A General Lattice Model for Merging Symbolic Execution Branches *

Dominic Scheurer, Reiner Hähnle, and Richard Bubel

TU Darmstadt, Dept. of Computer Science, Darmstadt, Germany
{scheurer, haehnle, bubel}@cs.tu-darmstadt.de

Abstract. Symbolic execution is a software analysis technique that has been used with success in the past years in program testing and verification. A main bottleneck of symbolic execution is the path explosion problem: the number of paths in a symbolic execution tree is exponential in the number of static branches of the executed program. Here we put forward an abstraction-based framework for state merging in symbolic execution. We show that it subsumes existing approaches and prove soundness. The method was implemented in the verification system KeY. Our empirical evaluation shows that reductions in proof size of up to 80% are possible by state merging when applied to complex verification problems; new proofs become feasible that were out of reach so far.

1 Introduction

Symbolic execution [7,20] is a classic program analysis technique that was used with considerable success in the past years, for example, in program testing [8] and program verification [4]. One of the main bottlenecks of symbolic execution is the path explosion problem [8]. It stems from the fact that symbolic execution must explore all symbolic paths of a program to achieve high coverage (in testing), respectively, soundness (in verification). As a consequence, the number of paths from the root to the leaves in a symbolic execution tree is usually exponential in the number of static branches of the executed program.

Various strategies are in use to mitigate path explosion, including subsumption [3,9], method contracts [5] and value summaries [23]. The last two allow one to perform symbolic execution per method: different symbolic execution paths are merged into the postcondition of a contract or a value summary (a conditional execution state over guard expressions). Summaries are computed on the fly and bottom-up, while contracts characterize all possible behaviors and must at least partially be written by hand. Unfortunately, even the use of rich contracts (instead of inlining) is insufficient to deal with complex problems [15].

A seemingly obvious technique to alleviate state explosion in symbolic execution trees consists of merging the states resulting from a symbolic execution

* **The final publication is available at Springer via** http://dx.doi.org/10.1007/978-3-319-47846-3_5.

step that caused a split (e.g., guard evaluation, statements that can throw exceptions, polymorphic method calls). After all, graph-based data structures are standard in model checking for the exploration of symbolic state spaces [10], as well as in other static software analyses. Indeed, several state merging variants were suggested for symbolic execution [18,21,23], but there are problems:

(1) State merging does not come for free, but creates considerable overhead: states must be merged, graph data structures are more complex than trees, path conditions as well as summaries tend to grow fast and must be simplified. Eager state merging can make things worse [18], therefore, it has to be carefully *controlled*. Simplification of intermediate expressions with the help of automated deduction is indispensable.

(2) All mentioned approaches assume that merged states are identical to corresponding unmerged states, possibly up to the differences encoded in value summaries. This is insufficient to merge large numbers of different behaviors.

In the present paper we address both issues. Regarding the second, we observe that instead of encoding merged states precisely into a conditional state, one might also *abstract* from the precise value of a variable. This results in a loss of precision, but reduces complexity. For example, consider symbolic execution of "**if** (b) x=1 **else** x=2;" in state σ. Precise state merging would result in a state identical to σ except the value of x is "$\mathrm{x} \mapsto (1, \text{ if } \sigma(\mathrm{b}) = \mathit{true}) \mid x \mapsto (2, \text{ if } \sigma(\mathrm{b}) = \mathit{false})$". This does not avoid path explosion, it only delays it. Now, assume that we define an abstract domain A for the possible values of x, where $\alpha(\mathrm{x})$ is the abstraction of x and A is an upper semilattice $\sqcup$. For example, A might be the sign lattice $\{\bot, -, 0, +, \top\}$. Then the merged state can be *over-approximated* by the partially abstract state that is identical to σ except $\mathrm{x} \mapsto +$. Path explosion is avoided. We lost precision about the exact value of x, but for many analyses this is acceptable provided that the abstract lattice is suitably chosen.

Based on the theory of symbolic execution with abstract interpretation [6], in the present paper we put forward a general lattice-based framework for state merging in symbolic execution where a family of abstract lattices is defined by formulas of a program logic. Our framework preserves soundness of verification and we show that it subsumes earlier approaches to state merging [18,21,23].

Regarding issue (1) above, as a second contribution, we improved automation by implementing suitable proof macros for KeY as well as an extension of the Java Modeling Language (JML)[1] which allows software engineers to annotate Java source code with instructions on when to perform state merges.

We implemented the framework in the state-of-art verification system KeY [5], where contracts are available to mitigate state explosion. Since the latter must be partially written by hand, state merging is a complementary technique that promises a high degree of automation. We confirmed the usefulness of our approach empirically with an extensive evaluation: Results for small to medium sized programs are, as expected, mixed, because of the overhead of state merging. The strength of symbolic execution with state merging emerges when applied to complex verification problems like the TimSort implementation in the Java

[1] `http://www.eecs.ucf.edu/~leavens/JML//OldReleases/jmlrefman.pdf`

standard library [15], where we observe reductions in proof size of up to 80%. Additionally, some proofs become feasible that were elusive before.

We continue with Sect. 2, which provides sufficient theoretical background to make the paper self-contained. Sect. 3 sets up our abstract lattice model. Sect. 4 defines the actual merge rules, states a soundness theorem and briefly mentions implementation issues. Sect. 5 contains the empirical evaluation, Sect. 6 lessons learned as well as future work, and Sect. 7 related work plus a brief conclusion.

2 Background

We formalize our theory in the program logic JavaDL [5], but the approach is easily adaptable to any program logic with an explicit notion of symbolic state.

2.1 Program Logic and Calculus

Our framework is realized in JavaDL, a sorted first-order dynamic logic [19] for sequential deterministic Java programs. For the sake of presentation, we restrict ourselves to a simplified JavaDL variant (simple imperative Java programs over primitive types **int**, **boolean**) and only give the essential definitions. The actual implementation is based on KeY which covers most sequential Java features: inheritance, dynamic dispatch, reference types, recursive methods, exceptions, and strings. Not covered are generic types (which are translated away), floating point types and lambda expressions. We refer the reader to [5] for a full account.

JavaDL extends sorted first-order logic by two modalities to express partial and total correctness of programs. For space reasons, we restrict ourselves to the former, the box modality $[\cdot]\cdot$. Its first argument is a program (more precisely, an executable sequence of Java statements); the second argument can be any JavaDL formula, possibly containing further modal operators. Given a program p and a JavaDL formula φ, the informal meaning of the formula $[p]\,\varphi$ is: *if* the program p terminates *then* the formula φ holds in the final state.

The syntax of terms and formulas is standard except for a few extra cases like modalities, conditional terms/formulas (Ex. 1) as well as *updates*. The set of all programs is *Prg*; the set of all program variables is denoted by PV. Updates represent state changes: an *elementary update* has the form $\mathrm{l} := t$ with $\mathrm{l} \in \mathsf{PV}$ and t a term of a type compatible with l. Informally, an update has the same meaning as an assignment, where the program variable on the left-hand side is assigned the value of the right-hand side. Elementary updates are combined to parallel updates $U_1 \parallel \cdots \parallel U_n$ which represent simultaneous assignments. In case of a clash where the same variable l is assigned different values in a parallel update, the syntactically later assignment wins. The set of all updates is Upd, *skip* is the "empty update". Updates U can be applied to terms t, written $\{U\}t$, and formulas φ, written $\{U\}\varphi$. We give the non-standard cases of the inductive definitions of terms and formulas:

Definition 1 (Terms). *Let φ denote a formula, t_1, t_2 are terms of type T_1 and T_2 and U an update, then (i) $\{U\}t_1$ is a term of type T_1 (ii)* $\mathsf{if}\,(\varphi)\,\mathsf{then}\,(t_1)\,\mathsf{else}\,(t_2)$

is a term of type T where T is the least common supertype of T_1 and T_2. The set of all terms is denoted by Trm.

Definition 2 (Formulas). *Let φ, ψ_1, ψ_2 denote formulas, U an update and p a program, then each of (i) $[p]\,\varphi$, (ii) $\{U\}\varphi$ and (iii)* $\mathsf{if}\,(\varphi)\,\mathsf{then}\,(\psi_1)\,\mathsf{else}\,(\psi_2)$ *is a formula. For open formulas with free variables $\overline{v}$ we use the notation $\varphi_{\overline{v}}$ to make their occurrence explicit. The set of all formulas is denoted by* Fml.

Example 1. Let i, j be program variables and x, y logic variables, all of sort **int**.

- The formula $\forall x, y; \big(\mathrm{i} \doteq x \land \mathrm{j} \doteq y \to \{\mathrm{i} := \mathrm{j} \parallel \mathrm{j} := \mathrm{i}\}(\mathrm{i} \doteq y \land \mathrm{j} \doteq x)\big)$ uses a parallel update to exchange the values of i and j.
- The formula $\mathrm{i} > \mathrm{j} \to [\mathrm{i}{=}\mathrm{i}{-}\mathrm{j};]\,\mathrm{i} > 0$ expresses that if program i=i−j; is executed in a state where the value of i is greater than the value of j and it terminates, then in its final state i is positive.
- $\mathsf{if}\,(\mathrm{i} > \mathrm{j})\,\mathsf{then}\,(\mathrm{i})\,\mathsf{else}\,(\mathrm{j}) \geq 0$ means that the maximum of i and j is positive.

Formulas are evaluated in first-order structures with a non-empty domain D and an interpretation function I giving meaning to sort, function and predicate symbols. To reason about programs, we extend this to *Kripke structures* $K = (D, I, S, \rho)$. The values of program variables depend on the current program state and cannot be evaluated by the static interpretation function I. Instead they are assigned values by *Kripke states* $\sigma : \mathsf{PV} \to D \in S$. The state transition function $\rho : \mathit{Prg} \to (S \to 2^S)$ captures the program semantics (here: Java's semantics [14]).[2]

As our programs are deterministic, the value of $\rho(p)(\sigma)$ (for any program p and state σ) is either the empty set (p does not terminate when started in state σ) or a singleton. Formulas and terms are assigned meaning by an *evaluation function* $\mathit{val}\,(K, \sigma, \beta; \cdot)$, parametric in a Kripke structure K, a state σ and a variable assignment β. The evaluation function assigns terms a value in their domain and formulas one of the truth values tt, ff. Fig. 1 shows some inductive definition cases. For expressions without free logic variables, we write $\mathit{val}\,(K, \sigma; \cdot)$; for sets of closed formulas C, we write $\mathit{val}\,(K, \sigma; C)$ meaning $\mathit{val}\left(K, \sigma; \bigwedge_{\varphi \in C} \varphi\right)$.

A sequent calculus [12], [5, Ch. 3] is used to prove the validity of JavaDL formulas. The rules for the first-order logic connectives are standard, those for programs follow the symbolic execution paradigm. Formulas with programs are transformed into pure first-order formulas by symbolically executing the programs in a forward manner and thereby computing the weakest precondition. Each execution step transforms or eliminates the first statement until the program is eliminated. We write $\vdash \varphi$ if a formula φ is *provable* using the calculus.

[2] Our notion of Kripke structure is derived from that commonly used in modal logic [13] and slightly differs from the one often used in model checking. E.g., we require no fixed set of initial states, and the labeling function is given implicitly by the interpretation and Kripke state which is natural for imperative programs. There is no essential difference, however.

Programs $val\,(K,\sigma,\beta;\cdot) : Prg \to (S \to 2^S)$ with $val\,(K,\sigma,\beta;p)\,(\sigma_1) = \rho(p)(\sigma_1)$

Terms $val\,(K,\sigma,\beta;\cdot) : \mathrm{Trm} \to D$ with

$$val\,(K,\sigma,\beta;(f(t_1,\ldots t_n)) = I(f)(val\,(K,\sigma,\beta;t_1),\ldots,val\,(K,\sigma,\beta;t_n))$$

$$val\,(K,\sigma,\beta;\mathsf{if}\,(\varphi)\,\mathsf{then}\,(t_1)\,\mathsf{else}\,(t_2)) = \begin{cases} val\,(K,\sigma,\beta;t_1) & val\,(K,\sigma,\beta;\varphi) = tt \\ val\,(K,\sigma,\beta;t_2) & \text{otherwise} \end{cases}$$

$$val\,(K,\sigma,\beta;\{U\}t) = val\,(K,val\,(K,\sigma,\beta;U)\,,\beta;t)$$

Formulas $val\,(K,\sigma,\beta;\cdot) : \mathrm{Fml} \to \{tt,ff\}$ with

$$val\,(K,\sigma,\beta;[p]\varphi) = \begin{cases} val\,(K,\sigma',\beta;\varphi) & val\,(K,\sigma,\beta;p)\,(\sigma) = \{\sigma'\} \\ tt & otherwise \end{cases}$$

$$val\,(K,\sigma,\beta;\{U\}\varphi) = val\,(K,val\,(K,\sigma,\beta;U)\,,\beta;\varphi)$$

Updates $val\,(K,\sigma,\beta;\cdot) : \mathrm{Upd} \to S$ with

$$val\,(K,\sigma,\beta;\mathrm{l} := t) = \sigma' \quad \text{where } \sigma'(x) = \begin{cases} \sigma(\mathrm{x}) & \text{if x} \neq \mathrm{l} \\ val\,(K,\sigma,\beta;t) & \text{otherwise} \end{cases}$$

Fig. 1. Excerpt of the definition of $val\,(K,\sigma,\beta;\cdot)$

2.2 Symbolic Execution

We explain how the notions and concepts introduced in the standard literature [7] relate to our logic-based setting. Symbolic Execution (SE) of a program results in a Symbolic Execution Tree (SET) consisting of SE *states*, i.e., triples (U,C,φ) with (1) an update U, the *symbolic state*, capturing the changes made to program variables in the course of the execution, (2) the *path condition* C, a set of closed formulas comprising the decisions leading to this particular SE path as well as additional preconditions, and (3) the *program counter* φ, a closed formula, typically containing the remaining program to be executed as well as the postcondition to prove.

Please note that we generalize the usual notion of a *program counter*, which is normally only a pointer to a statement in the executed program. In our setting, a program counter may be an arbitrary closed formula. This generalization facilitates reasoning about the *validity* of SE states.

Definition 3 (Validity of SE States). *An SE state $s = (U,C,\varphi)$ is called* valid *iff for all Kripke structures K and states σ either $val\,(K,\sigma;C) = ff$ or $val\,(K,\sigma;\{U\}\varphi) = tt$ holds. We write $valid\,(s)$.*

Consider an SE state $s = (U,C,[\mathrm{p}]\varphi)$. Intuitively, if the path condition C does not hold in s, then the state is unreachable and trivially valid. Otherwise, all final state(s) reached when executing p in state $val\,(K,\sigma;U)$ must satisfy φ.

SET transitions are constrained by a rule-based *SE transition relation* $\delta : 2^{SEStates} \to 2^{SEStates}$, where *SEStates* is the set of all SE states. Again, this is a generalization of traditional SE, since the result of applying δ does not have to be a singleton. Based on Defn. 3, we introduce a soundness notion for δ.

Definition 4 (Soundness of Symbolic Execution). *An SE transition relation $\delta : 2^{SEStates} \to 2^{SEStates}$ is called* sound *iff for each input-output pair $(I,O) \in \delta$ it is the case that $\bigwedge_{o\in O} valid\,(o) \implies \bigwedge_{i\in I} valid\,(i)$.*

The intuition behind the above definition is that, whenever one input state is not valid, also at least one output state must not be valid. Otherwise, it would be possible to derive an invalid property about a program.

2.3 Running Example

Listing 1. Distance of two positive integers

```
1 public int dist(int x, int y) {
2     if (y < x) {
3         int tmp = x;
4         x = y;
5         y = tmp;
6     } else {}
7     return y − x;
8 }
```

The program in Listing 1 is our running example. It computes the absolute difference (distance) between two positive numbers. Aiming to prove that the result is never negative, we start with the SE state below as initial SE state (the return value is assigned to the global program variable "result"):

$$(\overbrace{skip}^{U}, \overbrace{\{\mathrm{x} > 0, \mathrm{y} > 0\}}^{C}, \overbrace{[\textbf{if } (\mathrm{y}{<}\mathrm{x}) \ \{...\} \ \textbf{else } \{\} \ \mathrm{result}{=}\mathrm{y}{-}\mathrm{x};]\,\mathrm{result} \geq 0}^{\varphi})$$

The *SE* state (U, C, φ) given above is valid iff for any *Kripke* state σ satisfying the path condition C, whenever we execute the program **if** (y<x) ... ; in σ, then in the reached final state the value of program variable result is non-negative.

We explain how the SET for the example is constructed by stepwise symbolic execution: Symbolic execution of the first statement (by applying the appropriate calculus rule) splits the SET into the following two new intermediate states:

$$(skip, \{\mathrm{x} > 0,\ \mathrm{y} > 0,\ \mathrm{y} < \mathrm{x}\}, [\,\{...\}\ \ \mathrm{result}{=}\mathrm{y}{-}\mathrm{x};]\,\mathrm{result} \geq 0)$$
$$(skip, \{\mathrm{x} > 0,\ \mathrm{y} > 0,\ \neg\mathrm{y} < \mathrm{x}\}, [\{\}\ \ \mathrm{result}{=}\mathrm{y}{-}\mathrm{x};]\,\mathrm{result} \geq 0)$$

On each branch, either the body of the then-branch or the else-branch has to be executed, followed by the remaining program. The remaining program is just a single assignment statement here, but could be arbitrarily complex in general. In addition, the path condition in each branch has been extended by the conjunct $\mathrm{y} < \mathrm{x}$ and its negation, respectively. Continuing symbolic execution on the first branch results in the state

$$((\mathrm{x} \coloneqq \mathrm{y} \parallel \mathrm{y} \coloneqq \mathrm{x}), \{\mathrm{x} > 0,\ \mathrm{y} > 0,\ \mathrm{y} < \mathrm{x}\}, [\mathrm{result}{=}\mathrm{y}{-}\mathrm{x};]\,\mathrm{result} \geq 0)) \quad .$$

The motivation for state merging becomes very clear now: on each branch the same remaining program has to be executed.

3 The General Lattice Model

Symbolic Execution can be cast as *abstract interpretation* [11]. Each SE state describes a potentially infinite set of *concrete states*. As abstract interpretation demands a complete semilattice with join operation, partial order, least and top element, we define a concretization function from SE states to concrete states as well as a partial order relation between SE states.

Definition 5 (Concrete Execution States). *A* concrete execution state *is a pair (σ, φ) of a Kripke state σ and a formula φ (the program counter).*

A concrete execution state for a given program counter assigns to each program variable a concrete value of the universe. We define the semantics of SE states by stipulating a concretization function from SE states to concrete states based on the evaluation function $val\,(K, \sigma; \cdot)$ (β is not needed as formulas are closed).

Definition 6 (Concretization Function). *Let $s = (U, C, \varphi)$ be an SE state. The* concretization function *concr maps s to the set of concrete states*

$$concr\,(s) := \Big\{ (\sigma', \varphi) : \sigma' = val\,(K, \sigma; U) \text{ and } K, \sigma \text{ is an arbitrary structure / Kripke state such that } val\,(K, \sigma; C) = tt \Big\}$$

The set of concrete states for a symbolic state s contains all pairs of Kripke states σ' and the program counter such that σ' can be reached via some state σ satisfying s's path condition in some Kripke structure. So the set $concr\,(s)$ contains exactly the concrete states that are described by the SE state s. Consider, for instance, the SE state $(\mathrm{x} \coloneqq c, \{c > 0\}, \varphi)$: The set of concretizations for this state consists of all pairs (σ, φ), where σ is any function mapping the program variable x to a strictly positive integer.

Based on Defn. 6, we define a *weakening relation* expressing that one symbolic execution state describes more concrete states than another one.

Definition 7 (Weakening Relation). *Let s_1, s_2 be two SE states. State s_2 is* weaker than *(a* weakening of*) s_1 (written: $s_1 \lesssim s_2$) iff $concr\,(s_1) \subseteq concr\,(s_2)$.*

Given a state s_1 with satisfiable path condition, Defs. 6 and 7 imply that a state s_2 can only be weaker than s_1 if both have syntactically the same program counter. States with unsatisfiable path condition have an empty set of concretizations and hence are stronger than any other state.

Consider the SE states $s_1 = (\mathrm{x} \coloneqq c, \{c > 0\}, \varphi)$ and $s_2 = (\mathrm{x} \coloneqq c, \{c \geq 0\}, \varphi)$. The set of concretizations of s_2 contains all concrete states of s_1 and additionally all concrete states that map x to zero, hence s_2 is a weakening of s_1 ($s_1 \lesssim s_2$).

Consider the SE state $s_3 = (\mathrm{x} \coloneqq \mathsf{if}\,(\mathsf{true})\,\mathsf{then}\,(c)\,\mathsf{else}\,(t), \{c > 0\}, \varphi)$. Although s_1 and s_3 are syntactically different, all Kripke models coincide on the value of x and we would actually prefer to consider them as equal. Hence, we define an extensional equality $s_1 \overset{concr}{=} s_2 :\Leftrightarrow concr\,(s_1) = concr\,(s_2)$ stating that symbolic execution states are equal iff they evaluate to the same set of concrete execution states. Using $\overset{concr}{=}$ as equality, we can state the following lemma:

Lemma 1. *The weakening relation $\lesssim$ is a partial order relation.*

The core of our formal framework 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, see Defn. 7.

Definition 8 (Induced Join-Semilattice of States). *Let $\varphi \in$ Fml be a closed formula. The* carrier set *S_φ for φ is defined as*

$$S_\varphi \coloneqq \{\, (U, C, \varphi) \mid (U, C, \varphi) \text{ is an SE state} \}.$$

A join-semilattice of SE states *is a structure* $(S_\varphi, \dot\sqcup)$ *over* S_φ *with operator* $\dot\sqcup$ *s.t. the semilattice properties (based on* $\overset{concr}{=}$*) (SEL1)–(SEL3) hold for all* $a, b, c \in S_\varphi$*:*

(SEL1) Idempotency: $a \mathbin{\dot\sqcup} a \overset{concr}{=} a$ *(SEL2)* Commutativity: $a \mathbin{\dot\sqcup} b \overset{concr}{=} b \mathbin{\dot\sqcup} a$

(SEL3) Associativity: $(a \mathbin{\dot\sqcup} b) \mathbin{\dot\sqcup} c \overset{concr}{=} a \mathbin{\dot\sqcup} (b \mathbin{\dot\sqcup} c)$

Furthermore, we require that the partial order relation $\preceq$ *on* S_φ *defined as*

$$a \preceq b :\Leftrightarrow a \mathbin{\dot\sqcup} b \overset{concr}{=} b$$

satisfies (SEL4) and (SEL5) for $a = (U_a, C_a, \varphi) \in S_\varphi$ *and* $b = (U_b, C_b, \varphi) \in S_\varphi$*:*

(SEL4) Correctness: $a \preceq b$ *implies* $a \lesssim b$

(SEL5) Conservativity: $a \preceq b$ *implies that* C_b *is logically equivalent to a formula* $C \wedge Ax_{\overline{v}}[\overline{c}\,/\overline{v}]$,[3] *where (1)* $\overline{c}$ *are all uninterpreted Skolem constants occurring in* b *but not contained in* a*, (2)* C *does not contain any of the* $\overline{c}$*, (3)* $\bigwedge C_a \rightarrow C$ *is provable, and (4) the formula* $\exists \overline{v}; Ax_{\overline{v}}$ *is provable.*

We call $\{\mathbb{L}_\varphi\}_{\varphi \in \mathrm{Fml}} := \{(S_\varphi, \dot\sqcup)\}_\varphi$ *the* induced family of join-semilattices *for* $\dot\sqcup$*.*

We term (SEL4) *correctness* since it enables, together with (SEL5), to prove the correctness of our state merging rule (Thm. 1 below). The *conservativity* property (SEL5) imposes restrictions on merge operations that introduce Skolem constants (thus extending the signature), such as the abstraction technique introduced in Sect. 4.3. Property (SEL5) enforces that the path condition of a merged state is divisible into (1) a formula C without new constants which is implied by the states that are merged (for example, the disjunction of the path conditions of the merged states) and (2) a formula $Ax_{\overline{v}}[\overline{c}\,/\overline{v}]$ providing restrictions on the values of the new constants. In addition (3) it must be possible in every structure to assign values to the new constants such that $Ax_{\overline{v}}[\overline{c}\,/\overline{v}]$ holds. This is achieved by proving $\exists \overline{v}; Ax_{\overline{v}}$ in the unextended signature of the merged states. $Ax_{\overline{v}}$ may be seen as a (generalized) *defining axiom* [24] for the $\overline{c}$: we only demand the existence condition $\vdash \exists \overline{v}; Ax_{\overline{v}}$ and explicitly forgo the uniqueness condition to facilitate abstraction. In summary, (SEL5) allows only "conservative" extensions to a merged path condition. An example for a formula $Ax_{\overline{v}}[\overline{c}\,/\overline{v}]$ is $c > 0$, where c is a constant introduced in the merging step. Example 2 (Sect. 4.2) shows a fragment of a join-semilattice induced by a join operation based on the $\mathsf{if}\,(\cdot)\,\mathsf{then}\,(\cdot)\,\mathsf{else}\,(\cdot)$ operator.

4 State Merging Techniques

We instantiate our framework with two join operations: the *If-Then-Else* (ITE) technique, a "classic" of state merging for symbolic execution (e.g., [18,21,23])

[3] $\psi_{\overline{v}}[\overline{t}\,/\overline{v}]$ denotes the substitution of the terms $\overline{t}$ for the free variables $\overline{v}$ in $\psi_{\overline{v}}$.

with full precision; and an abstraction-based technique which trades efficiency with potential loss of precision. To simplify specification of the join operations, we define a pattern that can be instantiated with specific merging techniques.

4.1 A State Merging Pattern

Definition 9. *Given two SE states* $s_j = (U_j, C_j, \varphi)$, $j = 1, 2$, *with program variables* $\mathrm{x}_1, \ldots, \mathrm{x}_n \in \mathrm{PV}$ *of type* T *occurring in the* U_j. *A merge technique* M *defines two functions* $\mathsf{joinVal}\,(s_1, s_2; \mathrm{x}, c_{\mathrm{X}})$ *and* $\mathsf{constraints}\,(s_1, s_2; \mathrm{x}, c_{\mathrm{X}})$ *mapping* s_1, s_2, *program variable* x *and a fresh (for* x*) Skolem constant* c_{X} *to a closed term and a JavaDL formula, respectively. The join operation* $\dot{\sqcup}_M$ *is defined by*

$$s_1 \;\dot{\sqcup}_M\; s_2 \coloneqq (U^*, C^*, \varphi) = \left((U_1, C_1) \oplus (U_2, C_2)\,, (U_1, C_1) \circledcirc (U_2, C_2)\,, \varphi\right)$$

where $U^* = (U_1, C_1) \oplus (U_2, C_2) \coloneqq (\mathrm{x}_1 \coloneqq t_1 \parallel \mathrm{x}_2 \coloneqq t_2 \parallel \cdots \parallel \mathrm{x}_n \coloneqq t_n)$. *To define the terms* t_i, *let* $c_{\mathrm{X}_1}, c_{\mathrm{X}_2}, \ldots, c_{\mathrm{X}_n}$ *be fresh Skolem constants of suitable types. Then*

$$t_i \coloneqq \begin{cases} \{U_1\}\,\mathrm{x}_i & \textit{if } (\star) \textit{ holds} \\ \mathsf{joinVal}\,(s_1, s_2; \mathrm{x}_i, c_{\mathrm{X}_i}) & \textit{otherwise} \end{cases}$$

Define $C^* = (U_1, C_1) \circledcirc (U_2, C_2) \coloneqq (\bigwedge C_1 \vee \bigwedge C_2) \wedge \{U^*\}\big(\bigwedge C_i^{abs}\big)$ *where*

$$C_i^{abs} \coloneqq \begin{cases} \mathsf{true} & \textit{if } (\star) \textit{ holds} \\ \mathsf{constraints}\,(s_1, s_2; \mathrm{x}_i, c_{\mathrm{X}_i}) & \textit{otherwise} \end{cases}$$

Condition $(\star)$ *holds if* x_i *is evaluated identically in either state and defined as*

$$(\star) \quad \vdash (C_1 \to \{U_1\}\,P\,(\mathrm{x}_i)) \leftrightarrow (C_2 \to \{U_2\}\,P\,(\mathrm{x}_i))$$

where P *is a fresh (for* $U_1, U_2, C_1, C_2, \varphi$*) predicate symbol.*

The provability relation "$\vdash$" in $(\star)$ is undecidable, but it can be safely approximated in practice. For example, a prover may simply return "unprovable" after exceeding a fixed time limit. This way soundness is maintained at the cost of completeness due to overapproximation in some situations. The update application of $\{U^*\}$ to $\big(\bigwedge C_i^{abs}\big)$ allows to take into account relations between values of program variables changed by the merge (e.g., the merge by predicate abstraction for the `dist` example in Sect. 5). Otherwise, only relations between constants and values before the merge would be reflected.

4.2 The If-Then-Else Technique

Definition 10 (If-Then-Else Merge). *Given two SE states* $s_j = (U_j, C_j, \varphi)$, $j = 1, 2$, *the join operation* $\dot{\sqcup}_{ite}$ *is defined by*

$$\begin{aligned} \mathsf{joinVal}\,(s_1, s_2; \mathrm{x}, c_{\mathrm{X}}) &\coloneqq \mathsf{if}\left(\bigwedge C_1\right)\mathsf{then}\,(\{U_1\}\mathrm{x})\,\mathsf{else}\,(\{U_2\}\mathrm{x}) \\ \mathsf{constraints}\,(s_1, s_2; \mathrm{x}, c_{\mathrm{X}}) &\coloneqq \mathsf{true} \end{aligned}$$

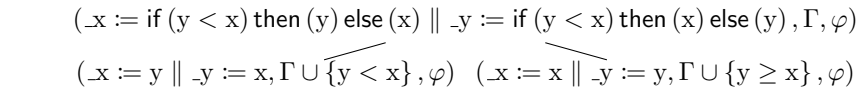

Fig. 2. Small excerpt of $(S_\varphi, \dot{\sqcup}_{ite})$ for the `dist` example.

The definition can be generalized by allowing a *distinguishing formula* for the first argument of the $\mathsf{if}\,(\cdot)\,\mathsf{then}\,(\cdot)\,\mathsf{else}\,(\cdot)$ term instead of $\bigwedge C_1$. It suffices to find a set of sub-conjuncts of C_1 whose negation implies C_2. Often one can simply choose the guard of the conditional statement which caused the SET to branch.

Proposition 1. *The "If-Then-Else Merge" technique induces a family of join-semilattices of SE states, i.e., the operation $\dot{\sqcup}_{ite}$ and its associated partial order relation $\preceq$ satisfy axioms (SEL1)–(SEL5) of Defn. 8.*

Example 2. Fig. 2 depicts a fragment of the join-semilattice $(S_\varphi, \dot{\sqcup}_{ite})$ induced by $\dot{\sqcup}_{ite}$ for Listing 1. The two states at the bottom of the figure correspond to the outcome of the execution until the end of the if block, where Γ represents a common set of preconditions. Since the values for both _x and _y differ in those states, the If-Then-Else construction is applied. The differing formulas in the path conditions, $\mathrm{y} < \mathrm{x}$ and $\mathrm{y} \geq \mathrm{x}$, vanish in the path condition of the merged state since their disjunction can be simplified to true.

4.3 Abstract Weakening and Predicate Abstraction

Our General Lattice Framework, along with the state merging technique proposed below, at least partially closes the gap between symbolic execution and abstract interpretation [11] by facilitating merges based on abstract domain lattices. We first define the notion of abstract domain elements.

Definition 11 (Abstract Domain Element). *An* Abstract Domain Element *is a function $defAx : \mathrm{Trm} \to \mathrm{Fml}$ mapping terms to* closed *formulas.*

Intuitively, an abstract domain element models an infinite set of *defining axioms* for JavaDL terms. If an axiom is true for a given term, then this term is described by the corresponding abstract domain element. This rather technical, syntactic definition is beneficial for the application in branch merging and allows for a straightforward embedding of predicate abstraction [16]. However, in contrast to predicate abstraction we allow infinite domains.

Definition 12 (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 as arguments. We impose the following requirements on A_T and $\sqsubseteq$:*
(1) A_T includes two elements with $\bot(t) = \mathsf{false}$, $\top(t) = \mathsf{true}$ for any $t \in \mathrm{Trm}$.
(2) For $a, b \in A_T$ with $a \sqsubseteq b$, $\vdash a(t) \to b(t)$ holds 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 below illustrates the above definitions in the context of predicate abstraction. As usual, we have a bottom and a top element, where the bottom element is the only one that is not satisfiable. Furthermore, for each lattice element a that is more concrete than an element b ($a \sqsubseteq b$), also the defining axiom of a has to be stronger than that of b. Now we are in a position to generalize the "If-Then-Else Merge" technique: instead of using conditional terms for the result of $\mathsf{joinVal}\,(s_1, s_2; \mathrm{x}, c_\mathrm{X})$ as in Defn. 10, we compute a sound abstraction of the SE states to be merged. Technically, we employ the symbols c_X and constrain them by defining axioms computed from a suitable join in the join semi-lattice.

Definition 13 (Abstract Weakening Merge Method). *Let $\mathcal{A}_T = (A_T, \sqcup)$ be an abstract domain lattice. Given two SE states $s_j = (U_j, C_j, \varphi)$, $j = 1, 2$, the join operation $\dot{\sqcup}_{abstr}$ is defined by*

$$\mathsf{joinVal}\,(s_1, s_2; \mathrm{x}, c_\mathrm{X}) \coloneqq c_\mathrm{X} \qquad \mathsf{constraints}\,(s_1, s_2; \mathrm{x}, c_\mathrm{X}) \coloneqq (\mathit{defAx}_1 \sqcup \mathit{defAx}_2)\,(c_\mathrm{X})$$

where, for $k \in 1, 2$, $\mathit{defAx}_k \in A_T$ are abstract domain elements such that $C_k \to \mathit{defAx}_k\,(\{U_k\}\,\mathrm{x})$ is provable and there is no element $\mathit{defAx}'_k \in A_T$ with $\mathit{defAx}'_k \neq \mathit{defAx}_k$ and $\mathit{defAx}'_k \sqsubseteq \mathit{defAx}_k$.

The constraints on defAx_k state that they must be contained in the abstract domain lattice. There is not necessarily a *unique* element such that $C_k \to \mathit{defAx}_k\,(\{U_k\}\,\mathrm{x}_i)$ is provable. Any element for which there is no strictly smaller one suffices. For countable abstract lattices with an enumerable linearization, the functions defAx_k are computable, in particular, for finite domains an enumeration is obtained by topological sorting. For the sign analysis domain, one enumeration is $\bot, -, 0, +, \top$. Generally, infinite domains should support *widening* [11] to ensure that suitable abstractions can be computed.

In Defn. 13 we consider lattices with a uniform type. It is possible to use different lattices for different types in the merge technique. When no lattice is specified for some type, If-Then-Else merges are used as fallback. Depending on the situation, it may also be appropriate to define multiple lattices for the same type (see Example 3 and Fig. 3 for a concrete example for $\dot{\sqcup}_{abstr}$).

Proposition 2. *The abstract weakening merge method 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)–(SEL5) of Defn. 8.*

Predicate abstraction [16] is an instance of abstract weakening where the domain elements are constructed from combinations of a given finite set of unary predicates. The following example defines a domain for predicate abstraction that captures relations between program variables.

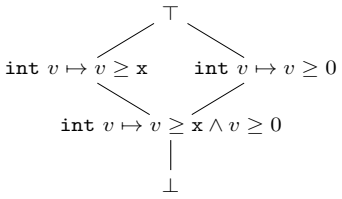

Fig. 3. Abstract domain for Example 3

Example 3 (Predicate Abstraction as Abstract Domain). Consider Listing 1. To prove that the result is non-negative, we need after the merge (line 7) the fact

Listing 2. `abs` example

```
1 public int abs(int num) {
2     int result;
3     if (num < 0) { result = −num; }
4     else { result = num; }
5     return result;
6 }
```

Listing 3. `posSum` example

```
1 public int posSum(int x, int y) {
2     while (x > 0) { y++; x−−; }
3     return abs(y);
4 }
```

Listing 4. `gcd` example

```
1  public static int gcd(int a, int b) {
2      if (a < 0) a = −a;
3      if (b < 0) b = −b;
4      int big, small;
5      if (a > b) {
6          big = a;
7          small = b;
8      } else {
9          big = b;
10         small = a;
11     }
12     return gcdHelp(big, small);
13 }
```

that the value of y is not smaller than the value of x. To capture this relation among the variables, we choose as abstraction predicates $v \geq \mathtt{x}$ and $v \geq 0$, where v is a placeholder for the input term. The resulting abstract domain is built from the conjunctions of all subsets of those predicates, see Fig. 3.

State Merging with Join-Semilattices The following theorem establishes the correctness of state merges with induced join-semilattices in the course of symbolic execution. We omit the proof for space reasons, and refer the reader to [22].

Theorem 1 (Correctness of Merging with Induced Join-Semilattices). *Let $\varphi \in \mathrm{Fml}$ be a formula and $\mathbb{L}_\varphi$ an induced join-semilattice for a join operation $\dot{\sqcup}$. Then, merging two SE states $s_i = (U_i, C_i, \varphi)$, $i = 1, 2$, to a state $s^* = s_1 \mathbin{\dot{\sqcup}} s_2$ is* sound*, i.e. if s^* is valid, then both s_1 and s_2 are valid.*

Example 4 (Continuation of Example 3). After symbolic execution of the conditional statement, we are left with two states that have identical program counters. So we can merge them using the abstraction predicates of Example 3 and end up in a single (valid) SE state as shown in Fig. 4.

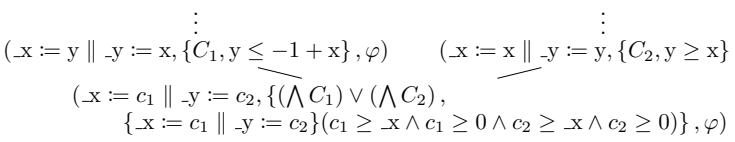

Fig. 4. Example: Merging by Predicate Abstraction

5 Evaluation

To assess the efficacy of our state merging methods, we implemented them in the KeY verification system and applied them on a micro benchmark suite consisting of four Java programs. We also present the results of a highly complex case study on the TimSort method [15], which has been redone using our implementation.

Example	# Rule Apps w/o merge	# Rule Apps with merge	Diff. (%)	#Merges	Merge Techn.	Abstr. Predicates
dist	219	254	-15.98 %	1	ITE	–
dist	219	206	5.94 %	1	PRED (conj)	$\{v \geq 0, v \leq \mathrm{y}\}$
abs	156	137	12.18 %	1	ITE	–
abs	156	132	15.38 %	1	PRED (disj)	$\{v > 0, v = 0, v < 0\}$
gcd	9,056	8,758	3.29 %	2	ITE	–
gcd	9,056	7,591	16.18 %	2	PRED (conj)	$\{v \geq 0, (v = a \vee v = -a)\}$ $\{v \geq 0, (v = b \vee v = -b)\}$
posSum	1,422	926	34.88 %	4	ITE	–
posSum	1,422	911	35.94 %	4	PRED	$\{v = x + y\}$

PRED (conj/disj): predicate abstraction with conjunctions/disjunctions of the predicates
ITE : the If-Then-Else merge technique.

Table 3. Micro benchmark results

5.1 Micro Benchmarks

Our micro benchmarks comprise the `dist` method (→ Listing 1), method `abs` (→ Listing 2) computing the absolute of a given integer parameter, method `gcd` (→ Listing 4) computing the Greatest Common Divisor (GCD) of two integers, and method `posSum` computing the absolute of the sum of two positive integers (→ Listing 3). In the `dist` example, the SE states after the execution of the if statement are suitable for merging. For `abs`, where the proof goal is to show that the result is positive, we use state merging after the execution of the if block before Line 5. In the case of `gcd`, we aim to prove that the returned result is actually the GCD of the input; state merging techniques are applied after Lines 2 and 3. Method `posSum` demonstrates the application of state merging for a while loop. Our goal is to prove that the returned result is the absolute of the sum of the inputs. To render the SET finite, we constrain the value of x by the upper bound 5. Thus, the loop is unwound five times during SE, giving the opportunity of four merges before the call to the method `abs` in Line 3.

For each example, we compare the number of rule applications in a proof without merging to the corresponding number in a proof containing merge rule applications on the basis of the If-Then-Else as well as the predicate abstraction technique. Results are shown in Table 3. In the last column, the predicates used for abstraction are listed; v is a placeholder for an input term of type `int`. The choice for `abs` induces a standard abstract domain for sign analysis of integers; in the other cases, the predicates are tailored to the specific situations.

The result for `dist` demonstrates that If-Then-Else merging can even increase the proof size when states are merged close to the end of SE. Merging with predicate abstraction was beneficial in all cases. However, If-Then-Else merging is easy to automate, whereas it is a harder problem to automatically infer abstraction predicates. Furthermore, the TimSort case study affirms that If-Then-Else merges can substantially decrease the sizes of larger proofs.

5.2 TimSort

In 2015, de Gouw et al. [15] discovered a bug in the TimSort implementation of the JDK library, Java's default sorting routine. The bug triggered, under certain

Method	#Rule Apps (in [1])	#Rule Apps (with Merging)	#Merges	Percentage Changes with State Merging
ensuresCapacity	44,346	50,707	1	-14%
ensuresCapacity*	44,346	37,815	1	15%
mergeAt	279,155	63,309	6	77%
gallopLeft	303,716	88,332	6	71%
sort(a,lo,hi,c)	235,632	152,752	1	35%
mergeHi	N/A	460,409	5	NaN

*) Proof by authors of this paper, uses predicate abstraction rather than If-Then-Else.

Table 4. Statistics comparing proofs with and without state merging

circumstances, an uncaught exception. The authors fixed the bug and proved its absence as well as that of any other uncaught exception. An extended journal version of [15] is currently under preparation, where all verification proofs are being redone using the state merging approach presented in this paper. De Gouw et al. kindly allowed us to include their current results as part of our evaluation.[4] Table 4 provides a comparison of the proof sizes with and without merging. It shows that the proof sizes improved significantly for most proofs. All merges used, if not stated otherwise, the If-Then-Else technique and thus required no expert knowledge. In particular, state merging allowed to verify the method mergeHi which was out of reach in [15] due to the path explosion problem.

For `ensuresCapacity`, where merging with If-Then-Else actually increased the proof size, we created a new proof using a merge based on predicate abstraction. The resulting proof size is 15% smaller compared to the version without merging and even 25% smaller than the proof with If-Then-Else based merging.

6 Lessons Learned and Future Work

The proposed state merging approach transforms an SET into a connected and rooted Directed Acyclic Graph (DAG). Changing the underlying data structures in a complex verification system such as KeY would be a substantial undertaking. We implemented a different solution by adding the new merge node as a child to *only one of* the parents and *linking* the second parent to it. Our implementation ensures that, if the subtree below a merge node is closed (or the merge node is pruned away), then the linked node is also closed (or "unlinked").

It is important to automate state merging as much as possible, in particular for less complex verification tasks that are otherwise fully automatic. To help this, we extended the specification language JML with the annotation */*@ *merge_proc* <*join_operator*> @*/*. It is placed in front of a Java block after which the merge is supposed to happen. For certain join operators, for example the If-Then-Else join operator, this requires much less expert knowledge than the definition of a *block contract*, i.e. an annotation of a block of statements with pre- and postconditions, as an alternative way of tackling path explosion.

In our experiments, we discovered that state merging with the If-Then-Else technique is most beneficial when applied in situations where (1) a substantial amount of code remains to be executed, and thus a lot of repetition can be avoided, and (2) the difference between the states to be merged is as small as

[4] Available at http://www.key-project.org/timsort/stats.html

possible. "Difference" means the number of variables attaining different values in the symbolic states and the number of different formulas in the path conditions.

Predicate abstraction-based state merging is applicable to a wider range of constellations. However, to come up with suitable predicates requires a certain amount of expertise. An unsuitable choice of abstraction predicate can cause the unfeasibility of the proof goal, because abstraction loses precision. At this time, to merge states with predicate abstraction is comparable in difficulty to writing block contracts. Nevertheless, we think that state merging is more suitable for automation, because it can be performed on-the-fly *during* the proof process. Future work will aim at integrating heuristic approaches to improve the performance of If-Then-Else state merging [21] as well as methods developed for specification generation to automatically infer abstraction predicates [17,25].

7 Related Work and Conclusion

Existing work on state merging in symbolic execution employs If-Then-Else based techniques [2,18,21,23] or addresses the automatic generation of loop invariants by the means of abstraction [6,25]. Kuznetsov et al. [21] try to assess the "cost-benefit ratio" of If-Then-Else based merges by heuristically trading off the reduction of states against the complexity of the resulting expressions. Bubel et al. [6] use value abstraction and Weiß et al. [25] use predicate abstraction for merging states in the course of the automatic generation of loop invariants.

In contrast to previous work, our approach is not limited to a particular state merging technique. We devised a general lattice-based framework for join operations and proved soundness of a state merging rule for join operations conforming to our framework. The two most popular state merging techniques in the literature, If-Then-Else and predicate abstraction, are instances of our framework. Our implementation is based on the state-of-the-art verification system KeY [1]. It has been extensively evaluated with the highly complex TimSort case study and it was demonstrated that significant improvements can be gained. This led to proofs that were out of reach before.

Acknowledgment We would like to thank the authors of [15] for the permission to quote data from the extended journal version of their paper under preparation.

References

1. Ahrendt, W., Beckert, B., et al.: The KeY Platform for Verification and Analysis of Java Programs. In: Giannakopoulou, D., Kroening, D. (eds.) 6th Working Conf. on Verified Software: Theories, Tools, and Experiments. Springer (2014)
2. Anand, S., Godefroid, P., et al.: Demand-Driven Compositional Symbolic Execution. In: Proc. of the 14th Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems. pp. 367–381. Springer (2008)
3. Anand, S., Păsăreanu, C.S., et al.: Symbolic Execution with Abstract Subsumption Checking. In: 13th Intl. Conf. on Model Checking Software. Springer (2006)
4. Beckert, B., Hähnle, R.: Reasoning and Verification. IEEE Intelligent Systems 29(1), 20–29 (2014)

5. Beckert, B., Hähnle, R., et al. (eds.): Verification of Object-Oriented Software: The KeY Approach. Springer (2006)
6. Bubel, R., Hähnle, R., et al.: Abstract Interpretation of Symbolic Execution with Explicit State Updates. In: de Boer, F., Bonsangue, M.M., et al. (eds.) 6th Intl. Symp. on Formal Methods for Components and Objects. Springer (2009)
7. Burstall, R.M.: Program Proving as Hand Simulation with a Little Induction. In: Information Processing, pp. 308–312. Elsevier (1974)
8. Cadar, C., Sen, K.: Symbolic Execution for Software Testing: Three Decades Later. Communications of the ACM 56(2), 82–90 (2013)
9. Chu, D., Jaffar, J., et al.: Lazy Symbolic Execution for Enhanced Learning. In: Bonakdarpour, B., Smolka, S.A. (eds.) Proc. of the 5th Intl. Conf. on Runtime Verification. pp. 323–339. Springer (2014)
10. Clarke, E.M., Grumberg, O., et al.: Model Checking. The MIT Press (1999)
11. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: 4th Symp. of POPL. pp. 238–252. ACM Press (Jan 1977)
12. Fitting, M.C.: First-Order Logic and Automated Theorem Proving. Springer, second edition edn. (1996)
13. Fitting, M.C., Mendelsohn, R.: First-Order Modal Logic. Kluwer (1998)
14. Gosling, J., Joy, B., et al.: The Java (TM) Language Specification. Addison-Wesley Professional, 3rd edn. (2005), `http://psc.informatik.uni-jena.de/languages/Java/javaspec-3.pdf`
15. Gouw, S.d., Rot, J., et al.: OpenJDK's Java.utils.Collection.sort() Is Broken: The Good, the Bad and the Worst Case. In: Kroening, D., Pasareanu, C.S. (eds.) Proc. of the 27th Intl. Conf. on Computer Aided Verification. Springer (2015)
16. Graf, S., Saidi, H.: Construction of Abstract State Graphs with PVS. In: Grumberg, O. (ed.) Proc. of the 9th Intl. Conf. on Computer Aided Verification. pp. 72–83. Springer (1997)
17. Hähnle, R., Wasser, N., et al.: Array Abstraction with Symbolic Pivots. In: Ábrahám, E., Bonsangue, M., et al. (eds.) Theory and Practice of Formal Methods. Springer (2016)
18. Hansen, T., Schachte, P., et al.: State Joining and Splitting for the Symbolic Execution of Binaries. In: Bensalem, S., Peled, D.A. (eds.) Proc. of the 9th Intl. Workshop on Runtime Verification. pp. 76–92. Springer (2009)
19. Harel, D., Tiuryn, J., et al.: Dynamic Logic. MIT Press, Cambridge, MA, USA (2000)
20. King, J.C.: Symbolic Execution and Program Testing. Communications of the ACM 19(7), 385–394 (1976)
21. Kuznetsov, V., Kinder, J., et al.: Efficient State Merging in Symbolic Execution. In: Proc. of the 33rd Conf. on PLDI. pp. 193–204. ACM (2012)
22. Scheurer, D.: From Trees to DAGs: A General Lattice Model for Symbolic Execution. Master's thesis, Technische Universität Darmstadt (2015), `http://tinyurl.com/Trees2DAGs`
23. Sen, K., Necula, G., et al.: MultiSE: Multi-Path Symbolic Execution using Value Summaries. In: 10th Joint Meeting on Foundations of Software Engineering. pp. 842–853. ACM (2015)
24. Shoenfield, J.R.: Mathematical logic. Addison-Wesley (1967)
25. Weiß, B.: Predicate Abstraction in a Program Logic Calculus. In: Leuschel, M., Wehrheim, H. (eds.) Proc. of the I7th Intl. Conf. on Integrated Formal Methods. pp. 136–150. Springer (2009)