *satisfy the axioms (SEL1) to (SEL5) of Definition 3.12.* ◊

Proof. We omit the proofs of the rather obvious properties (SEL1) to (SEL3).

For (SEL4), it suffices to show

$$a \,\dot\sqcup_{abstr}\, b = c \implies a \lesssim c$$

for any $a = (U_1, C_1, \varphi)$, $b = (U_2, C_2, \varphi)$, $c = (U^*, C_1 \vee C_2, \varphi) \in SEStates_{Prg}$ by Lemma 3.18. Assuming $a \,\dot\sqcup_{abstr}\, b = c$, we thus have to show that $concr(a) \subseteq concr(c)$. Let $\sigma \in concr(a)$, i.e. there is a model $(K_\Sigma, \sigma') \models C_1$ such that $\sigma = val_{(K_\Sigma,\sigma')}(U_1)(\sigma')$. Let K_Σ' be an expansion of K_Σ by the new constants c_i such that $I_{K_\Sigma'}(c_i) = \sigma(\mathrm{x}_i)$. Since C_1 is in the old language, $\left(K_\Sigma', \sigma'\right) \models C_1$ and $\left(K_\Sigma', \sigma'\right) \models C_1 \vee C_2$. By $\vdash C_k \to defAx_k(\{U_k\}\,\mathrm{x}_i)$, $k = 1, 2$, we know $\left(K_\Sigma', \sigma'\right) \models defAx_1(\{U_1\}\,\mathrm{x}_i)$ and, by

$$val_{\left(K_\Sigma',\sigma'\right)}(\{U_1\}\,\mathrm{x}_i) = val_{\left(K_\Sigma',\sigma'\right)}(U_1)\left(\sigma'\right)(\mathrm{x}_i) = val_{(K_\Sigma,\sigma')}(U_1)\left(\sigma'\right)(\mathrm{x}_i) = \sigma(\mathrm{x}_i) = I_{K_\Sigma'}(c_i),$$

$\left(K_\Sigma', \sigma'\right) \models defAx_1(c_i)$. Due to Definition 3.28, it thus holds that $\left(K_\Sigma', \sigma'\right) \models \left(defAx_1 \sqcup defAx_2\right)(c_i)$ and so $\left(K_\Sigma', \sigma'\right) \models \bigwedge C_i^{abs}$. Therefore, we have

$$val_{\left(K_\Sigma',\sigma'\right)}(U^*)\left(\sigma'\right) \in concr(c).$$

It suffices to show that, for all $i = 1, 2, \ldots, n$:

$$val_{\left(K_\Sigma',\sigma'\right)}(U^*)\left(\sigma'\right)(\mathrm{x}_i) = val_{(K_\Sigma,\sigma')}(U_1)\left(\sigma'\right)(\mathrm{x}_i).$$

We distinguish the following two cases:

Case "$(\star)$ is provable". In this case,

$$\begin{aligned} val_{\left(K_\Sigma',\sigma'\right)}(U^*)\left(\sigma'\right)(\mathrm{x}_i) &= val_{\left(K_\Sigma',\sigma'\right)}(\mathrm{x}_i := \{U_1\}\,\mathrm{x}_i)\left(\sigma'\right)(\mathrm{x}_i) \\ &= \left(\mathrm{x} \mapsto \begin{cases} \sigma'(\mathrm{x}) & \text{if } \mathrm{x} \neq \mathrm{x}_i \\ val_{\left(K_\Sigma',\sigma'\right)}(\{U_1\}\,\mathrm{x}_i) & \text{otherwise} \end{cases}\right)(\mathrm{x}_i) \\ &= val_{\left(K_\Sigma',\sigma'\right)}(\{U_1\}\,\mathrm{x}_i) \\ &= val_{\left(K_\Sigma',\sigma'\right)}(U_1)\left(\sigma'\right)(\mathrm{x}_i) \\ &\overset{(\dagger)}{=} val_{(K_\Sigma,\sigma')}(U_1)\left(\sigma'\right)(\mathrm{x}_i). \end{aligned}$$

where (†) is true since U_1 is in the language without the constants.

Case "otherwise". Then,

$$\begin{aligned}
val_{(K'_\Sigma,\sigma')}(U^*)(\sigma')(\mathrm{x}_i) &= val_{(K'_\Sigma,\sigma')}(\mathrm{x}_i := c_i)(\sigma')(\mathrm{x}_i)\\
&= \left(\mathrm{x} \mapsto \begin{cases}\sigma'(\mathrm{x}) & \text{if } \mathrm{x} \neq \mathrm{x}_i\\ val_{(K'_\Sigma,\sigma')}(c_i) & \text{otherwise}\end{cases}\right)(\mathrm{x}_i)\\
&= val_{(K'_\Sigma,\sigma')}(c_i) = I_{K'_\Sigma}(c_i) \overset{\text{def.}}{=} \sigma(\mathrm{x}_i)\\
&= val_{(K_\Sigma,\sigma')}(U_1)(\sigma')(\mathrm{x}_i).
\end{aligned}$$

For (SEL5), we have to show that $C^* \leftrightarrow (C \wedge C_{ax})$, where $\bigwedge C_1 \rightarrow C$ and $\exists \overline{v}; (C_{ax}[\overline{v}/\overline{c}])$, for $\overline{c} = c_1, c_2, \ldots, c_n$ and $\overline{v}$ is a tuple of suitable fresh logical variables, is provable. By definition, $C^* = (\bigwedge C_1 \vee \bigwedge C_2) \wedge \bigwedge C_i^{abs}$. We choose $C := \bigwedge C_1 \vee \bigwedge C_2$ and $C_{ax} := \bigwedge C_i^{abs}$. Obviously, $\bigwedge C_1 \rightarrow (\bigwedge C_1 \vee \bigwedge C_2)$. $\exists \overline{v}; (C_{ax}[\overline{v}/\overline{c}])$ equals

$$\exists \overline{v}; \left(\bigwedge \left\{ (defAx_1 \sqcup defAx_2)(v_i) : i = 1, 2, \ldots, n \right\}\right).$$

This formula is provable due to (2) of Definition 3.28, since neither $defAx_1$ nor $defAx_2$ are $\bot$, because otherwise $\vdash C_k \rightarrow defAx_k(\{U_k\}\,\mathrm{x}_i)$ would not hold for satisfiable C_k. □

3.3.5 Heap Treatment

All of the above discussed join technique are in principle able to deal with heap structures. Since heaps are modeled in KeY as program variables of the special type *Heap*, the techniques 3.20, 3.22 and 3.25 are able to treat heaps without special adaptations; Technique 3.30 would depend on a domain for the type *Heap*. However, the joining of whole heap expressions to, e.g., if-then-else expressions, leads to long terms and potential redundancies which may result in a suboptimal treatment of those by the automatic strategies of the KeY system. We therefore treat heaps in a "zip" procedure: Assume, for example, that we want to join two heap expressions

```
store(store(create(heap,s_2),s_2,<initialized>,TRUE),s_2,num,mul(num_0,-1))
```

and

```
store(store(create(heap,s_2),s_2,<initialized>,TRUE),s_2,num,num_0),
```

where `num` is negative in the corresponding first state and positive in the second. Instead of just connecting the whole terms by if-then-else, we simultaneously go deeper inside the expressions, as long as the top functions are equal (e.g., two "store" applications) and have the syntactically same expressions for objects / fields as targets, and apply join operators on stored values that differ. When two different top level functions are reached (e.g., a "create" and a "store" occurrence) or (syntactically) different objects / fields are targeted, we use if-then-else as a fallback. For the above example and a join with the sign lattice abstraction, the resulting heap expression is

```
store(store(create(heap,s_2),s_2,<initialized>,TRUE),s_2,num,geq),
```

where the constant `geq` is constrained by the formula `geq >= 0` in the antecedent. We define this procedure formally in Definition 3.32.

Definition 3.32 (General Heap Treatment). The function

$$joinHeaps : \mathrm{Terms}_\Sigma^{Heap} \times \mathrm{Terms}_\Sigma^{Heap} \times SEStates_{Prg} \times SEStates_{Prg} \rightarrow 2^{\mathrm{Form}_\Sigma} \times \mathrm{Terms}_\Sigma^{Heap}$$

is defined inductively over the structure of heap terms as follows:

$$joinHeaps(h_1, h_2, s_1, s_2) := \begin{cases}
(\emptyset, h_1) & \text{if } h_1 = h_2\\
(C \cup ((s_1, v_1) \circledvee (s_2, v_2)), & \text{if } h_1 = store(h'_1, o, f, v_1) \text{ and}\\
\quad store(h', o, f, (s_1, v_1) \oplus (s_2, v_2))) & \quad h_2 = store(h'_2, o, f, v_2)\\
(C, create(h', o)) & \text{if } h_1 = create(h'_1, o) \text{ and}\\
 & \quad h_2 = create(h'_2, o)\\
(\emptyset, if(C_1)\, then(h_1)\, else(h_2)) & \text{otherwise}
\end{cases}$$

where

- $joinHeaps(h'_1, h'_2, s_1, s_2) = (C, h')$ – the recursive step,

- C_1 is the path condition of s_1, and
- $\oplus : \mathit{SEStates}_{Prg} \times \mathrm{Terms}_{\Sigma}^{T} \times \mathit{SEStates}_{Prg} \times \mathrm{Terms}_{\Sigma}^{T} \rightarrow \mathrm{Terms}_{\Sigma}^{T}$ and $\varovee : \mathit{SEStates}_{Prg} \times \mathrm{Terms}_{\Sigma}^{T} \times \mathit{SEStates}_{Prg} \times \mathrm{Terms}_{\Sigma}^{T} \rightarrow 2^{\mathrm{Form}_{\Sigma}}$ are families of join operators for computing join values resp. additional path condition constraints for terms of types T. $\Diamond$

The second component of the result of *joinHeaps* is a term of type *Heap*, the joined heap, while the first component contains the set of formulae constraining introduced Skolem constants, if any. Instances of operators $\varovee$ and $\oplus$ in the above definition could return, for the example of abstract weakening, a defining axiom for a fresh Skolem constant and the constant itself, respectively. The difference to the operators $\varovee_{abstr}$ and $\oplus_{abstr}$ defined on SE states consists in the restriction to consider exactly one term instead of all right sides of contained program variables; otherwise, they work analogously. For the "otherwise" case, we also could have chosen the introduction of a fresh Skolem constant of type *Heap* as a basis for joining heaps by techniques 3.20 (full anonymization) or 3.25 (if-then-else-antecedent).

While this general approach does not really consider special characteristics of objects, there are methods in literature focusing on heap structures. Anand et al. [APV06] propose a technique for lists and arrays building upon shape analysis techniques [SRW02]. They contract list elements to so-called "summary nodes", which facilitates the joining of branches whenever such an abstraction "subsumes" another leaf in the SET. We refer to this in Section 6.1.

4 Implementation

Based on our theoretical framework presented in Chapter 3, we implemented support for joining branches of proof trees in the KeY system. The implementation consists of five components: (1) An abstract class `JoinRule` following the template method pattern and a class `JoinRuleUtils` comprising a set of commonly used static methods for join operations, (2) a framework for abstraction based on finite lattices, (3) a class `CloseAfterJoin` for closing partner goals after a join, (4) a macro for running proof steps on a sequent containing Java code until a potential join point is reached, and (5) an implementation of four sample join rules specializing `JoinRule`. Subsequently, we elaborate on these components.

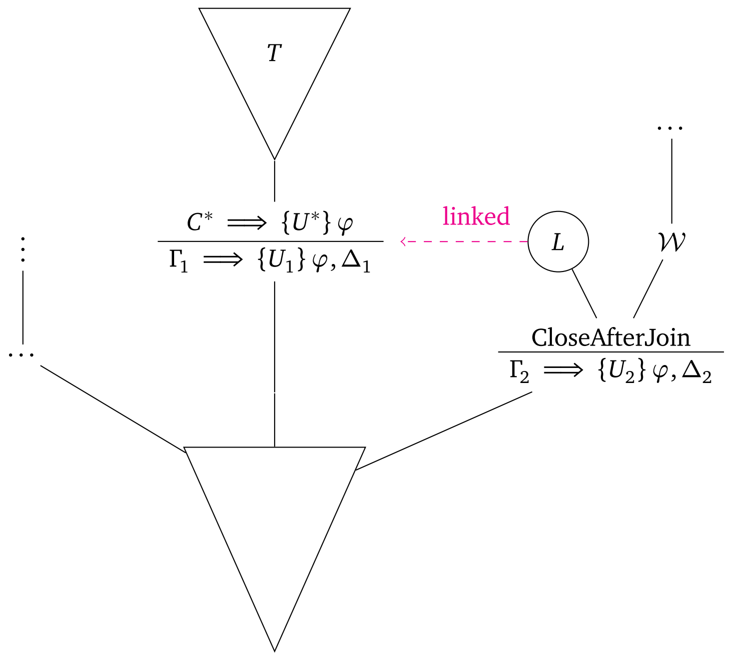
4.1 JoinRule and JoinRuleUtils

The class `JoinRule` is responsible for all tasks accumulated during the joining of branches in KeY proof except for the actual computation of a join state for two given symbolic execution states. Its implementation follows the template method pattern: Concrete join rules must implement the abstract method `joinValuesInStates(LocationVariable,SymbolicExecutionState,Term,SymbolicExecutionState,Term,Services)` for joining two terms in their respective SE states. `JoinRuleUtils` comprises a quite large set of static methods that are used by join classes. These methods can be categorized into simple auxiliary methods, methods related to general logic (syntax, provability, simplification, and calculus-related), and methods closely related to join operations. The UML class digram in Figure 6.1 visualizes the dependencies between the two general join classes and the concrete join rules; Figure 6.2 offers a more detailed view into the structure of `JoinRule` and `JoinRuleUtils`. Listing 4.1 shows an excerpt of the implementation of `JoinRule`. In particular, `JoinRule` checks whether joining is applicable for a given position in a sequent; for investigating whether this is the case, the method `JoinRule.findPotentialJoinPartners(Goal, PosInOccurrence)` searches the goals of the proof tree for potential partner goals. For enabling the user to choose partners among the candidates, the dialog class `JoinPartnerSelectionDialog` is used. After the join partner selection, the node on which the rule is being applied is joined with the partners in a loop (lines 27 – 33 in Listing 4.1). Since join operations are required to be commutative and associative (→ Definition 3.12), the order of the join does not matter; therefore, concrete join classes inheriting from `JoinRule` only need to specify a method for joining values in *two* SE states. After the join, the rule `CloseAfterJoin` (→ Section 4.3) is applied on the partner goals. The method `joinStates(...)` may be overridden for join operations requiring special behavior; the same holds for the method `joinHeaps(...)` realizing the procedure described in Definition 3.32.

Listing 4.1: Excerpt of the JoinRule implementation

```
1  public abstract class JoinRule extends JoinRuleUtils implements BuiltInRule {
2     // ...
3
4     @Override
5     public final ImmutableList<Goal> apply(Goal goal, final Services services,
6           RuleApp ruleApp) throws RuleAbortException {
7
8        // ...
9        ImmutableList<Pair<Goal, PosInOccurrence>> joinPartners =
10             findJoinPartners(newGoal, pio);
11
12       // ...
13       // Convert sequents to SE states
14       ImmutableList<SymbolicExecutionState> joinPartnerStates = ImmutableSLList.nil();
15       for (Pair<Goal, PosInOccurrence> joinPartner : joinPartners) {
16          Triple<Term, Term, Term> partnerSEState =
17                sequentToSETriple(joinPartner.first, joinPartner.second, services);
18
19          joinPartnerStates = joinPartnerStates.prepend(new SymbolicExecutionState(
20                partnerSEState.first, partnerSEState.second, joinPartner.first.node()));
21       }
22
```

```
      SymbolicExecutionStateWithProgCnt thisSEState =
            sequentToSETriple(newGoal, pio, services);

      // The join loop
      SymbolicExecutionState joinedState = new SymbolicExecutionState(
            thisSEState.first, thisSEState.second, goal.node());

      for (SymbolicExecutionState state : joinPartnerStates) {
         joinedState = joinStates(joinedState, state, thisSEState.third, services);
         joinedState.setCorrespondingNode(goal.node());
      }

      Term resultPathCondition = joinedState.second;
      resultPathCondition = trySimplify(services.getProof(), resultPathCondition, true);

      // ...
      // Close partner goals
      for (Pair<Goal, PosInOccurrence> joinPartner : joinPartners) {
         closeJoinPartnerGoal(
               newGoal.node(),
               joinPartner.first,
               joinedState,
               sequentToSEPair(joinPartner.first, joinPartner.second, services),
               thisSEState.third);
      }

      // ...
      return newGoals;
   }

   protected SymbolicExecutionState joinStates(
         SymbolicExecutionState state1,
         SymbolicExecutionState state2,
         Term programCounter,
         Services services) {

      // ...
      for (LocationVariable v : progVars) {
         // ...

         Pair<HashSet<Term>, Term> joinedVal =
            joinValuesInStates(v, state1, rightSide1, state2, rightSide2, services);

         newElementaryUpdates = newElementaryUpdates.prepend(
            tb.elementary(
                  v,
                  joinedVal.second));

         newPathCondition = tb.and(
            newPathCondition,
            tb.and(joinedVal.first));

         // ...
      }
      // ...

   }

   // ...

   protected abstract Pair<HashSet<Term>, Term> joinValuesInStates(
         LocationVariable v,
         SymbolicExecutionState state1,
         Term valueInState1,
         SymbolicExecutionState state2,
         Term valueInState2,
         Services services);

   // ...
}
```



T
$C^* \Longrightarrow \{U^*\}\,\varphi$
$\Gamma_1 \Longrightarrow \{U_1\}\,\varphi, \Delta_1$
linked
L
$\mathcal{W}$
CloseAfterJoin
$\Gamma_2 \Longrightarrow \{U_2\}\,\varphi, \Delta_2$

Figure 4.1: Visualization of `CloseAfterJoin` rule. L is the linking node, $\mathcal{W}$ contains the proof obligation corresponding to the logical weakening formula.

4.2 The Abstraction Framework

Our implemented lattice abstraction framework (UML diagram in Figure 6.3) centers around three classes. The abstract class `AbstractDomainElement` represents an element of an abstract lattice domain. Basically, an abstract domain element encapsulates a defining axiom that can be obtained by calling the method `getDefiningAxiom(Term, Services)`. An abstract domain inherits from the class `AbstractDomainLattice`: Most importantly, it has to implement the methods `join(AbstractDomainElement, AbstractDomainElement)` and `iterator()`. The abstract join rule `JoinWithLatticeAbstraction` uses `iterator()` to obtain an ordered enumeration of abstract domain elements; for each element, it checks whether its defining axiom holds for the concrete term that is to be abstracted. The such determined abstract elements are joined with the method `join(AbstractDomainElement, AbstractDomainElement)`. We implemented two abstract domains based on this framework: the sign lattice domain for integers and a simple domain for booleans. Note that this framework is designed to cope with finite and quite small domains only. It probably has to be extended to support more complicated abstract domains (→ Section 6.1).

4.3 The Partner Goals Closing Rule

As we pointed out in Remark 3.17, the defocusing join rule with two conclusions bears theoretical complications. Since furthermore the tree structure is strongly incorporated in the KeY system, a real DAG representation is definitely difficult to achieve. Instead of applying the weakening rule of Remark 3.17 twice and re-using the computed proof subtree of one of the branches (→ Figure 3.3), we apply the special rule `CloseAfterJoin` to partner goals of a join operation (UML diagram: Figure 6.4). The rule adds two sub branches to a partner goal. One of these branches contains a single node linking the partner goal to the corresponding join node. We added a field `linkedNode` to the `Node` class for the purpose of establishing this connection. If the join node is closed, an action is triggered that closes all of its linked nodes. The second branch added after the partner goal contains a proof obligation expressing that the join node is logically weaker than its partner node (→ Definition 3.7). Recall that by Proposition 3.11, logical weakening is equivalent to semantic weakening. Thus, this measure, assuming that the quite simple condition (SEL5) of Definition 3.12 is obeyed, ensures that proofs are not rendered unsound even for join methods with implementation failures, or such methods for which the conformity with the axioms of Definition 3.12 has not been theoretically proven. A visualization of the result of a `CloseAfterJoin` application, based upon Figure 3.3, is shown in Figure 4.1. The node after the sequent

$\Gamma_1 \implies \{U_1\}\varphi, \Delta_1$ is the *join node*, the node corresponding to the sequent $\Gamma_2 \implies \{U_2\}\varphi, \Delta_2$ is the *partner node*. The logical weakening formula is added to the node $\mathcal{W}$, whereas the node L links to the join node. The subtree T does appear only once and is not, as in Figure 3.3, appended to the partner node.

Listing 4.2 shows the part of the method `CloseAfterJoin.apply(Goal, Services, RuleApp)` which is responsible for linking partner nodes to their join nodes. Whenever the proof tree is extended or pruned, the state of the join node is checked. In the case that the join node has been deleted from the tree, the linked node is marked as independent again. Thus, the proof is still sound, which it would not necessarily be if we did ultimately close partner goals after a join. If the branch of the join node has been closed, the linked goal after the partner node is also being closed.

Listing 4.2: Linking of partner nodes to join nodes

```
services.getProof().addProofTreeListener(new ProofTreeAdapter() {
    @Override
    public void proofGoalsChanged(ProofTreeEvent e) {
        if (joinNode.isClosed()) {
            // The joined node has been closed; now also close this node.
            services.getProof().closeGoal(linkedGoal);
        }
    }

    @Override
    public void proofPruned(ProofTreeEvent e) {
        if (!proofContainsNode(e.getSource(), joinNodeF)) {
            // The joined node has been pruned; now mark this node
            // as not linked and set it to automatic again.
            linkedGoal.node().setLinkedNode(null);
            linkedGoal.setEnabled(true);
        }
    }
});
```

4.4 Macro for Execution until Join Points

For applying join rules on suitable nodes in a proof, it is clearly undesirable to generate proofs completely by hand or to prune automatically generated proofs with the intention to create a situation where joining is possible. In our implementation, we did not establish an inclusion of our join rules into the automatic Java DL strategies; this could be an object to future work. However, we created the macro `FinishSymbolicExecutionUntilJoinPointMacro` (→ Figure 6.5) which, after started on a sequent containing Java code, allows the execution of a parent strategy only until a previously registered break point has been reached. Whenever a Java block is seen that contains either an if block, a while loop with a break, or a try-catch block, the respective next statement afterward is added to the set of break points. Thus, the macro behaves like the given parent strategy, but might stop the proof procedure at points facilitating an application of join rules. This induces a semi-automatic proof strategy consisting of a repeated execution of the macro interleaved with manual applications of join rules on the generated proof nodes (→ Section 5.1). The creation of a fully automatic strategy / macro based on this procedure could be a first step into the direction of proof generation with automatic join rule applications in KeY.

Listing 4.3 contains an excerpt of the method `isApprovedApp(RuleApp, PosInOccurrence, Goal)` which decides whether a proposed rule application is permissible or not. In line 4, execution is stopped whenever a breakpoint is reached; in line 10, breakpoints are searched for and registered.

4.5 Sample Join Rule Implementations

We implemented concrete join rules based on our implemented framework and the techniques 3.20, 3.22, 3.25, and 3.30. The techniques for heap joining follow Definition 3.32. As depicted in the UML diagram of Figure 6.1, all join classes depend on `JoinIfThenElse`, in particular on the methods for creating if-then-else terms (→ Figure 6.6). The classes `JoinWithLatticeAbstraction` and `JoinIfThenElseAntecedent` use if-then-else as fall back for heap joining; `JoinIfThenElseAntecedent` further employs the method `createDistFormAndRightSidesForITEUpd(LocationVariable, ...)` for

Listing 4.3: FinishSymbolicExecutionUntilJoinPointMacro: Registering join points and deciding about continuation of execution

```
@Override
public boolean isApprovedApp(RuleApp app, PosInOccurrence pio, Goal goal) {
   if (!hasModality(goal.node())) { return false; }
   if (hasBreakPoint(goal.sequent().succedent())) { return false; }

   if (pio != null) {
      JavaBlock theJavaBlock = getJavaBlockRecursive(pio.subTerm());

      //...
      breakpoints.addAll(findJoinPoints((StatementBlock) theJavaBlock.program()));

      //...
   }

   return super.isApprovedApp(app, pio, goal);
}
```

Listing 4.4: Step two of distinguishing formulae simplification (Part 1)

```
public static Term trySimplify(
      final Proof parentProof, final Term term, boolean countDisjunctions) {
   try {
      Term simplified = simplify(parentProof, term);

      if (countAtoms(simplified) < countAtoms(term) &&
            (countDisjunctions(simplified, false) < countDisjunctions(term, false))) {
         return simplified;
      }
   } catch (ProofInputException e) {}

   return term;
}
```

obtaining the shortest distinguishing formula given two path conditions. We briefly elaborate on this simplification and another one for optimizing the structural complexity of disjunctive path conditions.

Simplified Distinguishing Formulae

This is an optimization of Technique 3.22 aimed to increase the proof performance by generating shorter if-then-else terms. Given two SE states (U_1, C_1, φ) and (U_2, C_2, φ), and assuming that C_1 and C_2 are contradicting, the terms

$$\mathit{if}\,\Big(\bigwedge C_1\Big)\,\mathit{then}\,(\{U_1\}\,\mathrm{x})\,\mathit{else}\,(\{U_2\}\,\mathrm{x}) \quad \text{and}$$
$$\mathit{if}\,\Big(\bigwedge C_2\Big)\,\mathit{then}\,(\{U_2\}\,\mathrm{x})\,\mathit{else}\,(\{U_1\}\,\mathrm{x})$$

are equivalent. The implementation chooses the shortest of those after having applied two steps of simplifications on the path conditions. The first step includes removing all common atoms from the path conditions, since they are not needed for distinguishing the states. In the second step, a side proof is run on the path conditions. If the conjunction of the open goals in the resulting proof is simpler regarding the number of atoms and disjunctions, this conjunction is chosen as distinguishing formula. Otherwise, the result of the first step is returned. The listings 4.4 and 4.5 show the corresponding code for this simplification step.

Optimized Path Conditions

Further optimizations take place in the creation of the disjunctive path condition. Instead of choosing the canonical $\bigwedge C_1 \vee \bigwedge C_2$ for the path condition of a join node, the method `JoinRuleUtils.createSimplifiedDisjunctivePathCondition(Term, Term, Services)` performs a two-step simplification to generate a more concise path condition. In step one, complementary elements

Listing 4.5: Step two of distinguishing formulae simplification (Part 2)

```java
private static Term simplify(Proof parentProof, Term term)
      throws ProofInputException {
   final Services services = parentProof.getServices();
   final ApplyStrategyInfo info = tryToProve(term, services, true);

   try {
      // The simplified formula is the conjunction of all open goals
      ImmutableList<Goal> openGoals = info.getProof().openEnabledGoals();
      final TermBuilder tb = services.getTermBuilder();
      if (openGoals.isEmpty()) { return tb.tt(); }
      else {
         ImmutableList<Term> goalImplications = ImmutableSLList.nil();
         for (Goal goal : openGoals) {
            Term goalImplication = sequentToFormula(goal.sequent(), services);
            goalImplications = goalImplications.append(goalImplication);
         }

         return tb.and(goalImplications);
      }
   } finally {
      SideProofUtil.disposeOrStore(
            "Simplification␣of␣"
                  + ProofSaver.printAnything(term,
                        parentProof.getServices()), info);
   }
}
```

of $\bigwedge C_1$ and $\bigwedge C_2$ are removed, i.e. if $C_1 \equiv \psi_1 \land \varphi$ and $C_2 \equiv \psi_2 \land \neg\varphi$, the result will be $\psi_1 \lor \psi_2$. This simplification is admissible since we assume that $\bigwedge C_1 \leftrightarrow \neg \bigwedge C_2$, and the sequent

$$(\psi_1 \land \varphi) \leftrightarrow \neg(\psi_2 \land \neg\varphi) \implies ((\psi_1 \land \varphi) \lor (\psi_2 \land \neg\varphi)) \leftrightarrow (\psi_1 \lor \psi_2)$$

formalizing this simplification is valid (which can be proven by hand or, for instance, in KeY). Since we also assume $\bigwedge C_1 \leftrightarrow \neg \bigwedge C_2$ in the proof of Proposition 3.24, which is justified for the reason that it holds in symbolic execution without joins and our join rules do not violate it, the assumption is also applicable at this place. Intuitively, the removal of contradicting conjuncts in joined path conditions is sensible since the condition that made symbolic execution split is no longer relevant after all of the such generated branches are joined again. The measure of removing it from the resulting path condition by syntactic checks spares KeY having to apply a potentially large number of rules for obtaining the same result. Finally, in the second step, we apply a distributivity law to further simplify the structure of the formula. We decided not to include the automatic simplification described in the above paragraph for optimized path conditions, since our experiments showed that the time overhead induced by this process is out of proportion to the achieved results.

Sign Lattice and Boolean Abstraction

For showing the feasibility of performing an abstract interpretation with KeY by the means of our theoretical and implemented framework, we implemented a join rule `JoinWithSignLattice` for joining states based on the sign domain (→ Example 3.29). The rule itself is very simple (→ Figure 6.3): It just returns an instance of `SignAnalysisLattice` for the Integer sort, and an instance of `BooleanLattice` for the Boolean sort. For other sorts, it returns null; in this case, the super class `JoinWithLatticeAbstraction` generates an if-then-else expression for the corresponding values. The core of the implementation are the classes realizing the abstract domains and their elements. Figure 6.7 shows the UML diagram for the sign analysis abstract domain lattice, Figure 6.8 for the Boolean domain. We created abstract classes representing the respective domain element types, and one concrete class for each domain element. Note that our framework does not require one class per abstract element; it would also be possible to define a single concrete class the objects of which are capable of remembering the elements they represent and of returning the right defining axioms. Since the domain element classes like `Neg` and `Leq` are all singletons, there is no substantial difference; this choice is just a matter of style. For extending our framework to more complex abstract domains (see, e.g., [Cou01; Cou+05]), we suppose that our abstraction framework will have to be extended to allow for more flexibility.

5 Evaluation and Case Study

Incorporating joins during symbolic execution can *increase* as well as *decrease* the length of proofs based on chosen parameters for proof generation and the concrete scenario. In Section 5.1, we investigate this impact for a set of example programs. Section 5.2 concerns the benefits of if-then-else joins for increasing the precision of an information flow analysis.

5.1 Performance Evaluation

We evaluate the performance of our implementation using four simple and two more complex Java programs: (1) A method containing an if-then-else block for computing the absolute of an integer (→ Listing 5.1), (2) a method computing the absolute of an integer inside an object, returning a new object (→ Listing 5.2), (3) a method computing the quotient of two integers, catching a potential exception arising from a division by zero (→ Listing 5.3), (4) a method comprising a while loop with a break statement for finding an element in an integer array (→ Listing 5.4), (5) an integer multiplication method (→ Listing 5.5) also used, in a slightly different form, in [HSS09], and (6) a method computing the greatest common divisor of two integers (→ Listing 5.6). The examples (1) to (4) serve as basic representatives for the cases where merges typically may take place, the most obvious amongst those being if-then-else and try-catch blocks. Example (4) allows for a merge of the "break" sub branch in the "preserves invariant" branch with the "use case" branch; those branches arise after an application of the loop invariant rule. Example (2) demonstrates that the implementation is also capable of handling heaps. The contract of Example (5) is proven using loop unwinding; for this reason, we restrict the range of the input values. The chosen bound 5 is rather arbitrary and quite low, but that is just for convenience of the proof; any concrete value at this place would allow for proving the contract with unwinding. The chosen join points, for this example, reside at the beginning of the while loop. Execution is split after the if statement in the body of the loop, therefore an alternative, earlier join point would be at line 16 in the listing. We chose the point two statements later to minimize the number of manual interactions with the proof. The gcd Example (6) involves a normalization before calling a helper method: the input variables are converted into their absolutes in lines 8 and 9 in the listing. In the proof, the method `gcdHelp` is integrated by its method contract.

Table 5.2 shows the experimental results for the example programs under chosen execution strategies. The recorded parameters comprise the number of nodes in the resulting proof of the respective method contract as well as the corresponding numbers of branches (excluding the branches / nodes for the logical weakening check after joins), rule and one step simplifier applications, joins, and the necessary number of manual user interactions for the proof. In the last column, the improvement of the respective strategy over the baseline strategy (fully automatic proof with/without one step simplification) is listed. Improvements of at least 2% are highlighted in green, declines of at least 2% in red; small changes are of gray color. The employed strategies are explained in Table 5.3. The "canonical" strategies, join-ite, join-ite2 and join-sign, involve a repeated symbolic execution until the next join point with the macro we developed for this purpose (→ Section 4.3), followed by a join of applicable leaf nodes; it should be possible to derive fully automatic strategies from this procedure in a quite straightforward manner. For the multiplication and gcd examples, we make use of specialized, less canonical strategies that are better suited for those two examples.

For the simple examples (1) to (4) with activated one step simplifier, joining branches with if-then-else techniques is clearly not beneficial. Those examples highly benefit from one step simplifier applications which are not applicable in that way after a join operation. Consider the abs example (2) with the join-ite2 strategy: In this case, the proof with join is about 23% longer than the corresponding automatic proof. The proof splits into two branches corresponding to the sign of the input variable. In the proof with join, the "$\texttt{num} \geq 0$" branch is 11 nodes shorter. However, the "$\texttt{num} < 0$" branch in the automatic proof can be closed quite quickly due to some very efficient one step simplifier applications: The goal $\Delta, \texttt{num} \leq -1 \implies \Gamma, \texttt{num} * -1 \geq 0$ in the automatic proof corresponds to the more complicated goal $\Delta', \texttt{num} \leq -1 \rightarrow \texttt{result\_1} = \texttt{num} * -1, \texttt{num} \leq -1 \vee \texttt{result\_1} = \texttt{num} \implies \Gamma, \texttt{result\_1} \geq 0$ in the proof with join. Therefore, the automatic proof can be closed in one step after the usage of the class invariant axiom for SimpleMath, whereas its counterpart with join needs another 37 steps including a cut over the sign of `num`. These observations also explain why joining with if-then-else methods is less disadvantageous, and sometimes even beneficial, for the very same examples with deactivated one step

Example	Strategy	# Nodes / # Branches	# Rule Apps / # OSS Apps	# Joins	# User Inter-actions	Improvement of # Nodes to baseline
div	auto	185 / 9	415 / 52	0	0	–
div	join-ite	196 / 9	394 / 51	1	1	-5.95%
div	join-ite2	193 / 8	394 / 51	1	1	-4.32%
div	auto-no-OSS	401 / 9	400 / 0	0	0	–
div	join-ite-no-OSS	380 / 9	379 / 0	1	1	5.24%
div	join-ite2-no-OSS	374 / 6	374 / 0	1	1	6.73%
abs	auto	78 / 5	155 / 18	0	0	–
abs	join-ite	79 / 5	136 / 16	1	1	-1.28%
abs	join-ite2	96 / 5	153 / 21	1	1	-23.08%
abs	join-sign	63 / 4	122 / 14	1	1	19.23%
abs	auto-no-OSS	158 / 5	157 / 0	0	0	–
abs	join-ite-no-OSS	138 / 5	137 / 0	1	1	12.66%
abs	join-ite2-no-OSS	155 / 5	154 / 0	1	1	1.90%
absObj	auto	211 / 10	451 / 52	0	0	–
absObj	join-ite	219 / 11	418 / 45	5	2	-3.79%
absObj	join-ite2	249 / 11	448 / 51	5	2	-18.01%
absObj	join-sign	190 / 9	389 / 43	5	2	9.95%
absObj	auto-no-OSS	444 / 10	443 / 0	0	0	–
absObj	join-ite-no-OSS	396 / 11	395 / 0	5	2	10.81%
absObj	join-ite2-no-OSS	426 / 11	425 / 0	5	2	4.05%
find	auto	695 / 21	1496 / 114	0	0	–
find	join-ite	702 / 19	1466 / 107	1	1	-1.01%
find	join-ite2	702 / 19	1466 / 107	1	1	-1.01%
find	auto-no-OSS	1438 / 21	1437 / 0	0	0	–
find	join-ite-no-OSS	1427 / 19	1426 / 0	1	1	0.76%
find	join-ite2-no-OSS	1421 / 19	1420 / 0	1	1	1.18%
multiply	auto	1062 / 28	1576 / 189	0	0	–
multiply	mult1	888 / 25	1284 / 158	1	2	16.38%
multiply	mult1-ite2	915 / 25	1308 / 158	1	2	13.84%
multiply	mult2	(timeout)		1	2	N/A
multiply	mult3	1044 / 26	1426 / 178	2	4	1.69%
gcd	auto	8371 / 37	10312 / 675	0	0	–
gcd	gcd-ite	8048 / 32	8923 / 609	2	2	3.86%
gcd	gcd-ite2	7315 / 31	8084 / 624	2	2	12.61%
gcd	auto-no-OSS	9103 / 34	9102 / 0	0	0	–
gcd	gcd-ite2-no-OSS	7197 / 29	7196 / 0	2	2	20.94%

Table 5.2: Performance evaluation for the example programs div, abs, absObj, find, multiply, and gcd

Strategy Name	Description
auto	Full usage of KeY's automated proof search function.
join-ite	Repeatedly run symbolic execution until join point using the FinishSymbolicExecutionUntilJoinPointMacro and join applicable brancehs by if-then-else until symbolic execution is finished; then proceed automatically.
join-ite2	Like join-ite, but with joins using the if-then-else-antecedent rule.
join-sign	Like join-ite, but with joins using sign lattice abstraction.
mult1	Deactivate loop treatment and start automatic strategy. Unwind the while loop once, and use "Full Auto Pilot" macro (applied on the sequent, with deactivated loop treatment). Two occurrences of while loops occur. Join the two branches by if-then-else join, automatically proceed from there.
mult1-ite2	Like mult1, but with joins using the if-then-else-antecedent rule.
mult2	Like mult1, but with using the automatic strategy instead of the "Full Auto Pilot" macro.
mult3	Like mult1, but with one additional unwinding and join step.
gcd-ite	Two times execution until join point with FinishSymbolicExecutionUntilJoinPointMacro followed by an if-then-else join, then start of automatic proof engine. The method `gcdHelp` is included by contract.
gcd-ite2	Like gcd-ite, but with joins using the if-then-else-antecedent rule.
*-no-OSS	Strategy "*" with deactivated OneStepSimplifier.

Table 5.3: Strategies used in performance evaluation.

simplification. An investigation of the abs example without one step simplification shows that in the "$\mathtt{num} \geq 0$" branch, 31 nodes are saved, while in the "$\mathtt{num} < 0$" branch the additional nodes for case distinctions are furthermore partly compensated due to some rule applications that previously appeared redundantly. Another reason for the negative results of the simple examples is the late occurrence of the join. When joining directly before the last statement, there is not much code left for which a redundant execution can be avoided; the price for the more complicated expressions after the join cannot be compensated by saving redundancy.

The more extensive examples (5) and (6) show that it is possible to achieve shorter proofs even with the general if-then-else / if-then-else-antecedent join rules and usage of one step simplification. The gcd example is furthermore the only case where if-then-else-antecedent shows better results than the if-then-else method. The largest improvement in our measurements for the if-then-else methods could be measured in the gcd example (6) for the difference between automatic execution without one step simplifier and the gcd-ite2-no-OSS strategy: An improvement of 20.94% in the number of nodes of the proofs. This even exceeds the abs example (2) with the specialized join method by sign lattice abstraction.

For the multiply example (5), we discovered that joins with if-then-else might disturb the automatic strategies of KeY such that the proof procedure does not terminate; KeY was, in the case of the strategy mult2 making use of the "Play" button starting automatic execution, not able to derive the terminating condition for the while loop, resulting in endless unwinding operations. However, when using the macro "Full Auto Pilot" on the whole sequent, the proof did indeed terminate, in this case even with an improvement over the number of nodes and branches when performing one join operation. We discovered that the sequent before the join is more complicated when using the automatic strategy than when using the "Full Auto Pilot" macro. A closer evaluation of this problem might be in the scope of future work.

5.2 Case Study: Information Flow Analysis

Information flow between variables in a program is the transmission of some kind of information from one program variable to another one. Certain types of flows may be undesirable, an observation that motivates imposing *information flow policies* on programs. For instance, information about a secret password should not be leaked to a program variable that is communicated to the outside, maybe over a network connection. We investigate a simple class of policies that involve a partitioning of

Listing 5.1: The abs example – If block

```
/*@ public normal_behavior
  @ ensures \result >= 0;
  @*/
public int abs(int num) {
    int result;

    if (num < 0) {
        result = -num;
    } else {
        result = num;
    }

    return result; // join point
}
```

Listing 5.2: The absObject example – Join with heaps

```
/*@ public normal_behavior
  @ ensures \result.num >= 0;
  @*/
public SimpleMath absObject() {
    SimpleMath result = new SimpleMath();

    if (num < 0) {
        result.num = -num;
    } else {
        result.num = num;
    }

    return result; // join point
}
```

Listing 5.3: The div example – Exceptional control flow

```
/*@ public normal_behavior
  @ ensures divisor != 0 ==> \result == divident / divisor;
  @ ensures divisor == 0 ==> \result == Integer.MAX_VALUE;
  @*/
public int div(int divident, int divisor) {
    int result;

    try {
        result = divident / divisor;
    } catch (ArithmeticException e) {
        result = Integer.MAX_VALUE;
    }

    return result; // join point
}
```

Listing 5.4: The find example – While loop with break statement

```
/*@ public normal_behavior
  @
  @  ensures ((\exists int i; i >= 0 && i < arr.length; arr[i] == toFind)
  @             ==> arr[\result] == toFind) &&
  @           ((\forall int i; i >= 0 && i < arr.length; arr[i] != toFind)
  @             ==> \result == -1);
  @  assignable \nothing;
  @*/
public int find(int[] arr, int toFind) {
    int curPos = 0;

    /*@ loop_invariant
      @   (\forall int i; i >= 0 && i < curPos; arr[i] != toFind) &&
      @   curPos >= 0;
      @
      @ decreases arr.length - curPos;
      @ assignable curPos;
      @*/
    while (curPos < arr.length) {
        if (arr[curPos] == toFind) {
            break;
        }

        curPos = curPos + 1;
    }

    if (curPos < arr.length) { // join point
        return curPos;
    } else {
        return -1;
    }
}
```

Listing 5.5: The multiply example – Loop unrolling

```
/*@ public normal_behavior
  @ requires x0 >= 0 && y0 >= 0;
  @ requires x0 < 5 && y0 < 5; // bounds for successful unwinding
  @ ensures \result == x0 * y0;
  @*/
int multiply(int x0, int y0) {
    int x = x0;
    int y = y0;
    int z = 0;

    while (x != 0) { // join point
        if (x % 2 != 0) {
            z += y;
        }

        x = x / 2;
        y = y * 2;
    }

    return z;
}
```

Listing 5.6: The gcd example – Normalization before method call

```
/*@ public normal_behavior
  @ ensures (a != 0 || b != 0) ==>
  @         (a % \result == 0 && b % \result == 0 &&
  @           (\forall int x; x > 0 && a % x == 0 && b % x == 0;
  @             \result % x == 0));
  @*/
public static int gcd(int a, int b) {
    if (a < 0) a = -a;
    if (b < 0) b = -b; // join point

    int big, small;    // join point
    if (a > b) {
        big = a;
        small = b;
    } else {
        big = b;
        small = a;
    }

    return gcdHelp(big, small);
}
```

program variables into two classes “high” and “low”, disallowing a flow from variables classified “high” to variables classified “low”, such that a “low” observer cannot infer information about the initial values of the “high” variables.

A common technique for a language-based analysis of information flow is based on specialized type systems [SM06]. Such systems should necessarily be sound, i.e. they never classify insecure programs as secure. However, they will classify certain secure programs as insecure, due to a lack of sensitivity to the particular control-flow of a program. We subsequently propose a quite simple information flow analysis technique based on symbolic execution which is sound for our chosen examples, but classifies some secure examples as insecure. We then show how the if-then-else join method can be utilized to achieve a classification as “secure” for those intuitively secure examples. Our goal is not to devise a general sound and precise analysis technique, but to show how the precision of an example analysis can be improved using join methods. We believe that it should be possible to similarly extend existing approaches for information flow analysis based on symbolic execution, thereby retaining the soundness of these systems and increasing their precision.

Algorithm 1 Information flow analysis without joins

Require: p is a Java program with program variables $\overline{h}$ classified as “high” and $\overline{l}$ classified as “low”; $P(\ldots)$ is a fresh predicate accepting $\overline{l}$ as inputs.
Ensure: Returns true only if p does not contain a flow from a variable in $\overline{h}$ to a variable in $\overline{l}$.

```
 1: problem ← <\{ try { p } catch (Exception e) {} }\> P($\overline{l}$)
 2: Load problem into KeY and finish symbolic execution
 3: $G$ ← all leafs in the resulting SET containing $P$
 4: for all $g \in G$ do
 5:     Apply as many OneStepSimplifier applications on $g$ as possible
 6:     Transform $g$ to the form $\Gamma \implies P(\ldots)$ by “negation right” applications
 7:     if $\Gamma$ contains an atomic formula containing a variable in $\overline{h}$ or
            $P(\ldots)$ contains a term comprising a variable in $\overline{h}$ then
 8:         return false
 9:     end if
10: end for
11:
12: return true
```

We investigate the behavior of Algorithm 1 based on five examples, where `l`, `res` are “low” variables and `h`, `secret` are “high” variables.

Listing 5.7: "Exceptional Flow" Example

```
public class ExcptFlow {

    private /*@ spec_public @*/ int secret;

    public ExcptFlow(int secret) {
        this.secret = secret;
    }

    /* Roughly following "RIFLE: An Architectural
       Framework for User-Centric Information-Flow Security"
       by Vachharajani et al. */
    public boolean insecureExceptional(int input) {
        if (input == secret) {
            work();
        }

        return true;
    }

    public boolean secureExceptional(int input) {
        if (input == secret) {
            try {
                work();
            } catch (Exception e) {}
        }

        return true;
    }

    /*@ public exceptional_behavior
      @ signals_only RuntimeException;
      @*/
    public void work() {
        throw new RuntimeException("Crashed.");
    }

}
```

(1) `if (h>0) {l=1;} else {l=2;}`. This program is *insecure* since an attacker is able to derive from the final value of `l` whether the initial value of `h` was positive or not. Algorithm 1 outputs *false* for the program since the final goals contain formulae `h >= 1` and `h <= 0`, respectively.

(2) `if (h>0) {l=1;} else {l=2;} l=0;`. This program is *secure* since an attacker cannot draw any conclusions about the initial value of `h` since the final value of `l` is always 0. However, Algorithm 1 outputs *false* for the program since the final goals contain formulae `h >= 1` and `h <= 0`, respectively, as in example (1).

(3) `if (h>0) {l=2; h=1;} else {l=2; h=2;}`. This program is *secure* since an attacker cannot draw any conclusions about the initial value of `h` since the final value of `l` is always 2. However, Algorithm 1 outputs *false* for the program since the final goals contain formulae `h >= 1` and `h <= 0`, respectively, as in example (1).

(4) `ExcptFlow a = new ExcptFlow(secret); res = a.secureExceptional(input);` where the class `ExcptFlow` is defined according to Listing 5.7. The program is *secure* since the result of the function call is always *true*. However, Algorithm 1 outputs *false* for the program since the final goals contain formulae `secret = input` and `!secret = input`.

(5) `ExcptFlow a = new ExcptFlow(secret); res = a.insecureExceptional(input);` where the class `ExcptFlow` is defined according to Listing 5.7. The program is *insecure* since the result of the function call is *true* iff the input equals the secret. Algorithm 1 outputs *false* for the program since the final goals contain formulae `secret = input` and `!secret = input`.

The examples show that Algorithm 1 is too restrictive. It would return *true* only for examples like `l=0;` that are very simple concerning their control-flow. We therefore refine the algorithm to Algorithm 2 using the if-then-else join technique.

Algorithm 2 Information flow analysis with joins

```
**Require:** $p$ is a Java program with program variables $\overline{h}$ classified as "high" and $\overline{l}$ classified as "low";
  $P(\ldots)$ is a fresh predicate accepting $\overline{l}$ as inputs.
**Ensure:** Returns true only if $p$ does not contain a flow from a variable in $\overline{h}$ to a variable in $\overline{l}$.
 1: $problem$ ← <\{ try { $p$ } catch (Exception e) {} }\> P($\overline{l}$)
 2: Load $problem$ into KeY and finish symbolic execution
 3: $G$ ← all leafs in the resulting SET containing $P$
 4: Apply, if possible, "if-then-else" join on $P(\ldots)$, if necessary after OneStepSimplifier applications
 5: for all $g \in G$ do
 6:   Apply as many OneStepSimplifier applications on $g$ as possible
 7:   Transform $g$ to the form $\Gamma \Longrightarrow P(\ldots)$ by "negation right" applications
 8:   if $\Gamma$ contains an atomic formula containing a variable in $\overline{h}$ or
        $P(\ldots)$ contains a term comprising a variable in $\overline{h}$ then
 9:     return false
10:   end if
11: end for
12:
13: return true
```

The only change in Algorithm 2 is the addition of line 4, where all branches containing the $P(\ldots)$ formulae are joined. However, this change is strong enough to increase the precision of the results for our "secure" examples, while still classifying "insecure" examples correctly:

(1) Algorithm 2 outputs *false* for the program since the final goals contain formulae `h >= 1` and `h <= 0`, respectively.

(2) Algorithm 2 outputs *true* for the program since the constraints `h >= 1` and `h <= 0` could be eliminated during the join.

(3) Algorithm 2 outputs *true* for the program since the constraints `h >= 1` and `h <= 0` could be eliminated during the join.

(4) Algorithm 2 outputs *true* for the program since the final goals no longer contain atomic formulae `secret = input` and `!secret = input`. Instead, those atoms occur in sub formulae of a disjunction in the antecedent; it is not possible to directly derive information on the initial value of `secret`.

(5) Algorithm 2 outputs *false* for the program since the final goals contain formulae `secret = input` and `!secret = input`. After the join, the succedent of the join node consists of the interesting formula

$$\{\texttt{result} := \mathit{if}\,(\neg(\texttt{secret} \doteq \texttt{input}))\,\mathit{then}\,(\mathit{true})\,\mathit{else}\,(\texttt{result})\}\,P(\texttt{result})$$

which concisely reflects the dependency between `secret` and `result`.

6 Related Work and Conclusion

6.1 Related Work

This thesis may be seen as a contribution to research on the path explosion problem of symbolic execution as well as in relation to other static analysis techniques like abstract interpretation. Furthermore, joins in KeY proof trees might motivate the study of different kinds of information flow analyses based on the KeY system.

The Path Explosion Problem of Symbolic Execution
Existing work on the path explosion problem can be divided into two distinct lines of research. A popular approach, especially in the area of automatic test generation and bug detection, is the pruning of redundant paths in symbolic execution trees by subsumption checking (e.g. [APV06; BCE08; Jaf+12; JMN13; CJM14]). The second line of research applies state merging using if-then-else techniques (e.g. [HSS09; Kuz+12; Sen+14]).

Pruning by Subsumption Checks
When utilizing symbolic execution for the automatic generation of test cases, the repeated exploration of the same program statements is undesirable. A concept employed for subsumption checking that is used, for instance, in [Jaf+12; JMN13; CJM14], is the computation of *interpolants*: Given two formulae A and B (here: two path conditions) s.th. $A \wedge B$ is unsatisfiable, a Craig interpolant Ψ is a formula for which it holds that (1) $A \models \Psi$, (2) $\Psi \wedge B$ is unsatisfiable, and (3) all variables in Ψ are common to A and B. The Tracer system [Jaf+12] systematically underapproximates interpolants and stops the exploration of paths the path conditions of which are subsumed by the interpolants of previously explored paths. Chu et al. [CJM14] compute interpolants and continue the execution even of infeasible paths to obtain better interpolants which they also apply in subsumption checking. The RWset system [BCE08] prunes redundant paths by checking whether the remainder of a particular execution is capable of exploring new behavior. A work by Anand et al. [APV06] especially addresses subsumption in symbolic execution trees containing objects on a heap. They abstract from heap objects (in particular lists and arrays) by summarizing elements sharing common properties using shape analysis techniques and perform subsumption checking for the such abstracted objects. Their approach follows a trend in software model checking which proposes underapproximation based abstractions for the purpose of falsification. Since the intended scenario for our work is software verification, and thus proving the correctness (validity) of properties, underapproximation is not an option.

If-Then-Else State Merging
Existing approaches for joining states in symbolic execution are usually based upon if-then-else techniques similar to the techniques 3.22 and 3.25. Hansen et al. [HSS09] devise an algorithm that symbolically executes a program according to its control flow graph, thereby joining states with the same program counter using an if-then-else construct to create the new symbolic state and a disjunction of path conditions as new path condition for the join node. They use external tools to simplify the generated path conditions. An evaluation of their approach for a set of four examples shows mixed results: While joining is beneficial for three out of four examples, the execution time explodes for the fourth one. Kuznetsov et al. [Kuz+12] tackle this problem by investigating how state joining based on if-then-else constructs can be rendered “practical”: Their approach aims to automatically find an advantageous balance between exploring fewer complex (i.e., merged) states vs. more simpler (i.e. unmerged) states. Based on heuristics, they only merge states when this promises to reduce the exploration time. The central idea of the MultiSE system by Sen et al. [Sen+14] is the concept of so-called “value summaries” that map program variables to sets of guarded symbolic expressions. Value summaries are comparable to the if-then-else expressions of KeY, but without the else part; instead of this, another guarded expression is added to the summary. MultiSE explicitly represents the data structure induced by symbolic execution as a DAG. Join points are identified dynamically, the system does not rely on an explicit CFG of the program. For efficiency reasons, path conditions and guards in value summaries are implemented using binary decision diagrams. Experiments show significant speedup of this technique compared to standard (dynamic) SE in their experiments. Compared to our framework, all these systems / approaches

incorporate one chosen join technique based on an if-then-else construct, while our work permits a huge set of overapproximation techniques, also covering those based on if-then-else.

Abstract Interpretation

Abstract interpretation is a static analysis technique introduced in the 1970s by Cousot and Cousot [CC77]. The analysis is parametric in the chosen abstraction and proceeds fully automatically by the means of a fixpoint iteration for loops. State merging, following the CFG of the analyzed program, is inherent to the system. With ASTRÉE [Cou+05], there exists a mature system for the runtime error detection in C programs based on abstract interpretation which has been successfully applied in industry. Numerous abstract domains have been implemented for ASTRÉE. The system is sound and therefore necessarily incomplete, that is it may yield false alarms. Bubel et al. [BHW09] propose a dynamic logic with abstraction for the KeY calculus, mainly to facilitate the fully automatic discovery of loop invariants. The idea is based on the employment of a fixpoint algorithm in the spirit of abstract interpretation until an invariant is found. Their framework is based on partially interpreted constant symbols for each abstract value, accompanied by dedicated calculus rules for interpreting them. They use a join rule for while loops in side branches for the computation of invariants. Our work generalizes the abstraction and join aspects of Bubel et al. by devising a parametric framework with a join rule for arbitrary program statements. However, we do not cover the automatic discovery of loop invariants by fixpoint iteration. In contrast to abstract interpretation, our work supports fully precise (if-then-else) as well as abstraction-based state merging.

Information Flow Security

The problem of information flow security is the enforcement or assessment of information flow policies restricting the flow of information between locations in a program. Traditional techniques for the (language-based) enforcement of such policies are based on security type systems (see [SM06] for an overview). Those systems usually suffer from a lack of precision, that is they may classify many secure programs as insecure; however, they can reach full automation. The previously mentioned work by Bubel et al. [BHW09] tracks dependencies between variables to assess the validity of non-interference properties. Their employment of abstraction promises a higher degree of automation, but is likely to decrease the precision compared to deductive systems without abstraction (like, e.g., [BDR04; DHS05; JH14]). However, the approach is more precise than typical security type systems. A different line of research [SRK06; HKS06] establishes information flow security by analyzing "path conditions" computed from the *Program Dependence Graph (PDG)* [FOW84; OO84] of the investigated program. A path condition in this context is a generalization of the corresponding notion in symbolic execution: It is computed between two arbitrary basic blocks of a program by accumulating control and data conditions on all possible paths between the blocks. In symbolic execution, a path condition is computed from the root node on, and mostly captures control dependencies. Snelting et al. [SRK06] establish a straightforward theorem connecting PDG*s* to non-interference; Hammer et al. [HKS06] apply this result to medium-sized Java programs in a fully automatic approach. Generalized path conditions are precise necessary conditions for information flow between two program points. Our simple application to information flow security (→ Section 5.2) also takes path conditions (in the sense of SE) into account; it might be interesting to study the relation between SE path conditions and the generalized concept based on PDG*s* by Hammer et al. Definitely, KeY proof trees come closer to the CFG and therefore also to the PDG of programs by incorporating our join techniques.

6.2 Conclusion and Future Work

Conclusion

We extended symbolic execution by a parametric join rule which is sound for all join operations satisfying, besides three basic semilattice properties, an additional correctness property and a simple constraint on the generated path conditions. Using our join rule, symbolic execution trees are rendered into directed acyclic graphs. Thereby, we contribute to closing the gap between symbolic execution and abstract interpretation and to solving the path explosion problem of symbolic execution. Four example join rules based upon the general lattice framework, including two if-then-else based methods and one method based on abstract domains, are theoretically defined and practically implemented for the state-of-the-art deductive verification system KeY. We showed that the employment of join rules indeed reduces the number of nodes and branches in KeY proof trees for a set of example programs. Furthermore, we demonstrated that branch joining could be beneficial for analysis techniques in the area of information flow security. We are confident that our work can support making symbolic execution more efficient, and give rise to new analysis methods based on symbolic execution DAG*s* with suitable join techniques. In addition to evaluating our join techniques on larger case studies, we identified four lines of future work for extending and improving our system.

Performance of the Implementation

We expect that there is a significant potential for improving the performance of our implementation. On the theoretical side, it might be rewarding to investigate whether Craig-Interpolants are employable for generating simplified distinguishing formulae in if-then-else constructs. The generation of those invariants is only easy for quantifier-free formulae [McM05; Bri+11]. Preconditions in sequents often contain quantifiers arising from method specifications, while "pure" path conditions usually are quantifier-free. If applicable, Craig-Interpolants could serve as efficiently computable and concise distinguishing formulae. Other implemented optimization methods, e.g. for the computation of optimized path conditions, also are likely to be improvable in terms of effectiveness and efficiency. In the case of abstraction-based join methods, the performance of validity checks for defining axioms of abstract domain values might be improvable be outsourcing the evaluation to a third-party Satisfiability Modulo Theories (SMT) solver. The last point concerning performance addresses the non-termination problem for the "multiply" example in Section 5.1. The question of why the results of the automatic strategy and the (automatic) macro differ and why this difference actually caused the non-determination issue should be studied closer to fix the problem and possibly generalize the gained experience to different scenarios.

Abstraction Framework

To catch up with abstract interpretation systems like ASTRÉE [Cou+05], many more abstract domains will have to be realized for KeY. For some of those, it might be necessary to extend the flexibility of our abstraction framework which for now is built to cope with quite small, static domains. For some applications, the implementation of abstractions specific to certain heap structures like lists or arrays (cf., e.g., [SRW02; APV06]) could be beneficial; so far, we only apply abstractions for integers or booleans to the elements of suitably uniformly shaped heaps. One of the strengths of abstract interpretation systems, the fixpoint iteration for unbounded loops, would, when implemented as a set of new rules or strategies for the KeY system, push forward the abstraction framework very much and help to further close the gap between symbolic execution and abstract interpretation. The work by Bubel et al. [BHW09] could serve as a guideline here.

Usability

The major starting point for making our implemented framework more usable is the creation of new (or the extension of existing) automatic strategies that are aware of join rules and join points. A very first and simple idea is the repeated execution of our macro (→ Section 4.4). However, a solution integrated into the main automatic strategies that identifies join points dynamically, such as [Sen+14], would be desirable. Such a solution could also take into account metrics for assessing the reward of a join operation, as in the approach by [Kuz+12]. A more technical improvement concerns the integration of the logical weakening proof goals into proofs with joins. At the moment, we append two nodes to partner nodes participating in a join operation (→ Section 4.3): One node linking to the join node, and one containing the logical weakening formula. As an alternative, we plan to add this side proof goal as a "contract target" to the meta information of a proof, in the fashion of the current treatment of method contracts. Thus, the size of a proof containing join rule applications would not be blown up by the proofs corresponding to logical weakening; instead, the validity of the main proof goal would be marked as relative to the validity of the logical weakening formulae.

New Analysis Techniques

We assume that the representation of symbolic execution as a DAG, being closer to the actual control flow of the considered programs than the tree representation, opens up symbolic execution to different kinds of static analysis techniques. In Section 5.2, we briefly studied the area of information flow analysis as one potential candidate for a class of such techniques. A closer investigation of the benefits of branch joining for an existing, provably sound information flow analysis based on symbolic execution trees would be interesting. Furthermore, we endorse the study of approaches based on generalized path conditions [SRK06; HKS06] with respect to their application on SE DAG*s*.

Large Figures

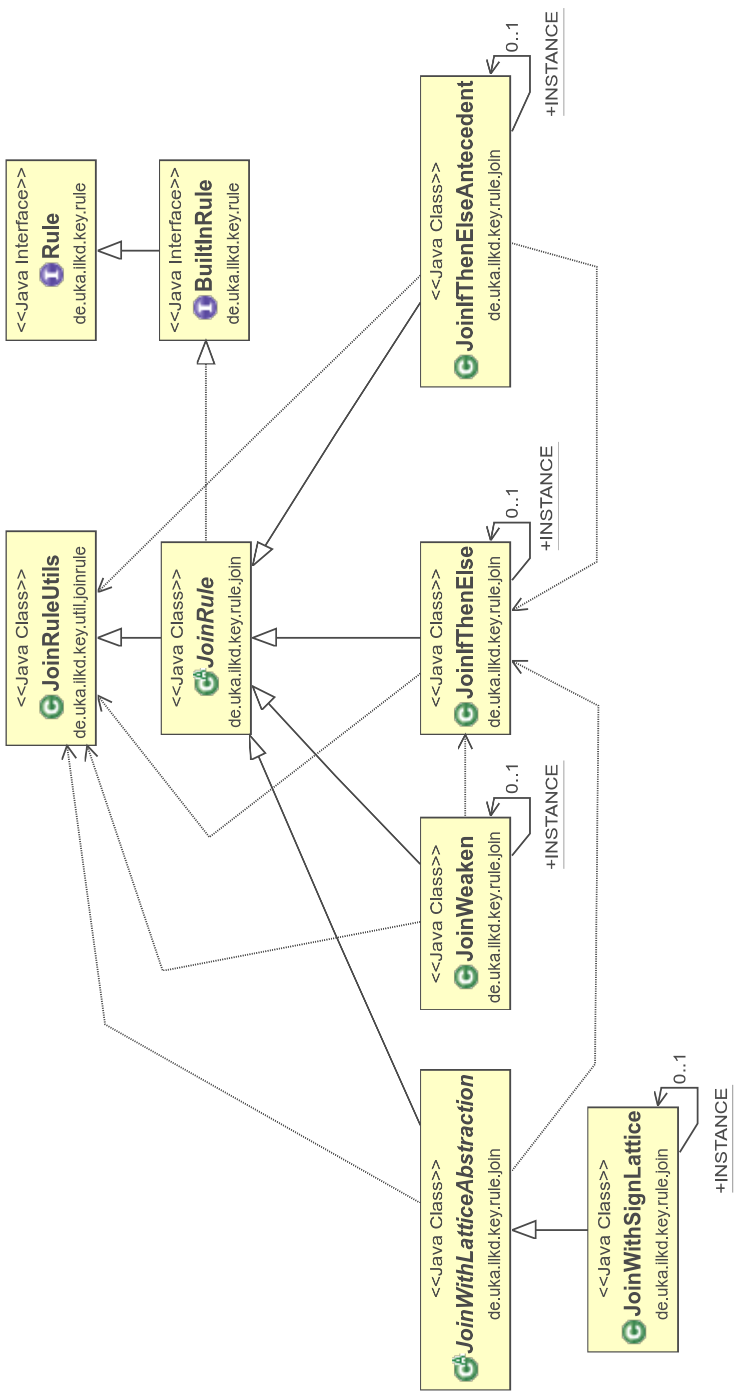


Figure 6.1: Overview of join classes

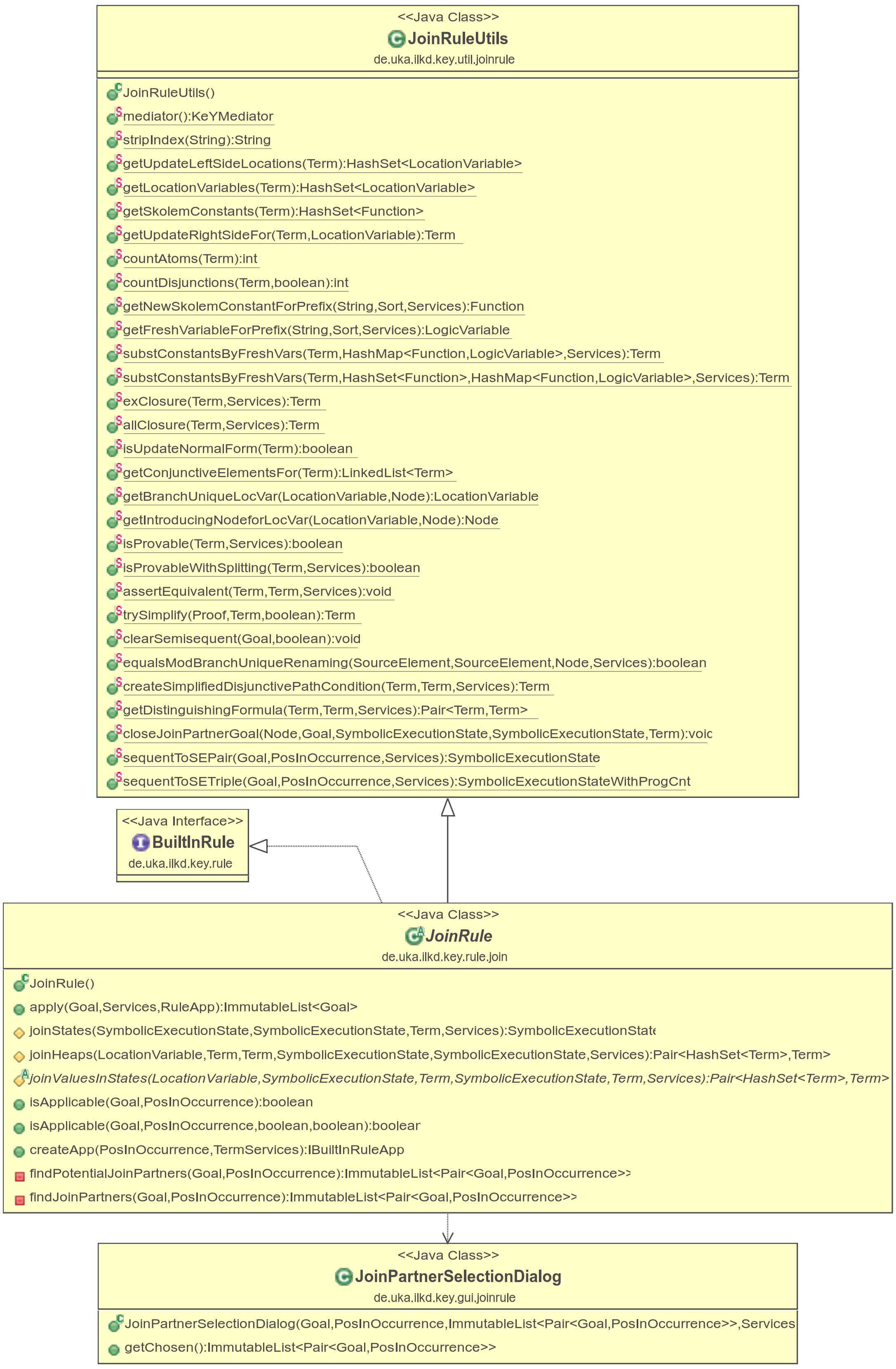
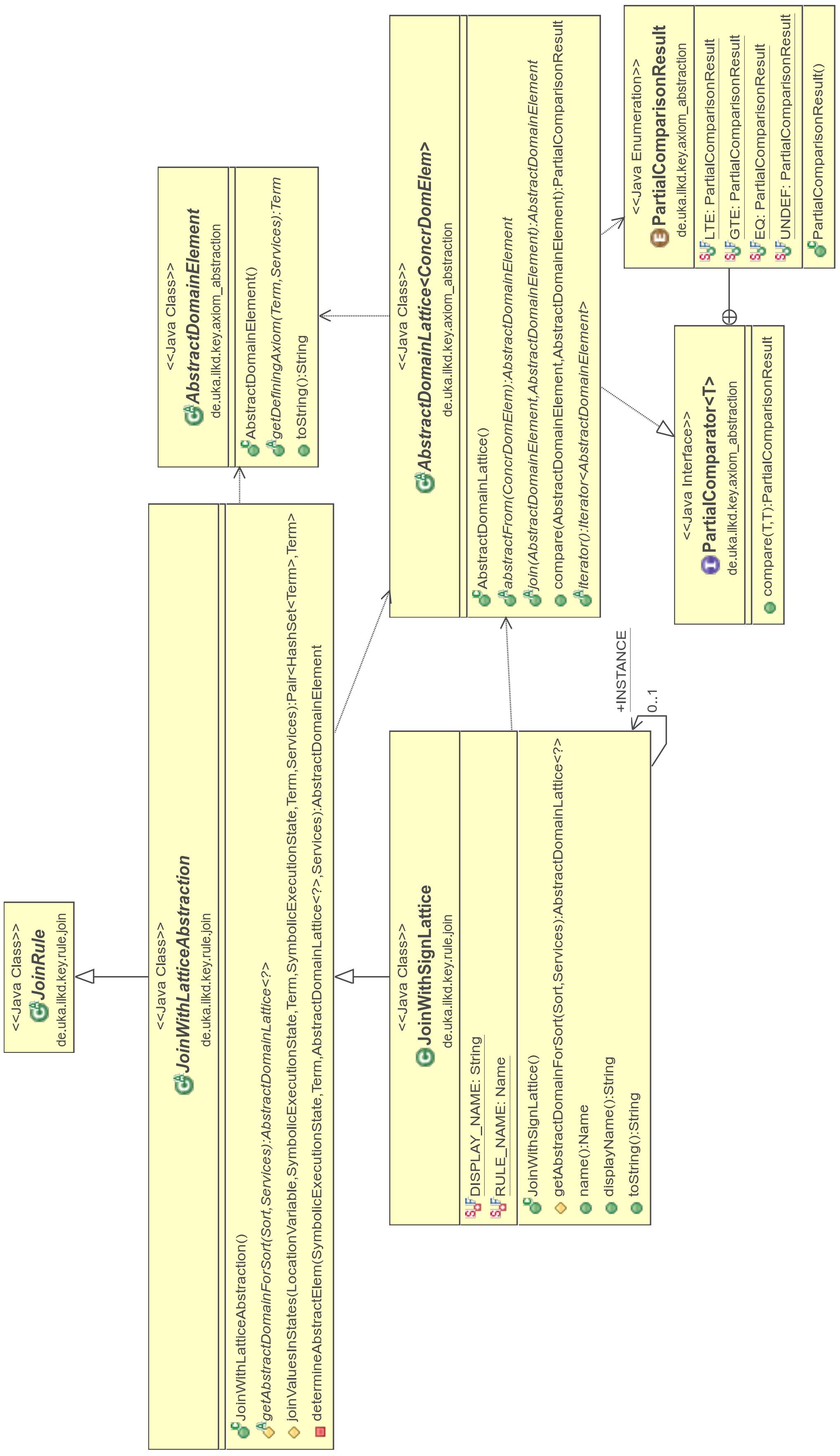

<<Java Class>>
JoinRuleUtils
de.uka.ilkd.key.util.joinrule
JoinRuleUtils()
mediator():KeYMediator
stripIndex(String):String
getUpdateLeftSideLocations(Term):HashSet<LocationVariable>
getLocationVariables(Term):HashSet<LocationVariable>
getSkolemConstants(Term):HashSet<Function>
getUpdateRightSideFor(Term,LocationVariable):Term
countAtoms(Term):int
countDisjunctions(Term,boolean):int
getNewSkolemConstantForPrefix(String,Sort,Services):Function
getFreshVariableForPrefix(String,Sort,Services):LogicVariable
substConstantsByFreshVars(Term,HashMap<Function,LogicVariable>,Services):Term
substConstantsByFreshVars(Term,HashSet<Function>,HashMap<Function,LogicVariable>,Services):Term
exClosure(Term,Services):Term
allClosure(Term,Services):Term
isUpdateNormalForm(Term):boolean
getConjunctiveElementsFor(Term):LinkedList<Term>
getBranchUniqueLocVar(LocationVariable,Node):LocationVariable
getIntroducingNodeforLocVar(LocationVariable,Node):Node
isProvable(Term,Services):boolean
isProvableWithSplitting(Term,Services):boolean
assertEquivalent(Term,Term,Services):void
trySimplify(Proof,Term,boolean):Term
clearSemisequent(Goal,boolean):void
equalsModBranchUniqueRenaming(SourceElement,SourceElement,Node,Services):boolean
createSimplifiedDisjunctivePathCondition(Term,Term,Services):Term
getDistinguishingFormula(Term,Term,Services):Pair<Term,Term>
closeJoinPartnerGoal(Node,Goal,SymbolicExecutionState,SymbolicExecutionState,Term):voic
sequentToSEPair(Goal,PosInOccurrence,Services):SymbolicExecutionState
sequentToSETriple(Goal,PosInOccurrence,Services):SymbolicExecutionStateWithProgCnt
<<Java Interface>>
BuiltInRule
de.uka.ilkd.key.rule
<<Java Class>>
JoinRule
de.uka.ilkd.key.rule.join
JoinRule()
apply(Goal,Services,RuleApp):ImmutableList<Goal>
joinStates(SymbolicExecutionState,SymbolicExecutionState,Term,Services):SymbolicExecutionStat
joinHeaps(LocationVariable,Term,Term,SymbolicExecutionState,SymbolicExecutionState,Services):Pair<HashSet<Term>,Term>
joinValuesInStates(LocationVariable,SymbolicExecutionState,Term,SymbolicExecutionState,Term,Services):Pair<HashSet<Term>,Term>
isApplicable(Goal,PosInOccurrence):boolean
isApplicable(Goal,PosInOccurrence,boolean,boolean):boolear
createApp(PosInOccurrence,TermServices):IBuiltInRuleApp
findPotentialJoinPartners(Goal,PosInOccurrence):ImmutableList<Pair<Goal,PosInOccurrence>>
findJoinPartners(Goal,PosInOccurrence):ImmutableList<Pair<Goal,PosInOccurrence>>
<<Java Class>>
JoinPartnerSelectionDialog
de.uka.ilkd.key.gui.joinrule
JoinPartnerSelectionDialog(Goal,PosInOccurrence,ImmutableList<Pair<Goal,PosInOccurrence>>,Services
getChosen():ImmutableList<Pair<Goal,PosInOccurrence>>

Figure 6.2: Join rule implementation



<<Java Class>>
JoinRule
de.uka.ilkd.key.rule.join
<<Java Class>>
JoinWithLatticeAbstraction
de.uka.ilkd.key.rule.join
JoinWithLatticeAbstraction()
getAbstractDomainForSort(Sort,Services):AbstractDomainLattice<?>
joinValuesInStates(LocationVariable,SymbolicExecutionState,Term,SymbolicExecutionState,Term,Services):Pair<HashSet<Term>,Term>
determineAbstractElem(SymbolicExecutionState,Term,AbstractDomainLattice<?>,Services):AbstractDomainElement
<<Java Class>>
AbstractDomainElement
de.uka.ilkd.key.axiom_abstraction
AbstractDomainElement()
getDefiningAxiom(Term,Services):Term
toString():String
<<Java Class>>
JoinWithSignLattice
de.uka.ilkd.key.rule.join
DISPLAY_NAME: String
RULE_NAME: Name
JoinWithSignLattice()
getAbstractDomainForSort(Sort,Services):AbstractDomainLattice<?>
name():Name
displayName():String
toString():String
+INSTANCE
0..1
<<Java Class>>
AbstractDomainLattice<ConcrDomElem>
de.uka.ilkd.key.axiom_abstraction
AbstractDomainLattice()
abstractFrom(ConcrDomElem):AbstractDomainElement
join(AbstractDomainElement,AbstractDomainElement):AbstractDomainElement
compare(AbstractDomainElement,AbstractDomainElement):PartialComparisonResult
iterator():Iterator<AbstractDomainElement>
<<Java Interface>>
PartialComparator<T>
de.uka.ilkd.key.axiom_abstraction
compare(T,T):PartialComparisonResult
<<Java Enumeration>>
PartialComparisonResult
de.uka.ilkd.key.axiom_abstraction
LTE: PartialComparisonResult
GTE: PartialComparisonResult
EQ: PartialComparisonResult
UNDEF: PartialComparisonResult
PartialComparisonResult()

Figure 6.3: Lattice abstraction framework and rules

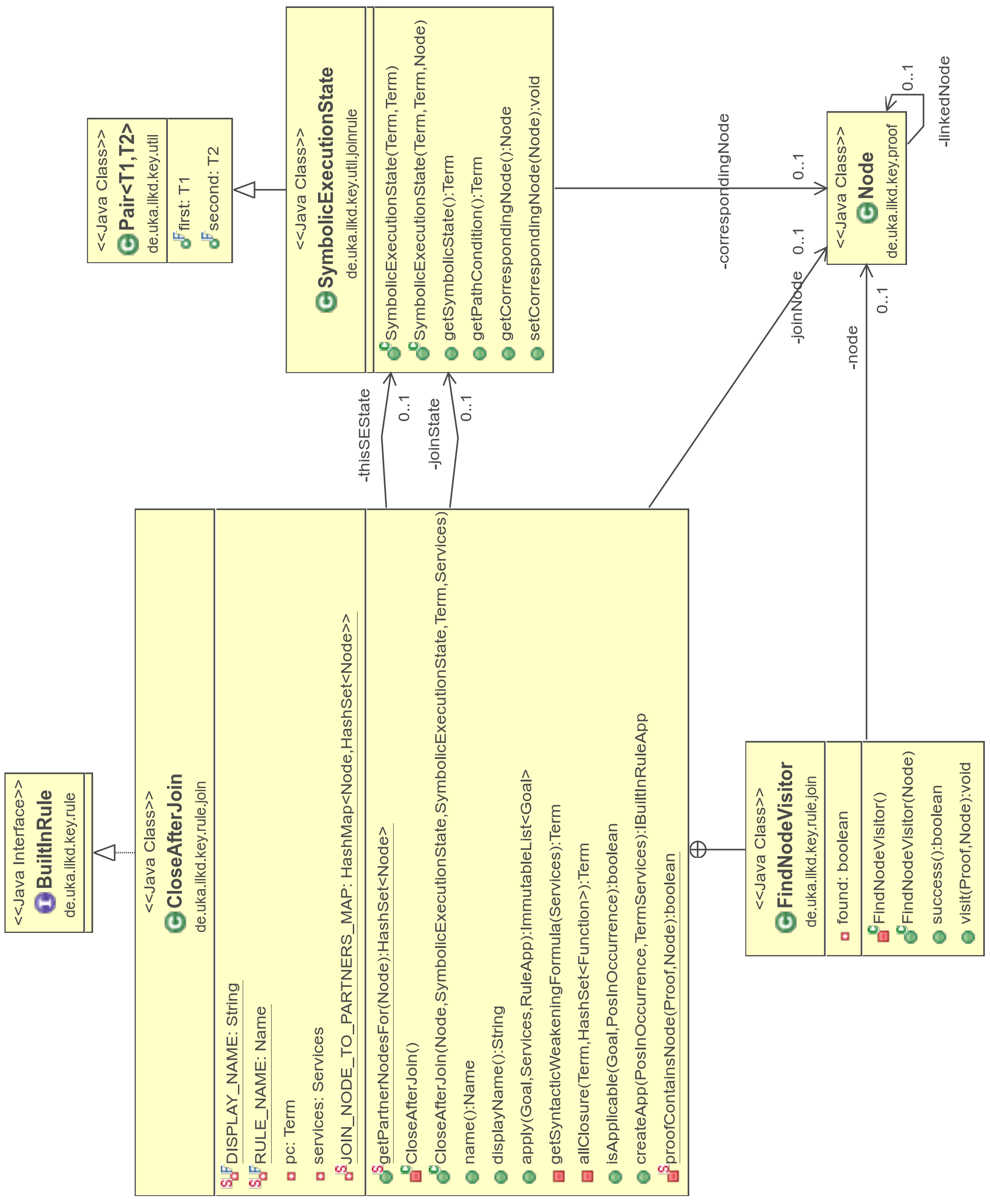

<<Java Interface>>
BuiltInRule
de.uka.ilkd.key.rule
<<Java Class>>
CloseAfterJoin
de.uka.ilkd.key.rule.join
DISPLAY_NAME: String
RULE_NAME: Name
pc: Term
services: Services
JOIN_NODE_TO_PARTNERS_MAP: HashMap<Node,HashSet<Node>>
getPartnerNodesFor(Node):HashSet<Node>
CloseAfterJoin()
CloseAfterJoin(Node,SymbolicExecutionState,SymbolicExecutionState,Term,Services)
name():Name
displayName():String
apply(Goal,Services,RuleApp):ImmutableList<Goal>
getSyntacticWeakeningFormula(Services):Term
allClosure(Term,HashSet<Function>):Term
isApplicable(Goal,PosInOccurrence):boolean
createApp(PosInOccurrence,TermServices):IBuiltInRuleApp
proofContainsNode(Proof,Node):boolean
<<Java Class>>
Pair<T1,T2>
de.uka.ilkd.key.util
first: T1
second: T2
<<Java Class>>
SymbolicExecutionState
de.uka.ilkd.key.util.joinrule
SymbolicExecutionState(Term,Term)
SymbolicExecutionState(Term,Term,Node)
getSymbolicState():Term
getPathCondition():Term
getCorrespondingNode():Node
setCorrespondingNode(Node):void
-thisSEState
0..1
-joinState
0..1
-correspondingNode
0..1
-joinNode
0..1
-node
0..1
<<Java Class>>
Node
de.uka.ilkd.key.proof
0..1
-linkedNode
<<Java Class>>
FindNodeVisitor
de.uka.ilkd.key.rule.join
found: boolean
FindNodeVisitor()
FindNodeVisitor(Node)
success():boolean
visit(Proof,Node):void

Figure 6.4: CloseAfterJoin rule

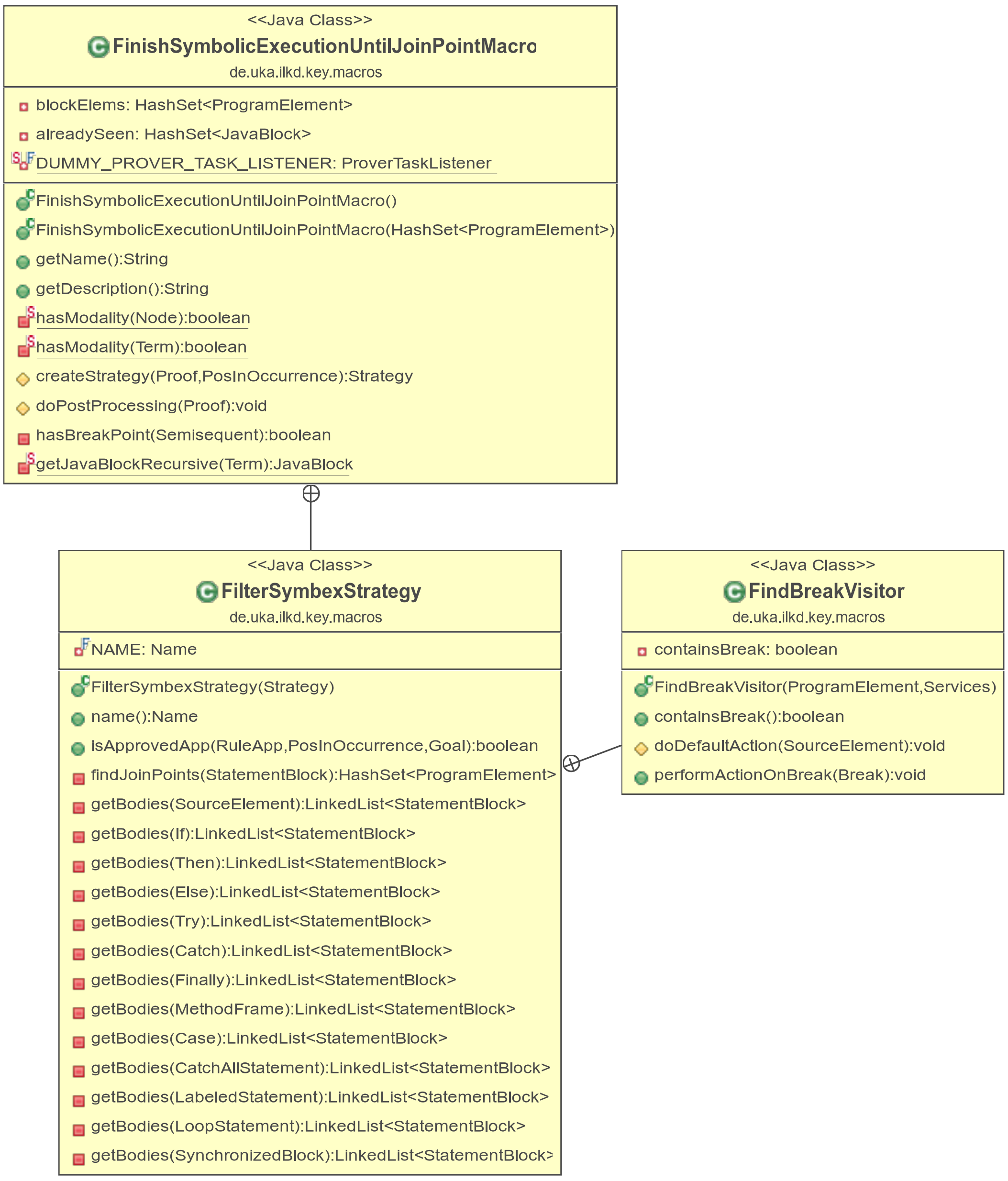

<<Java Class>>
FinishSymbolicExecutionUntilJoinPointMacro
de.uka.ilkd.key.macros
blockElems: HashSet<ProgramElement>
alreadySeen: HashSet<JavaBlock>
DUMMY_PROVER_TASK_LISTENER: ProverTaskListener
FinishSymbolicExecutionUntilJoinPointMacro()
FinishSymbolicExecutionUntilJoinPointMacro(HashSet<ProgramElement>)
getName():String
getDescription():String
hasModality(Node):boolean
hasModality(Term):boolean
createStrategy(Proof,PosInOccurrence):Strategy
doPostProcessing(Proof):void
hasBreakPoint(Semisequent):boolean
getJavaBlockRecursive(Term):JavaBlock
<<Java Class>>
FilterSymbexStrategy
de.uka.ilkd.key.macros
NAME: Name
FilterSymbexStrategy(Strategy)
name():Name
isApprovedApp(RuleApp,PosInOccurrence,Goal):boolean
findJoinPoints(StatementBlock):HashSet<ProgramElement>
getBodies(SourceElement):LinkedList<StatementBlock>
getBodies(If):LinkedList<StatementBlock>
getBodies(Then):LinkedList<StatementBlock>
getBodies(Else):LinkedList<StatementBlock>
getBodies(Try):LinkedList<StatementBlock>
getBodies(Catch):LinkedList<StatementBlock>
getBodies(Finally):LinkedList<StatementBlock>
getBodies(MethodFrame):LinkedList<StatementBlock>
getBodies(Case):LinkedList<StatementBlock>
getBodies(CatchAllStatement):LinkedList<StatementBlock>
getBodies(LabeledStatement):LinkedList<StatementBlock>
getBodies(LoopStatement):LinkedList<StatementBlock>
getBodies(SynchronizedBlock):LinkedList<StatementBlock>
<<Java Class>>
FindBreakVisitor
de.uka.ilkd.key.macros
containsBreak: boolean
FindBreakVisitor(ProgramElement,Services)
containsBreak():boolean
doDefaultAction(SourceElement):void
performActionOnBreak(Break):void

Figure 6.5: FinishSymbolicExecutionUntilJoinPointMacro

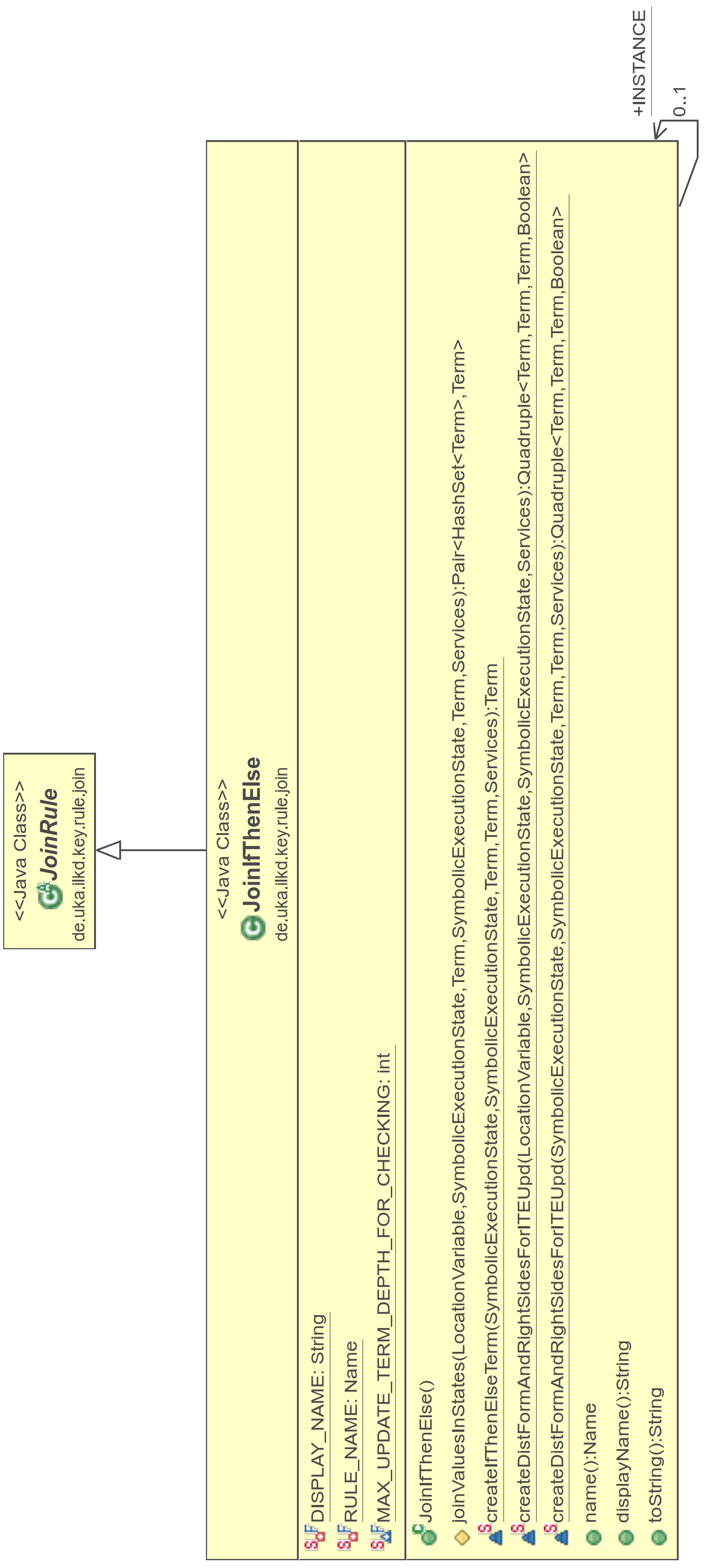
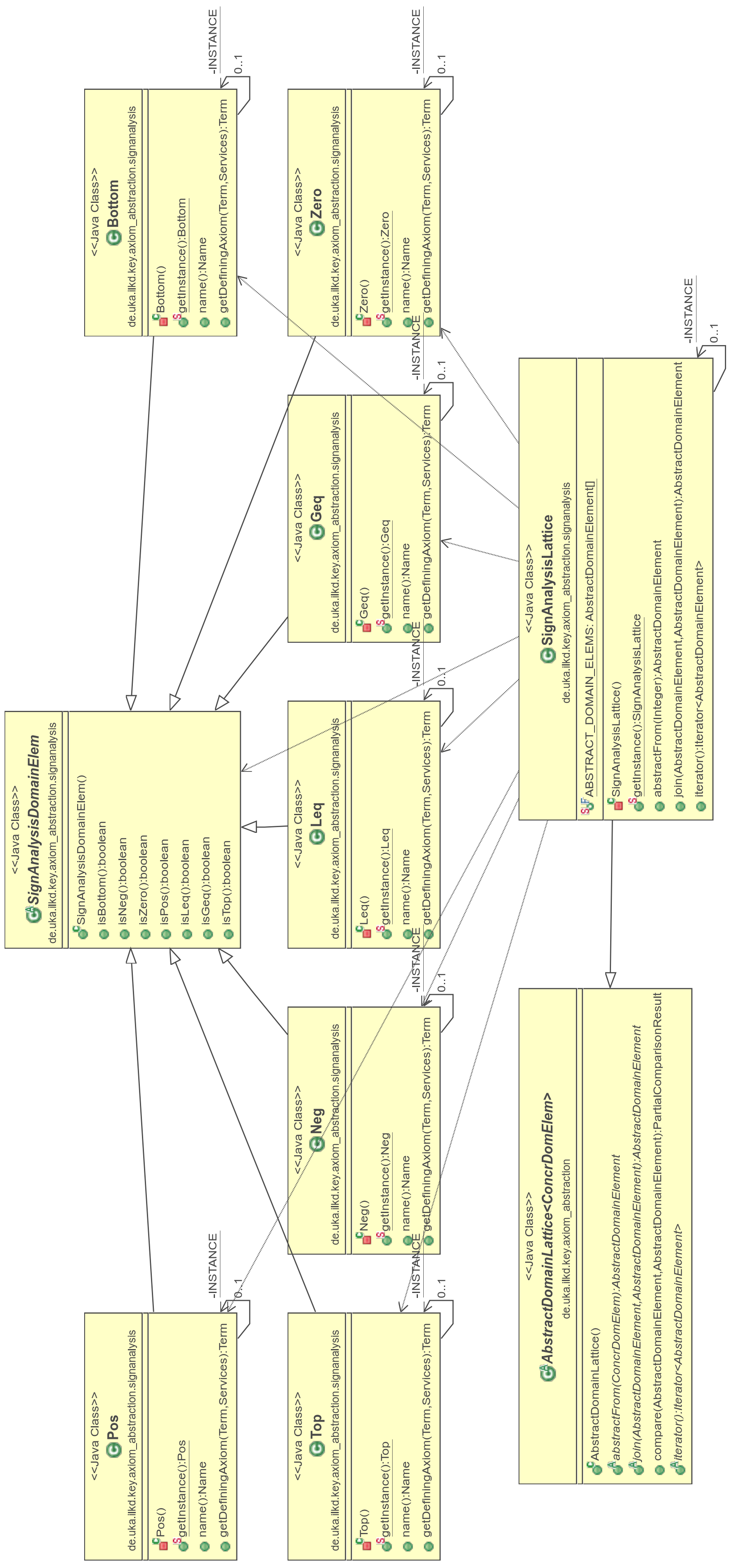
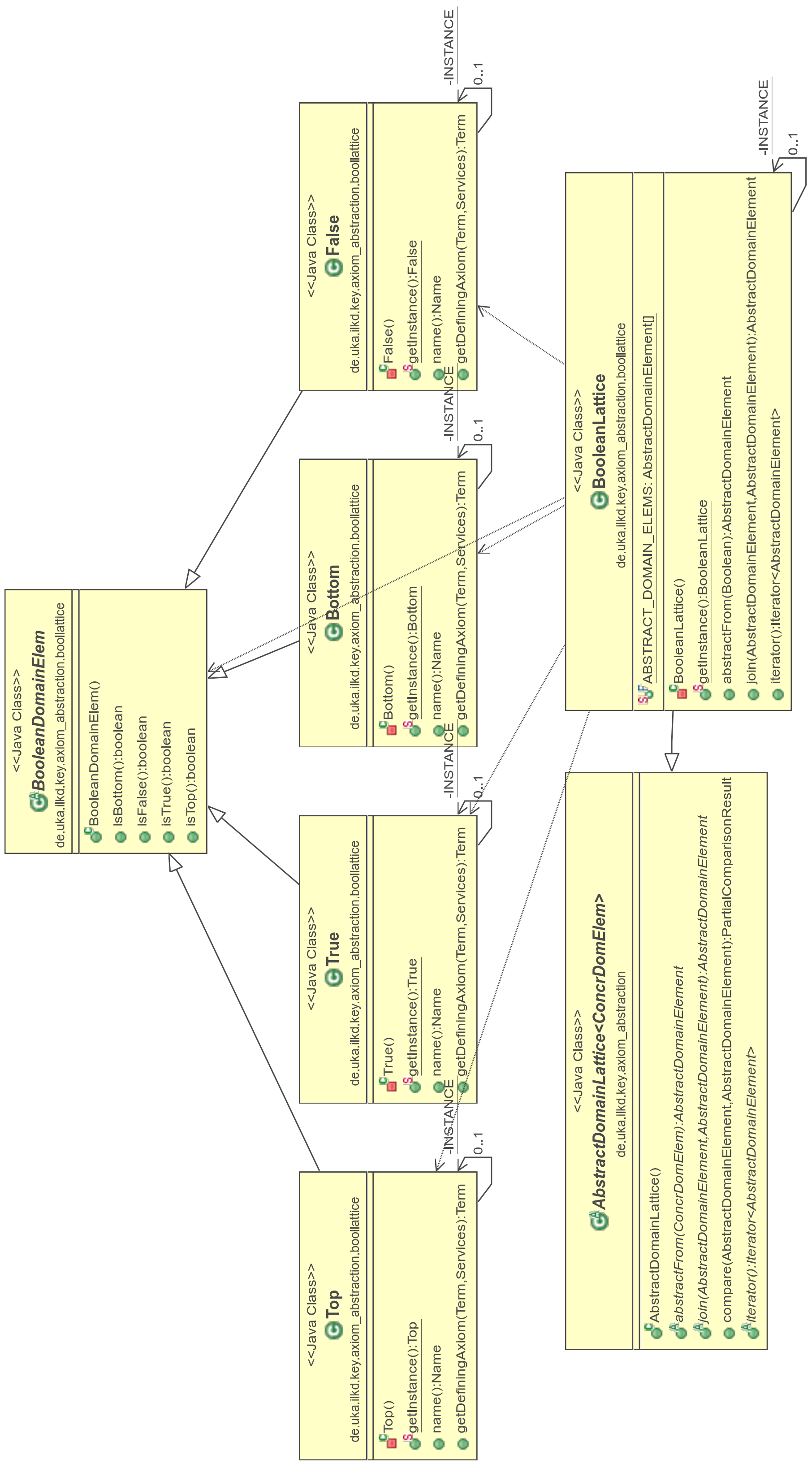

<<Java Class>>
JoinRule
de.uka.ilkd.key.rule.join
<<Java Class>>
JoinIfThenElse
de.uka.ilkd.key.rule.join
DISPLAY_NAME: String
RULE_NAME: Name
MAX_UPDATE_TERM_DEPTH_FOR_CHECKING: int
JoinIfThenElse()
joinValuesInStates(LocationVariable,SymbolicExecutionState,Term,SymbolicExecutionState,Term,Services):Pair<HashSet<Term>,Term>
createIfThenElseTerm(SymbolicExecutionState,SymbolicExecutionState,Term,Term,Services):Term
createDistFormAndRightSidesForITEUpd(LocationVariable,SymbolicExecutionState,SymbolicExecutionState,Services):Quadruple<Term,Term,Term,Boolean>
createDistFormAndRightSidesForITEUpd(SymbolicExecutionState,SymbolicExecutionState,Term,Term,Services):Quadruple<Term,Term,Term,Boolean>
name():Name
displayName():String
toString():String
+INSTANCE
0..1

Figure 6.6: If-then-else rule implementation



<<Java Class>>
Bottom
de.uka.ilkd.key.axiom_abstraction.signanalysis
Bottom()
getInstance():Bottom
name():Name
getDefiningAxiom(Term,Services):Term
-INSTANCE
0..1
<<Java Class>>
Zero
de.uka.ilkd.key.axiom_abstraction.signanalysis
Zero()
getInstance():Zero
name():Name
getDefiningAxiom(Term,Services):Term
<<Java Class>>
Geq
de.uka.ilkd.key.axiom_abstraction.signanalysis
Geq()
getInstance():Geq
name():Name
getDefiningAxiom(Term,Services):Term
<<Java Class>>
Leq
de.uka.ilkd.key.axiom_abstraction.signanalysis
Leq()
getInstance():Leq
name():Name
getDefiningAxiom(Term,Services):Term
<<Java Class>>
Neg
de.uka.ilkd.key.axiom_abstraction.signanalysis
Neg()
getInstance():Neg
name():Name
getDefiningAxiom(Term,Services):Term
<<Java Class>>
Pos
de.uka.ilkd.key.axiom_abstraction.signanalysis
Pos()
getInstance():Pos
name():Name
getDefiningAxiom(Term,Services):Term
<<Java Class>>
Top
de.uka.ilkd.key.axiom_abstraction.signanalysis
Top()
getInstance():Top
name():Name
getDefiningAxiom(Term,Services):Term
<<Java Class>>
SignAnalysisDomainElem
de.uka.ilkd.key.axiom_abstraction.signanalysis
SignAnalysisDomainElem()
isBottom():boolean
isNeg():boolean
isZero():boolean
isPos():boolean
isLeq():boolean
isGeq():boolean
isTop():boolean
<<Java Class>>
SignAnalysisLattice
de.uka.ilkd.key.axiom_abstraction.signanalysis
ABSTRACT_DOMAIN_ELEMS: AbstractDomainElement[]
SignAnalysisLattice()
getInstance():SignAnalysisLattice
abstractFrom(Integer):AbstractDomainElement
join(AbstractDomainElement,AbstractDomainElement):AbstractDomainElement
iterator():Iterator<AbstractDomainElement>
<<Java Class>>
AbstractDomainLattice<ConcrDomElem>
de.uka.ilkd.key.axiom_abstraction
AbstractDomainLattice()
abstractFrom(ConcrDomElem):AbstractDomainElement
join(AbstractDomainElement,AbstractDomainElement):AbstractDomainElement
compare(AbstractDomainElement,AbstractDomainElement):PartialComparisonResult
iterator():Iterator<AbstractDomainElement>

Figure 6.7: Sign Analysis Lattice



<<Java Class>>
BooleanDomainElem
de.uka.ilkd.key.axiom_abstraction.boollattice
BooleanDomainElem()
isBottom():boolean
isFalse():boolean
isTrue():boolean
isTop():boolean
<<Java Class>>
False
de.uka.ilkd.key.axiom_abstraction.boollattice
False()
getInstance():False
name():Name
getDefiningAxiom(Term,Services):Term
<<Java Class>>
Bottom
de.uka.ilkd.key.axiom_abstraction.boollattice
Bottom()
getInstance():Bottom
name():Name
getDefiningAxiom(Term,Services):Term
<<Java Class>>
True
de.uka.ilkd.key.axiom_abstraction.boollattice
True()
getInstance():True
name():Name
getDefiningAxiom(Term,Services):Term
<<Java Class>>
Top
de.uka.ilkd.key.axiom_abstraction.boollattice
Top()
getInstance():Top
name():Name
getDefiningAxiom(Term,Services):Term
<<Java Class>>
BooleanLattice
de.uka.ilkd.key.axiom_abstraction.boollattice
ABSTRACT_DOMAIN_ELEMS: AbstractDomainElement[]
BooleanLattice()
getInstance():BooleanLattice
abstractFrom(Boolean):AbstractDomainElement
join(AbstractDomainElement,AbstractDomainElement):AbstractDomainElement
iterator():Iterator<AbstractDomainElement>
<<Java Class>>
AbstractDomainLattice<ConcrDomElem>
de.uka.ilkd.key.axiom_abstraction
AbstractDomainLattice()
abstractFrom(ConcrDomElem):AbstractDomainElement
join(AbstractDomainElement,AbstractDomainElement):AbstractDomainElement
compare(AbstractDomainElement,AbstractDomainElement):PartialComparisonResult
iterator():Iterator<AbstractDomainElement>
-INSTANCE
0..1
-INSTANCE
0..1
-INSTANCE
0..1
-INSTANCE
0..1
-INSTANCE
0..1

Figure 6.8: Boolean Lattice

List of Acronyms

CFG	Control Flow Graph
DAG	Directed Acyclic Graph
DL	Dynamic Logic
PDG	Program Dependence Graph
SE	Symbolic Execution
SET	Symbolic Execution Tree
SMT	Satisfiability Modulo Theories

List of Figures

List of Tables

List of Algorithms

List of Listings

Bibliography

[All70] Frances E. Allen. "Control Flow Analysis". In: *Proceedings of a Symposium on Compiler Optimization*. New York, NY, USA: ACM, 1970, pp. 1–19.

[APV06] Saswat Anand, Corina S. Pasareanu, and Willem Visser. "Symbolic Execution with Abstract Subsumption Checking". In: *Model Checking Software*. Ed. by Antti Valmari. Lecture Notes in Computer Science 3925. Springer Berlin Heidelberg, Jan. 2006, pp. 163–181. ISBN: 978-3-540-33102-5, 978-3-540-33103-2.

[BCE08] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. "RWset: Attacking path explosion in constraint-based test generation". In: *IN TACAS'08: INTERNATIONAL CONFERENCE ON TOOLS AND ALGORITHMS FOR THE CONSTRUCTIONS AND ANALYSIS OF SYSTEMS*. 2008.

[BDR04] G. Barthe, P.R. D'Argenio, and T. Rezk. "Secure Information Flow by Self-Composition". In: *17th IEEE Computer Security Foundations Workshop, 2004. Proceedings*. June 2004, pp. 100–114.

[BEL75] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. "SELECT – A Formal System for Testing and Debugging Programs by Symbolic Execution". In: *Proceedings of the International Conference on Reliable Software*. New York, NY, USA: ACM, 1975, pp. 234–245.

[Ben11] Benjamin Weiß. "Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction". PhD thesis. Karlsruhe: Karlsruhe Institute of Technology, 2011.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*. LNCS 4334. Berlin, Heidelberg: Springer-Verlag, 2007. ISBN: 3-540-68977-X, 978-3-540-68977-5.

[BHW09] Richard Bubel, Reiner Hähnle, and Benjamin Weiß. "Abstract Interpretation of Symbolic Execution with Explicit State Updates". In: *Formal Methods for Components and Objects*. Ed. by Frank S. de Boer, Marcello M. Bonsangue, and Eric Madelaine. Lecture Notes in Computer Science 5751. Springer Berlin Heidelberg, Jan. 2009, pp. 247–277. ISBN: 978-3-642-04166-2, 978-3-642-04167-9.

[Bri+11] Angelo Brillout et al. "Beyond Quantifier-Free Interpolation in Extensions of Presburger Arithmetic". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Ranjit Jhala and David Schmidt. Lecture Notes in Computer Science 6538. Springer Berlin Heidelberg, 2011, pp. 88–102. ISBN: 978-3-642-18274-7, 978-3-642-18275-4.

[BS08] J. Burnim and K. Sen. "Heuristics for Scalable Dynamic Test Generation". In: *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. ASE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 443–446. ISBN: 978-1-4244-2187-9.

[Bur74] Rodney Matineau Burstall. "Program Proving as Hand Simulation with a Little Induction". In: *Information Processing*. Elsevier/North-Holland, 1974, pp. 308–312.

[CC77] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '77. New York, NY, USA: ACM, 1977, pp. 238–252.

[CJM14] Duc-Hiep Chu, Joxan Jaffar, and Vijayaraghavan Murali. "Lazy Symbolic Execution for Enhanced Learning". In: *Runtime Verification*. Ed. by Borzoo Bonakdarpour and Scott A. Smolka. Lecture Notes in Computer Science 8734. Springer International Publishing, Sept. 2014, pp. 323–339. ISBN: 978-3-319-11163-6, 978-3-319-11164-3.

[Cou+05] Patrick Cousot et al. "The ASTREÉ Analyzer". In: *Programming Languages and Systems*. Ed. by Mooly Sagiv. Lecture Notes in Computer Science 3444. Springer Berlin Heidelberg, 2005, pp. 21–30. ISBN: 978-3-540-25435-5, 978-3-540-31987-0.

[Cou01] Patrick Cousot. "Abstract Interpretation Based Formal Methods and Future Challenges". In: *Informatics*. Ed. by Reinhard Wilhelm. Lecture Notes in Computer Science 2000. Springer Berlin Heidelberg, Jan. 2001, pp. 138–156. ISBN: 978-3-540-41635-7, 978-3-540-44577-7.

[CS00] Alessandra Carbone and Stephen Semmes. *A Graphic Apology for Symmetry and Implicitness*. Oxford University Press, 2000. ISBN: 0198507291.

[CS13] Cristian Cadar and Koushik Sen. “Symbolic Execution for Software Testing: Three Decades Later”. In: *Communications of the ACM* 56.2 (Feb. 2013), pp. 82–90.

[DE82] R.B. Dannenberg and G.W. Ernst. “Formal Program Verification Using Symbolic Execution”. In: *IEEE Transactions on Software Engineering* SE-8.1 (Jan. 1982), pp. 43–52.

[DHS05] Ádám Darvas, Reiner Hähnle, and David Sands. “A Theorem Proving Approach to Analysis of Secure Information Flow”. In: *Security in Pervasive Computing*. Ed. by Dieter Hutter and Markus Ullmann. Lecture Notes in Computer Science 3450. Springer Berlin Heidelberg, 2005, pp. 193–209. ISBN: 978-3-540-25521-5, 978-3-540-32004-3.

[Fin05] Marcelo Finger. “DAG Sequent Proofs with a Substitution Rule”. In: *We will show Them - Essays in honour of Dov Gabbays 60th birthday, vol. 1*. Ed. by S. Artemov et al. London: Kings College Publications, 2005, pp. 671–686.

[FOW84] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. “The Program Dependence Graph and its Use in Optimization”. In: *International Symposium on Programming*. Ed. by M. Paul and B. Robinet. Lecture Notes in Computer Science 167. Springer Berlin Heidelberg, Jan. 1984, pp. 125–132. ISBN: 978-3-540-12925-7, 978-3-540-38809-8.

[Gen64] Gerhard Gentzen. “Investigations into Logical Deduction”. In: *American philosophical quarterly* (1964), pp. 288–306.

[God12] Patrice Godefroid. “Test Generation using Symbolic Execution”. In: *LIPIcs-Leibniz International Proceedings in Informatics*. Vol. 18. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012, pp. 24–33.

[Gos+05] James Gosling et al. *The Java (TM) Language Specification*. 3rd. Addison-Wesley Professional, 2005. ISBN: 0321246780.

[Grä78] Georg Grätzer. *Lattice Theory: Foundation*. Pure and Applied Mathematics 75. Academic Press, 1978. ISBN: 0-12-2957504.

[Häh+10] Reiner Hähnle et al. “A Visual Interactive Debugger based on Symbolic Execution”. In: *Proceedings of the IEEE/ACM international conference on Automated software engineering*. Ed. by Jamie Andrews and Elisabetta Di Nitto. ACM Press, 2010, pp. 143–146.

[HHB14] Martin Hentschel, Reiner Hähnle, and Richard Bubel. “Visualizing Unbounded Symbolic Execution”. In: *Tests and Proofs*. Ed. by Martina Seidl and Nikolai Tillmann. Lecture Notes in Computer Science 8570. Springer International Publishing, Jan. 2014, pp. 82–98. ISBN: 978-3-319-09098-6, 978-3-319-09099-3.

[HKS06] Christian Hammer, Jens Krinke, and Gregor Snelting. “Information Flow Control for Java Based on Path Conditions in Dependence Graphs”. In: *Proceedings of the IEEE International Symposium on Secure Software Engineering (ISSSE 2006)*. Arlington, VA: IEEE, Mar. 2006, pp. 87–96.

[HSS09] Trevor Hansen, Peter Schachte, and Harald Søndergaard. “State Joining and Splitting for the Symbolic Execution of Binaries”. In: *Runtime Verification*. Ed. by Saddek Bensalem and Doron A. Peled. Lecture Notes in Computer Science 5779. Springer Berlin Heidelberg, 2009, pp. 76–92. ISBN: 978-3-642-04693-3, 978-3-642-04694-0.

[Jaf+12] Joxan Jaffar et al. “TRACER: A Symbolic Execution Tool for Verification”. In: *Proceedings of the 24th International Conference on Computer Aided Verification*. CAV’12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 758–766. ISBN: 978-3-642-31423-0.

[JH14] Ran Ji and Reiner Hähnle. *Information Flow Analysis Based on Program Simplification*. Technical Report TUD-CS-2014-0877. Department of Computer Science, Technische Universität Darmstadt, 2014.

[JMN13] Joxan Jaffar, Vijayaraghavan Murali, and Jorge A. Navas. “Boosting Concolic Testing via Interpolation”. In: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 48–58. ISBN: 978-1-4503-2237-9.

[Kin76] James C. King. “Symbolic Execution and Program Testing”. In: *Communications of the ACM* 19.7 (July 1976), pp. 385–394.

[Kuz+12] Volodymyr Kuznetsov et al. “Efficient State Merging in Symbolic Execution”. In: *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’12. New York, NY, USA: ACM, 2012, pp. 193–204. ISBN: 978-1-4503-1205-9.

[McM05] K. L. McMillan. “Applications of Craig Interpolants in Model Checking”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Nicolas Halbwachs and Lenore D. Zuck. Lecture Notes in Computer Science 3440. Springer Berlin Heidelberg, 2005, pp. 1–12. ISBN: 978-3-540-25333-4, 978-3-540-31980-1.

[OO84] Karl J. Ottenstein and Linda M. Ottenstein. “The Program Dependence Graph in a Software Development Environment”. In: *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. SDE 1. New York, NY, USA: ACM, 1984, pp. 177–184. ISBN: 0-89791-131-8.

[RRR13] Ran Ji, Reiner Hähnle, and Richard Bubel. *Program Transformation Based on Symbolic Execution and Deduction*. Technical Report TUD-CS-2013-0348. Darmstadt: Department of Computer Science, Technische Universität Darmstadt, 2013.

[Rüm03] Philipp Rümmer. *Ensuring the Soundness of Taclets – Constructing Proof Obligations for Java Card DL Taclets*. Minor Thesis, University of Karlsruhe, 2003.

[Sen+14] Koushik Sen et al. *MultiSE: Multi-Path Symbolic Execution using Value Summaries*. Tech. rep. UCB/EECS-2014-173. EECS Department, University of California, Berkeley, Oct. 2014.

[SM06] A. Sabelfeld and A. C. Myers. “Language-based Information-flow Security”. In: *IEEE J.Sel. A. Commun.* 21.1 (Sept. 2006), pp. 5–19.

[SRK06] Gregor Snelting, Torsten Robschink, and Jens Krinke. “Efficient Path Conditions in Dependence Graphs for Software Safety Analysis”. In: *ACM Trans. Softw. Eng. Methodol.* 15.4 (2006), pp. 410–457.

[SRW02] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. “Parametric Shape Analysis via 3-valued Logic”. In: *ACM Trans. Program. Lang. Syst.* 24.3 (2002), pp. 217–298.

[Xie+09] Tao Xie et al. “Fitness-guided path exploration in dynamic symbolic execution”. In: *IEEE/IFIP International Conference on Dependable Systems Networks, 2009. DSN ’09*. June 2009, pp. 359–368.

Further Reading

[Ahr+02] Wolfgang Ahrendt et al. "The Key System: Integrating Object-Oriented Design and Formal Methods". In: *Fundamental Approaches to Software Engineering*. Ed. by Ralf-Detlef Kutsche and Herbert Weber. Lecture Notes in Computer Science 2306. Springer Berlin Heidelberg, Jan. 2002, pp. 327–330. ISBN: 978-3-540-43353-8, 978-3-540-45923-1.

[Ahr+05] Wolfgang Ahrendt et al. "The KeY Tool". In: *Software & Systems Modeling* 4.1 (Feb. 2005), pp. 32–54.

[Ahr+06] Wolfgang Ahrendt et al. "Verifying Object-Oriented Programs with KeY: A Tutorial". In: *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*. Ed. by Frank S. de Boer et al. Vol. 4709. Lecture Notes in Computer Science. Springer, 2006, pp. 70–101. ISBN: 978-3-540-74791-8.

[Ahr+14] Wolfgang Ahrendt et al. "The KeY Platform for Verification and Analysis of Java Programs". In: *Verified Software: Theories, Tools and Experiments*. Ed. by Dimitra Giannakopoulou and Daniel Kroening. Lecture Notes in Computer Science. Springer International Publishing, Jan. 2014, pp. 55–71. ISBN: 978-3-319-12153-6, 978-3-319-12154-3.

[Bal+00] Michael Balser et al. "Formal System Development with KIV". In: *Fundamental Approaches to Software Engineering*. Ed. by Tom Maibaum. Lecture Notes in Computer Science 1783. Springer Berlin Heidelberg, Jan. 2000, pp. 363–366. ISBN: 978-3-540-67261-6, 978-3-540-46428-0.

[BHJ12] Richard Bubel, Reiner Hähnle, and Ran Ji. "Program Specialization via a Software Verification Tool". In: *Formal Methods for Components and Objects*. Ed. by Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue. Lecture Notes in Computer Science 6957. Springer Berlin Heidelberg, Jan. 2012, pp. 80–101. ISBN: 978-3-642-25270-9, 978-3-642-25271-6.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224.

[Deu76] L. Peter Deutsch. *An Interactive Program Verifier*. Xerox Palo Alto Research Center, 1976.

[DLR06] Xianghua Deng, Jooyong Lee, and Robby. "Bogor/Kiasan: A k-bounded Symbolic Execution for Checking Strong Heap Properties of Open Systems". In: *21st IEEE/ACM International Conference on Automated Software Engineering, 2006. ASE '06*. Sept. 2006, pp. 157–166.

[EH07] Christian Engel and Reiner Hähnle. "Generating Unit Tests from Formal Proofs". In: *Tests and Proofs*. Ed. by Yuri Gurevich and Bertrand Meyer. Lecture Notes in Computer Science 4454. Springer Berlin Heidelberg, Jan. 2007, pp. 169–188. ISBN: 978-3-540-73769-8, 978-3-540-73770-4.

[FJM05] Jeffrey Fischer, Ranjit Jhala, and Rupak Majumdar. "Joining Dataflow with Predicates". In: *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ESEC/FSE-13. New York, NY, USA: ACM, 2005, pp. 227–236. ISBN: 1-59593-014-0.

[FQ02] Cormac Flanagan and Shaz Qadeer. "Predicate Abstraction for Software Verification". In: *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '02. New York, NY, USA: ACM, 2002, pp. 191–202. ISBN: 1-58113-450-9.

[GAP10] Miguel Gómez-Zamalloa, Elvira Albert, and Germán Puebla. "Test Case Generation for Object-oriented Imperative Languages in Clp*". In: *Theory Pract. Log. Program.* 10.4-6 (July 2010), pp. 659–674.

[GL11] Patrice Godefroid and Daniel Luchaup. "Automatic Partial Loop Summarization in Dynamic Test Generation". In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ACM, 2011, pp. 23–33.

[GL75] S. Galand and G. Loncour. “Structured Implementation of Symbolic Execution: A First Part in a Program Verifier”. In: *Information Processing Letters* 3.4 (Mar. 1975), pp. 97–103.

[Häh+86] R. Hähnle et al. “An Interactive Verification System based on Dynamic Logic”. In: *8th International Conference on Automated Deduction*. Ed. by Jörg H. Siekmann. Lecture Notes in Computer Science 230. Springer Berlin Heidelberg, Jan. 1986, pp. 306–315. ISBN: 978-3-540-16780-8, 978-3-540-39861-5.

[HK76] Sidney L. Hantler and James C. King. “An Introduction to Proving the Correctness of Programs”. In: *ACM Comput. Surv.* 8.3 (Sept. 1976), pp. 331–353.

[HRS87] M. Heisel, W. Reif, and W. Stephan. “Program Verification by Symbolic Execution and Induction”. In: *GWAI-87 11th German Workshop on Artifical Intelligence*. Ed. by Katharina Morik. Informatik-Fachberichte 152. Springer Berlin Heidelberg, Jan. 1987, pp. 201–210. ISBN: 978-3-540-18388-4, 978-3-642-73005-4.

[HT08] Jonathan de Halleux and Nikolai Tillmann. “Parameterized Unit Testing with Pex”. In: *Tests and Proofs*. Ed. by Bernhard Beckert and Reiner Hähnle. Lecture Notes in Computer Science 4966. Springer Berlin Heidelberg, Jan. 2008, pp. 171–181. ISBN: 978-3-540-79123-2, 978-3-540-79124-9.

[HTK00] David Harel, Jerzy Tiuryn, and Dexter Kozen. *Dynamic Logic*. Cambridge, MA, USA: MIT Press, 2000. ISBN: 0262082896.

[Jam+12] Konrad Jamrozik et al. “Augmented Dynamic Symbolic Execution”. In: *Automated Software Engineering (ASE), 2012 Proceedings of the 27th IEEE/ACM International Conference on*. Ed. by Michael Goedicke, Tim Menzies, and Motoshi Saeki. Essen, Germany: IEEE, 2012, pp. 254–257.

[JB12] Ran Ji and Richard Bubel. “PE-KeY: A Partial Evaluator for Java Programs”. In: *Integrated Formal Methods*. Ed. by John Derrick et al. Lecture Notes in Computer Science 7321. Springer Berlin Heidelberg, Jan. 2012, pp. 283–295. ISBN: 978-3-642-30728-7, 978-3-642-30729-4.

[JHB13] Ran Ji, Reiner Hähnle, and Richard Bubel. “Program Transformation Based on Symbolic Execution and Deduction”. In: *Software Engineering and Formal Methods*. Ed. by Robert M. Hierons, Mercedes G. Merayo, and Mario Bravetti. Lecture Notes in Computer Science 8137. Springer Berlin Heidelberg, Jan. 2013, pp. 289–304. ISBN: 978-3-642-40560-0, 978-3-642-40561-7.

[Ji14] Ran Ji. “Sound Program Transformation Based on Symbolic Execution and Deduction”. Dissertation. TU Darmstadt, June 2014.

[JP08] Bart Jacobs and Frank Pessens. *The VeriFast Program Verifier*. Technical Report CW-520. Department of Computer Science, Katholieke Universiteit Leuven, Aug. 2008.

[Kne91] Ralf Kneuper. “Symbolic Execution: A Semantic Approach”. In: *Science of Computer Programming* 16.3 (1991), pp. 207–249.

[Lei05] K. Rustan M. Leino. “Efficient Weakest Preconditions”. In: *Inf. Process. Lett.* 93.6 (2005), pp. 281–288.

[NNH99] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999. ISBN: 3540654100.

[PV04] Corina S. Păsăreanu and Willem Visser. “Verification of Java Programs Using Symbolic Execution and Invariant Generation”. In: *Model Checking Software*. Ed. by Susanne Graf and Laurent Mounier. Lecture Notes in Computer Science 2989. Springer Berlin Heidelberg, 2004, pp. 164–181. ISBN: 978-3-540-21314-7, 978-3-540-24732-6.

[SST13] Jiri Slaby, Jan Strejček, and Marek Trtík. “Compact Symbolic Execution”. In: *Automated Technology for Verification and Analysis*. Ed. by Dang Van Hung and Mizuhito Ogawa. Lecture Notes in Computer Science 8172. Springer International Publishing, 2013, pp. 193–207. ISBN: 978-3-319-02443-1, 978-3-319-02444-8.

[VP11] Dries Vanoverberghe and Frank Piessens. “Theoretical Aspects of Compositional Symbolic Execution”. In: *Fundamental Approaches to Software Engineering*. Ed. by Dimitra Giannakopoulou and Fernando Orejas. Lecture Notes in Computer Science 6603. Springer Berlin Heidelberg, Jan. 2011, pp. 247–261. ISBN: 978-3-642-19810-6, 978-3-642-19811-3.