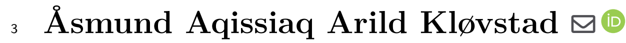
Compositional Correctness and Completeness for Symbolic Partial Order Reduction


Åsmund Aqissiaq Arild Kløvstad
University of Oslo, Norway

Eduard Kamburjan
University of Oslo, Norway

Einar Broch Johnsen
University of Oslo, Norway

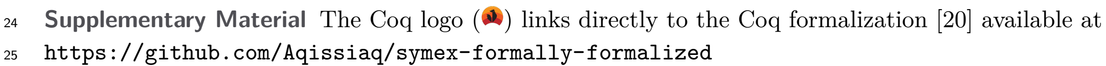
Abstract

Partial Order Reduction (POR) and Symbolic Execution (SE) are two fundamental abstraction techniques in program analysis. SE is particularly useful as a state abstraction technique for sequential programs, while POR addresses equivalent interleavings in the execution of concurrent programs. Recently, several promising connections between these two approaches have been investigated, which result in *symbolic partial order reduction*: partial order reduction of symbolically executed programs. In this work, we provide *compositional* notions of completeness and correctness for symbolic partial order reduction. We formalize completeness and correctness for (1) abstraction over program states and (2) trace equivalence, such that the abstraction gives rise to a complete and correct SE, the trace equivalence gives rise to a complete and correct POR, and their combination results in complete and correct symbolic partial order reduction. We develop our results for a core parallel imperative programming language and mechanize the proofs in Coq.

2012 ACM Subject Classification Theory of computation → Parallel computing models

Keywords and phrases Symbolic Execution, Coq, Trace Semantics, Partial Order Reduction

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2023.23


Supplementary Material The Coq logo links directly to the Coq formalization [20] available at `https://github.com/Aqissiaq/symex-formally-formalized`

Acknowledgements The first author would like to thank Yannick Zakowski for help with Coq formatting and Erik Voogd for valuable insights on symbolic semantics.

1 Introduction

Program analyses rely on representing the possible reachable states and traces of a program run efficiently and are commonly accompanied by a correctness theorem (*all representable states and traces are reachable*) and possibly a completeness theorem (*all reachable states and traces are represented*). Explicitly listing all states or traces leads to the "state space explosion", as even for simple programs, the number of possible program states may grow so fast that examining them all explicitly becomes infeasible.

One source of this growth is the domain of data — the number of possible values is very large, even for a single integer. Symbolic execution [7, 18, 19] (SE) mitigates this problem by representing values symbolically, thus covering many possible concrete states at once. SE is utilized to great effect in program analysis [3]. Another source of growth is *concurrency*, as the number of possible interleavings grows exponentially. Partial Order Reduction (POR) is a technique for tackling this explosion by taking advantage of the fact that independent events can be reordered without affecting the final result [16].

© Åsmund Aqissiaq Arild Kløvstad, Eduard Kamburjan, Einar Broch Johnsen;
licensed under Creative Commons License CC-BY 4.0

The combined use of both POR and SE has recently begun to be investigated [6, 29], called *symbolic partial order reduction (SPOR)*. Notions of correctness and completeness are available for both SE and POR, but how these notions can be composed to obtain correctness and completeness of SPOR remains an open challenge. In this paper, we tackle this challenge and give a compositional notion of correctness and completeness for SPOR, based on the abstraction and equivalence notions that define SE and POR. To formulate such a theory we use *trace*-based semantics. Trace semantics is both expressive [24, 32] and compositional [11], and allows a natural formulation of partial order reduction [6].

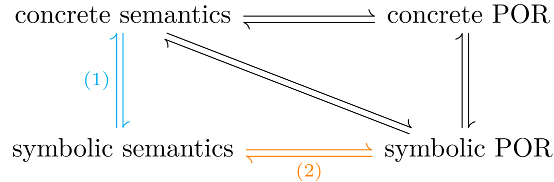


Figure 1 State of the art and our contribution

State of the Art. Figure 1 shows the available correctness and completeness results for SE and for POR. Each corner denotes a program semantics, and the arrows denote correctness and completeness. First, let us examine the left side of the square, which is concerned with SE.

The left edge of Figure 1, labeled (1), is provided by de Boer and Bonsangue [5], who define symbolic and concrete semantics for several minimal imperative languages to formulate and prove notions of correctness and completeness for SE. However, their work is limited to a *sequential setting*. The proof is based on using a suitable abstraction between concrete and symbolic states, that defines the SE.

The bottom edge of Figure 1, labeled (2), is studied by de Boer et al. [6], who formulate partial order reduction for symbolic execution with explicit threads using a syntactic notion of interference freedom and implement this approach in the rewriting logic framework Maude [10]. Their results are not connected to the concrete semantics. The result is based on an equivalence relation between symbolic traces, that defines the SPOR, but does not use an explicit abstraction between states. We discuss further, related results on symbolic execution in Sec. 6.

The top of Figure 1 concerns POR [1, 13, 16, 26] for concrete executions, where numerous implementations are available. The correctness of such a reduction corresponds to the top edge of Figure 1, though it is not usually presented in terms of an equivalence relation as proposed by de Boer et al. Results directly of SPOR are given by Schemmel et al. [29], who apply (dynamic) partial order reduction to symbolic execution using “unfolding” to explore paths. This shows that POR is applicable directly to SE, but does not discuss a generic notion of state abstraction and trace equivalence.

While all four corners of Figure 1 are well established, and several edges have been explored, there exists no general formalization of the properties for state abstraction and trace equivalence needed for a uniform and compositional treatment of different POR algorithms and SE techniques. Hence, the present work unifies notions of correctness and completeness for symbolic execution and partial order reduction, and fills in the remaining (black) edges of Figure 1. By compositional completeness and correctness, we mean that the diagonal follows automatically from the other edges of the figure.

Approach. To fill the gap we formulate concrete and symbolic trace semantics for a

small imperative language with parallel composition and show that these semantics enjoy a bisimulation relationship. We then formulate partial order reduction in terms of an equivalence relation on traces, and show that this also leads to a bisimulation of reduced and non-reduced semantics. These bisimulations extend to correctness and completeness results, and compose naturally to semantics with *both* symbolic execution and partial order reduction.

The results are obtained in a framework extending the work of de Boer et al. and are centered around the notions of state abstraction and trace equivalence. Following de Boer et al., state abstraction is given by transforming concrete states according to symbolic states, and a concrete state is abstracted if it can be obtained by some symbolic transformation. Trace equivalence defines an equivalence relation on sequences of events which allows for partial order reduction. In particular, it suffices to explore one trace per equivalence class.

Both symbolic and concrete states are implemented by total functions of variable names with generic properties. To reduce the number of rules and allow for elegant parallel composition the semantics are given by a reduction system in the style of Felleisen and Hieb [12] with contexts formalized as functions on statements and an inductive relation [21]. The full semantics are obtained by stepwise transitive closure, which allows for proofs by induction and case analysis of the final step.

Contributions. Our contribution is threefold.

1. We unify and fill in the remaining edges in the above diagram. In particular we give correctness and completeness relations for concrete partial order reduction, directly relate partial order reduction in the symbolic and concrete case, and compose the results to relate concrete semantics to reduced symbolic semantics.
2. Correctness and completeness for both symbolic execution and partial order reduction are formulated in a parametric fashion, allowing for different implementations of both, providing they fulfill certain conditions.
3. Finally, the entire development is mechanized in Coq [4, 33]. This lends credence to the results and allows for extensions and further work in a systematic manner.

Structure. Section 2 introduces basic notions for symbolic execution with trace semantics for a basic imperative language with parallel composition. Then both concrete and symbolic semantics are given as reduction systems with contexts to handle both sequential and parallel composition. Finally we formulate and prove correctness and completeness of the symbolic semantics with respect to the concrete semantics. Section 3 introduces a notion of trace equivalence that connects correctness and completeness to partial order reduction, which is used in Section 4 to define independence of events in a semantic manner. We then define new PO-reduced semantics for both symbolic and concrete cases, and show that they bisimulate their non-reduced counterparts. Finally, Section 5 connects previous results and shows that bisimulation carries through POR to fill in the upper right half of the diagram. Section 6 and 7 give further related work and concludes.

2 Symbolic Trace Semantics

In this section we introduce the basic notions of our framework. In particular, we define a small imperative language with parallel composition and formulate symbolic and concrete trace semantics for it. We relate the two semantics by a bisimulation defining both trace completeness and trace correctness.

$$
\begin{array}{lr}
e ::= n \mid x \mid e_1 + e_2 & \text{arith. expr.} \\
b ::= \text{true} \mid \text{false} \mid \neg b \mid b_1 \wedge b_2 \mid e_1 \leq e_2 & \text{bool. expr.} \\
s ::= x := e \mid s_1 \,;\, s_2 \mid s_1 \parallel s_2 \mid \texttt{if } b \, \{s_1\}\{s_2\} \mid \texttt{while } b \, \{s\} \mid \texttt{skip} & \text{statements}
\end{array}
$$

Figure 2 Grammar for expressions and statements

126
2.1 Basic Notions

127 For the basic setup we assume a set of program variables *Var*, a set of arithmetic expressions
128 *Aexpr* and a set of Boolean expressions *Bexpr*. Our basic programming language is an
129 imperative language with (side effect free) assignment, conditional branching, iteration and
130 both sequential and parallel composition.

131 ▶ **Definition 2.1** (Syntax). *The sets of arithmetic expressions Aexpr, Boolean expressions*
132 *Bexpr, and statements Stmt are defined by the grammar in Figure 2, where we let x range*
133 *over Var, n over $\mathbb{N}$, b over Bexpr, e over Aexpr and s over statements.*

134 Before we define the semantics, we require a notion of store to express program state. We
135 distinguish between symbolic stores, for symbolic execution, and concrete stores, for concrete
136 execution.

137 ▶ **Definition 2.2** (Symbolic Store). *A symbolic store σ is a* substitution*, i.e., a map from*
138 *Var to Aexpr denoted by σ.*

139 We take equality of substitutions to be extensional, that is $\sigma = \sigma'$ if $\sigma(x) = \sigma'(x)$ for all
140 x. An *update* to a substitution is denoted by $\sigma[x := e]$. A substitution can be recursively
141 applied to a Boolean or arithmetic expression, resulting in a new expression. We denote such
142 an application by $e\sigma$.

143 ▶ **Definition 2.3** (Concrete Store). *A concrete store V is a* valuation*, i.e., a map from Var*
144 *to $\mathbb{N}$ denoted by V.*

145 Like substitutions, valuations can be updated (denoted $V[x := n]$) and a valuation can be
146 used to evaluate an expression. This evaluation is denoted $V(e)$ and results in a natural
147 number for arithmetic expressions and a Boolean for Boolean expressions. For a Boolean
148 expression b, we say V is a model of b if $V(b) = \text{true}$ and denote this by $V \models b$. The
149 definitions of substitution and evaluation are standard and given in the auxiliary material.

150
2.2 Trace Semantics

151 Based on the notion of symbolic and concrete stores, we now give the symbolic and concrete
152 semantics. Both semantics are based on traces, i.e., sequences of *events*. Events are
153 assignments or guards in the symbolic case, or just assignments in the concrete case.

154 ▶ **Definition 2.4** (Symbolic Trace). *A symbolic trace is a sequence of conditions or symbolic*
155 *assignments defined by the grammar*

156 $$\tau_S ::= [\,] \mid \tau_S :: (x := e) \mid \tau_S :: b$$

157 ▶ **Definition 2.5** (Concrete Trace). *A concrete trace is a sequence of concrete assignments*
158 *defined by the grammar*

159 $$\tau_C ::= [\,] \mid \tau_C :: (x := e)$$

In both cases $[\,]$ denotes the empty trace and we write the trace $[\,] :: x :: y :: z \ldots$ simply as $[x, y, z \ldots]$. The concatenation of τ and τ' is denoted by $\tau \cdot \tau'$. The trace syntax is shared between symbolic and concrete traces, but the difference will be clear from context.

We represent the current program state as a pair of a statement (the program remaining to be executed) and the trace generated so far. Evaluating expressions requires to evaluate the expression in the last substitution or valuation of the trace. To do so, we extract this *final* substitution or valuation from a trace and an initial substitution or valuation by folding over the trace. In the case of a symbolic trace, the result is a symbolic substitution, while a concrete trace results in a concrete valuation.

▶ **Definition 2.6** (Final Substitution). *Given an initial substitution σ, the final substitution of a trace τ_S is denoted $\tau_S \Downarrow_\sigma$ and inductively defined by*

$$\begin{aligned} [\,] \Downarrow_\sigma &= \sigma \\ \tau_S :: b \Downarrow_\sigma &= \tau_S \Downarrow_\sigma \\ \tau_S :: (x := e) \Downarrow_\sigma &= \sigma'[x := (e\sigma')] \text{ where } \sigma' = \tau_S \Downarrow_\sigma \end{aligned}$$

When $\sigma = id$ we omit it and write $\tau_S \Downarrow$

▶ **Definition 2.7** (Final Valuation). *Given an initial valuation V, the final valuation of a trace τ_C is denoted $\tau_C \Downarrow_V$ and inductively defined by*

$$\begin{aligned} [\,] \Downarrow_V &= V \\ \tau_C :: (x := e) \Downarrow_V &= V'[x := V'(e)] \text{ where } V' = \tau_C \Downarrow_V \end{aligned}$$

Semantics can then be given by a simple reduction relation on atomic statements (Figure 3), which extends to the full language by S/C-IN-CONTEXT. The symbolic (resp. concrete) relation works on pairs of statements and symbolic (resp. concrete) traces to extend them with appropriate events.

▶ **Definition 2.8** (Symbolic and Concrete Semantics). *The symbolic semantics $\rightarrow$ between two symbolic configurations is given on the left of Fig. 3. The concrete semantics $\Rightarrow$ between two concrete configurations is given on the right of Fig. 3.*

Both semantics are straightforward, we point out three details. First, the main difference is that the rules with branching ($*$-IF-T, $*$-IF-F, $*$-WHILE-T, $*$-WHILE-F) are non-deterministic and add an event in the symbolic case, but are deterministic in the concrete case.

Second, in order to concisely deal with both sequential and parallel composition, we use *contexts* [12]. A context C represents a statement with a "hole" ($\square$) in it and is generated by the grammar:

$$C ::= \square \mid (C \; ; \; s) \mid (C \parallel s) \mid (s \parallel C)$$

Intuitively, the statement we are interested in may occur on its own, sequentially before some other statement, or on either side of a parallel operator. By $C[s]$ we denote the statement s in the hole in context C.

Finally, we point out that we model termination by reduction to `skip`.

▶ **Example 2.9.** Consider the program $s = \texttt{y} := \texttt{1} \parallel \texttt{x} := \texttt{3} \parallel \texttt{if X} \leq \texttt{1} \; \{\texttt{Y} := \texttt{2}\} \; \{\texttt{Y} := \texttt{3}\}$. We will show that $(s, [\,]) \rightarrow^* (\texttt{skip}, [x := 3, y := 1, x > 1, y := 3])$. In other words that $[x := 3, y := 1, x > 1, y := 3]$ is one possible trace of the program.

$$\text{S-ASGN}\ \frac{}{(\mathtt{x := e}, \tau) \rightsquigarrow (\mathtt{skip}, \tau :: (x := e))} \qquad \frac{}{(\mathtt{x := e}, \tau) \rightsquigarrow_V (\mathtt{skip}, \tau :: (x := e))}\ \text{C-ASGN}$$

$$\text{S-IF-T}\ \frac{}{(\mathtt{if\ b}\ \{\mathtt{s_1}\}\{\mathtt{s_2}\}, \tau) \rightsquigarrow (\mathtt{s_1}, \tau :: b)} \qquad \frac{\tau \Downarrow_V (b) = \text{true}}{(\mathtt{if\ b}\ \{\mathtt{s_1}\}\{\mathtt{s_2}\}, \tau) \rightsquigarrow_V (\mathtt{s_1}, \tau)}\ \text{C-IF-T}$$

$$\text{S-IF-F}\ \frac{}{(\mathtt{if\ b}\ \{\mathtt{s_1}\}\{\mathtt{s_2}\}, \tau) \rightsquigarrow (\mathtt{s_2}, \tau :: \neg b)} \qquad \frac{\tau \Downarrow_V (b) = \text{false}}{(\mathtt{if\ b}\ \{\mathtt{s_1}\}\{\mathtt{s_2}\}, \tau) \rightsquigarrow_V (\mathtt{s_2}, \tau)}\ \text{C-IF-F}$$

$$\text{S-WHILE-T}\ \frac{}{(\mathtt{while\ b}\ \{\mathtt{s}\}, \tau) \rightsquigarrow (\mathtt{s\ ;\ while\ b}\ \{\mathtt{s}\}, \tau :: b)} \qquad \frac{\tau \Downarrow_V (b) = \text{true}}{(\mathtt{while\ b}\ \{\mathtt{s}\}, \tau) \rightsquigarrow_V (\mathtt{s\ ;\ while\ b}\ \{\mathtt{s}\}, \tau)}\ \text{C-WHILE-T}$$

$$\text{S-WHILE-F}\ \frac{}{(\mathtt{while\ b}\ \{\mathtt{s}\}, \tau) \rightsquigarrow (\mathtt{skip}, \tau :: \neg b)} \qquad \frac{\tau \Downarrow_V (b) = \text{false}}{(\mathtt{while\ b}\ \{\mathtt{s}\}, \tau) \rightsquigarrow_V (\mathtt{skip}, \tau)}\ \text{C-WHILE-F}$$

$$\text{S-SEQ}\ \frac{}{(\mathtt{skip\ ;\ s}, \tau) \rightsquigarrow (\mathtt{s}, \tau)} \qquad \frac{}{(\mathtt{skip\ ;\ s}, \tau) \rightsquigarrow_V (\mathtt{s}, \tau)}\ \text{C-SEQ}$$

$$\text{S-PAR}\ \frac{}{(\mathtt{skip} \parallel \mathtt{skip}, \tau) \rightsquigarrow (\mathtt{skip}, \tau)} \qquad \frac{}{(\mathtt{skip} \parallel \mathtt{skip}, \tau) \rightsquigarrow_V (\mathtt{skip}, \tau)}\ \text{C-PAR}$$

$$\text{S-IN-CONTEXT}\ \frac{(s, \tau) \rightsquigarrow (s', \tau')}{(C[s], \tau) \to (C[s'], \tau')} \qquad \frac{(s, \tau) \rightsquigarrow_V (s', \tau')}{(C[s], \tau) \Rightarrow_V (C[s'], \tau')}\ \text{C-IN-CONTEXT}$$

Figure 3 Reduction rules for symbolic and concrete semantics

202 First apply S-IN-CONTEXT with $C = \mathtt{y := 1} \parallel \square \parallel \mathtt{if\ X \leq 1\ \{Y := 2\}\ \{Y := 3\}}$ and S-ASGN
203 to obtain

204 $$(s, [\,]) \to (\mathtt{y := 1} \parallel \mathtt{skip} \parallel \mathtt{if\ X \leq 1\ \{Y := 2\}\ \{Y := 3\}}, [x := 3])$$

205 The second assignment is similar, followed by S-IF-F in the context $\mathtt{skip} \parallel \mathtt{skip} \parallel \square$ to obtain

206 $$(\mathtt{skip} \parallel \mathtt{skip} \parallel \mathtt{if\ X \leq 1\ \{Y := 2\}\ \{Y := 3\}}, [x := 3, y := 1]) \to (\mathtt{skip} \parallel \mathtt{skip} \parallel \mathtt{Y := 3}, [x := 3, y := 1, x > 1])$$

207 After the last assignment, the superfluous $\mathtt{skip}$s are dispensed with by S-PAR and putting
208 the steps in sequence gives the desired

209 $$(s, [\,]) \to^* (\mathtt{skip}, [x := 2, y := x, z := x])$$

210 Note that we could choose to apply the contexts in a different order, resulting in five
211 other potential traces.

212 2.3 Correctness and Completeness

213 The value of symbolic execution comes from its ability to simultaneously capture many
214 possible concrete execution paths. However, not all of these paths will be feasible for all initial
215 valuations. The feasibility of any particular symbolic trace depends on its *path condition* —
216 a conjunction of guards that allow execution to follow down this particular path — which is
217 computed in a similar fashion to final substitutions.

218 ▶ **Definition 2.10** (Path Condition). *The path condition of a symbolic trace τ_S is denoted*
219 *$pc(\tau_S)$ and defined by*

$$\begin{aligned} pc([\,]) &= \text{true} && (220)\\ pc(\tau_S :: b) &= pc(\tau_S) \wedge b(\tau_S \Downarrow) && (221)\\ pc(\tau_S :: (x := e)) &= pc(\tau_S) && (222/223) \end{aligned}$$

Because it is a conjunction of terms, once a path condition becomes false, it cannot become true again. The following lemma captures the contrapositive: a model of a trace's path condition is also a model of any prefix's path condition.

▶ **Lemma 2.11** (Path Condition Monotonicity). *If $V \models pc(\tau :: ev)$, then $V \models pc(\tau)$*

To relate the symbolic and concrete traces we define a notion of abstraction based on the correctness and completeness relations of de Boer and Bonsangue.

▶ **Definition 2.12** (Trace abstraction [5]). *Given an initial valuation V, a symbolic trace τ_S and a concrete trace τ_C we say τ_S V-abstracts τ_C if $V \models pc(\tau_S)$ and $\tau_C \Downarrow_V = V \circ \tau_S \Downarrow$*

The steps of the symbolic and concrete systems correspond very closely. Every concrete step corresponds to a symbolic step whose path condition is satisfiable, and every symbolic step with a satisfiable path condition corresponds to a concrete step. In both cases the resulting final states are related by simple composition. This relationship is formalized in the following bisimulation result.

▶ **Theorem 2.13** (Bisimulation). *For any initial valuation V and initial traces τ_0, τ_0' such that τ_0 V-abstracts τ_0':*

- *if there is a concrete step $(s, \tau_0) \Rightarrow_V (s', \tau)$, then there exists a symbolic step $(s, \tau_0') \rightarrow (s', \tau')$ such that τ' V-abstracts τ, and*
- *if there is a symbolic step $(s, \tau_0') \rightarrow (s', \tau')$ and $V \models pc(\tau')$, then there exists a concrete step $(s, \tau_0) \Rightarrow_V (s', \tau)$ such that $\tau \Downarrow_V = V \circ \tau' \Downarrow$*

By induction over the transitive closure and Lemma 2.11 we obtain correctness and completeness results. Intuitively, correctness means that each symbolic execution whose path condition is satisfied by some initial valuation V corresponds to a concrete execution with the same initial valuation. Additionally its trace abstracts the concrete trace in the sense that the final concrete state is the concretization of V by the final symbolic state. In other words the subset of states described by its path condition contains V, and there is a concrete execution corresponding to the transformation described by its final symbolic state.

▶ **Corollary 2.14** (Trace Correctness). *If $(s, \tau_S) \rightarrow^* (s', \tau_S')$, τ_S V-abstracts τ_C, and $V \models pc(\tau_S')$, then there exists a concrete trace τ_C' s.t $(s, \tau_C) \Rightarrow_V^* (s', \tau_C')$ and $\tau_C' \Downarrow_V = \tau_C \Downarrow_V \circ (\tau_S' \Downarrow)$*

Completeness captures the opposite relationship: every concrete execution has a symbolic counterpart. Furthermore the symbolic trace recovers the concrete state, and its path condition is satisfied by the initial valuation.

▶ **Corollary 2.15** (Trace Completeness). *If $(s, \tau_C) \Rightarrow_V^* (s', \tau_C')$ and τ_S V-abstracts τ_C, there exist τ_S' s.t $(s, \tau_S) \rightarrow^* (s', \tau_S')$ and τ_S' V-abstracts τ_C'.*

3 Trace Equivalence

In this section we introduce a notion of trace equivalence which will be used to formulate partial order reduction in Section 4. Intuitively two traces should be equivalent if execution could continue from either one, i.e., if partial order reduction would prune away one of them.

This is surely the case when their final states are the same. In the symbolic case their path conditions must also be equivalent. Additionally, we do not want to equate traces describing observably different behavior, so equivalent traces must contain the same events. These considerations motivate the following definition.

265 ▶ **Definition 3.1** (Symbolic Trace Equivalence). *Symbolic traces τ and τ' are equivalent*
266 *(denoted $\tau \sim \tau'$) if*
267 - τ' *is a permutation of* τ,
268 - $\tau \Downarrow_\sigma = \tau' \Downarrow_\sigma$ *for all initial substitutions* σ, *and*
269 - $V \models pc(\tau) \iff V \models pc(\tau')$ *for all valuations* V

270 ▶ **Definition 3.2** (Concrete Trace Equivalence). *Concrete traces τ and τ' are equivalent*
271 *(denoted $\tau \simeq \tau'$) if*
272 - τ' *is a permutation of* τ,
273 - $\tau \Downarrow_V = \tau' \Downarrow_V$ *for all initial valuations* V

274 ▶ **Example 3.3.** Let $\tau_1 = [y := x, z := x]$ and $\tau_2 = [z := x, y := x]$. It is both the case
275 that $\tau_1 \sim \tau_2$ and $\tau_1 \simeq \tau_2$.[1] They evidently contain the same events and have the same
276 (trivially true) path condition. Any initial substitution σ results in a final substitution
277 $\sigma'(v) = \begin{cases} x, & v \in \{y, z\} \\ \sigma(v), & \text{otherwise} \end{cases}$ and any initial valuation V results in $V'(v) = \begin{cases} V(x), & v \in \{y, z\} \\ V(v), & \text{otherwise} \end{cases}$

278 Clearly, trace equivalence defines an equivalence relation. Furthermore it allows continued
279 execution in the following sense: given a statement s and a trace τ, we can replace τ with
280 an equivalent trace τ', such that the next execution step will result in two different, but
281 equivalent traces.

282 ▶ **Lemma 3.4** (). *For equivalent traces $\tau \sim \tau'$, if $(s, \tau) \to (s', \tau_1)$ then there exists τ_2 such*
283 *that $(s, \tau') \to (s', \tau_2)$ and $\tau_1 \sim \tau_2$.*

284 This lemma also holds for concrete traces with concrete equivalence and reduction system
285 and underlies partial order reduction in both cases.
286 Crucially, the properties of trace equivalence ensure that it preserves abstraction. The
287 following theorem shows that the notion of V-abstraction carries through trace equivalence,
288 which will allow us to connect it with partial order reduction in the sequel.

289 ▶ **Theorem 3.5** (Abstraction Congruence). *For equivalent symbolic traces $\tau_S \sim \tau'_S$ and*
290 *concrete traces $\tau_C \simeq \tau'_C$, if τ_S V-abstracts τ_C then τ'_S V-abstracts τ'_C*

291 ▶ **Example 3.6.** Continuing Example 3.3, the symbolic trace τ_1 V-abstracts the concrete
292 trace τ_1 for every V, and so τ_1 also V-abstracts the equivalent concrete trace τ_2.

293 In fact, every symbolic trace V-abstracts itself viewed as a concrete trace for any V.

294 3.1 Example: Interference Freedom

295 The reordering of independent events is the core of many POR approaches. In practice true
296 independence is prohibitively expensive to compute, so some over-approximation is used.
297 Interference freedom is a syntactic over-approximation of independence of events. We show
298 that reordering interference free events is an instance of our notion of trace equivalence.
299 Interference freedom between ev_1 and ev_2 means that ev_1 does not read or write a variable
300 written by ev_2 and vice versa. Formally:

[1] Recall that symbolic traces are also concrete traces if they contain no branching events (guards).

▶ **Definition 3.7** (Interference Freedom). *Let ev be either a Boolean expression b or an assignment $(x := e)$. $R(ev)$ denotes the set of variables read by ev, ie. all the variables in b or e. $W(ev)$ denotes the set of variables written by ev, ie. x. Then ev_1, ev_2 are* interference free *iff*

$$W(ev_1) \cap W(ev_2) = R(ev_1) \cap W(ev_2) = R(ev_2) \cap W(ev_1) = \emptyset$$

Denote the interference freedom of ev_1 and ev_2 by $ev_1 \diamond ev_2$

Interference freedom is an independence relation in the sense that if $ev_1 \diamond ev_2$, then the final state of $[ev_1, ev_2]$ is equal to that of $[ev_2, ev_1]$. The reason is that interference freedom allows for "simultaneous" updates without worrying about the order of operations in the assignment case, and the variables involved in a Boolean expression can not be changed in the guard case.

On the other hand, interference freedom is an over-approximation which is perhaps most easily seen by events like $(x := x)$ and $(x \leq 3)$. Clearly they are semantically independent since the value of x does not change, but they are not interference free.

Equipped with a concrete independence relation we can construct new traces by reordering adjacent independent events. Such a reordering is captured by the equivalence define above in the sense that it results in an equivalent trace.

▶ **Theorem 3.8** (Interference free reordering is a trace equivalence). *Let $\sim_{IF}$ be the smallest equivalence relation on symbolic traces such that $\tau \cdot [ev_1, ev_2] \cdot \tau' \sim_{IF} \tau \cdot [ev_2, ev_1] \cdot \tau'$ for all τ, τ' and $ev_1 \diamond ev_2$.*

The equivalence relation $\sim_{IF}$ is contained in $\sim$.

The analogous result holds for concrete traces and $\simeq$.

This example shows that a POR scheme based on reordering of independent events is captured by trace equivalence.

4 Correctness and Completeness for Symbolic Partial Order Reduction

We formulate POR in the present setting through the use of trace equivalence (defined above) and use it to define new PO-reduced reduction systems. These new systems bisimulate the non-reduced systems of Section 2, leading directly to correctness and completeness results.

At its core, partial order reduction works by observing that some events commute in the execution of a parallel program. These events can be reordered without affecting the final result, and so it it not necessary to explore *every* interleaving. The reduction is often formulated in terms of an (in)dependence relation that determines which events may be reordered. Such a relation must make sure that independent steps leave the system in equivalent states, regardless of the order they are performed in.

An independence relation lifts to an equivalence relation on traces by permuting adjacent independent events. POR approaches then employ some algorithm to compute the equivalence classes of such a relation and avoid exploring traces in the same class. In practice it is difficult to compute the independence of events, so a sound over-approximation is used instead.

We instead take a more high-level approach. Considering trace equivalence to be a fundamental semantic building block, we develop our POR semantics parametric in this notion. This gives us an abstract notion, independent of the specific algorithm for POR.

To take advantage of partial order reduction, we define new transition systems.

344 ▶ **Definition 4.1** (POR Semantics). *The transition rules for symbolic POR are:*

$$\frac{\tau_0 \sim \tau_0' \qquad (s, \tau_0) \rightsquigarrow (s', \tau)}{(s, \tau_0') \rightsquigarrow_{POR} (s', \tau)} \qquad \frac{(s, \tau) \rightsquigarrow_{POR} (s', \tau')}{(C[s], \tau) \rightarrow_{POR} (C[s'], \tau')}$$

346 *And the transition rules for concrete POR are:*

$$\frac{\tau_0 \simeq \tau_0' \qquad (s, \tau_0) \rightsquigarrow_V (s', \tau)}{(s, \tau_0') \rightsquigarrow_{\boldsymbol{POR,V}} (s', \tau)} \qquad \frac{(s, \tau) \rightsquigarrow_{\boldsymbol{POR,V}} (s', \tau')}{(C[s], \tau) \Rightarrow_{\boldsymbol{POR,V}} (C[s'], \tau')}$$

348 This new reduction relation includes the steps of the symbolic case but requires only that the initial trace is *equivalent* in the sense defined in Section 3. Crucially, given a class of equivalent traces we may choose only one of them to continue execution. This is the source of *reduction*. Note that it is possible for (s, τ_0') to be unreachable in the original semantics, however the following completeness and correctness results ensure that this does not affect the final result. This approach most closely resembles *sleep sets* [15, 17] which keeps track of equivalent traces that do not need to be explored.

355 ▶ **Example 4.2.** Consider again the program from Example 2.9 and note that $(y := 1)$ and $(x := 3)$ are independent assignments. In the middle of some computation we are left with `skip` ∥ `skip` ∥ `if x` $\leq$ `1 {Y := 2}{Y := 3}` and the trace $[x := 3, y := 1]$. However, we have previously explored a computation from the state

$$(\texttt{skip} \parallel \texttt{skip} \parallel \texttt{if x} \leq \texttt{1 \{Y := 2\}\{Y := 3\}}, [y := 1, x := 3])$$

360 Now the POR semantics let us replace the equivalent traces and use this computation instead.

361 In order to utilize POR, we need to know that the reduced traces still model our programs' behavior. It should not throw away any important traces, nor should it invent new ones by taking unsound equivalence classes. Formally, we want the POR semantics to bisimulate their non-reduced counterpart up to trace equivalence.

365 ▶ **Theorem 4.3** (POR bisimulation). *For equivalent initial traces $\tau_0 \sim \tau_0'$:*

- *If $(s, \tau_0) \rightarrow_{POR} (s', \tau)$ then there exists $(s, \tau_0') \rightarrow (s', \tau')$ such that $\tau \sim \tau'$, and*
- *If $(s, \tau_0) \rightarrow (s', \tau)$ then there exists $(s, \tau_0') \rightarrow_{POR} (s', \tau')$ such that $\tau \sim \tau'$*

368 *For equivalent initial traces $\tau_0 \simeq \tau_0'$ and initial valuation V:*

- *If $(s, \tau_0) \Rightarrow_{\boldsymbol{POR,V}} (s', \tau)$ then there exists $(s, \tau_0') \Rightarrow_V (s', \tau')$ such that $\tau \simeq \tau'$, and*
- *If $(s, \tau_0) \Rightarrow_V (s', \tau)$ then there exists $(s, \tau_0') \Rightarrow_{\boldsymbol{POR,V}} (s', \tau')$ such that $\tau \simeq \tau'$*

371 From these bisimulation results, correctness and completeness follow by induction. Correctness captures the intuition that every PO-reduced execution corresponds to a non-reduced execution with equivalent final traces. This means that partial order reduction is precise in the sense that it does not introduce new traces with different final states.

375 Completeness is the opposite relationship: every direct execution has a corresponding reduced execution with equivalent traces. Since equivalent traces result in the same final state, completeness means that we do not lose any possible states when performing partial order reduction.

379 ▶ **Corollary 4.4** (Correctness and Completeness). *For two equivalent symbolic traces $\tau_0 \sim \tau_0'$:*
Completeness *If $(s, \tau_0) \rightarrow^*_{POR} (s', \tau)$ then there exists $(s, \tau_0') \rightarrow^* (s', \tau')$ with $\tau \sim \tau'$*
Correctness *If $(s, \tau_0) \rightarrow^* (s', \tau)$ then there exists $(s, \tau_0') \rightarrow^*_{POR} (s', \tau')$ with $\tau \sim \tau'$*

382 *For two equivalent concrete traces $\tau_0 \simeq \tau_0'$ and initial valuation V:*
Completeness *If $(s, \tau_0) \Rightarrow^*_{\boldsymbol{POR,V}} (s', \tau)$ then there exists $(s, \tau_0') \Rightarrow^*_V (s', \tau')$ with $\tau \simeq \tau'$*
Correctness *If $(s, \tau_0) \Rightarrow^*_V (s', \tau)$ then there exists $(s, \tau_0') \Rightarrow^*_{\boldsymbol{POR,V}} (s', \tau')$ with $\tau \simeq \tau'$*

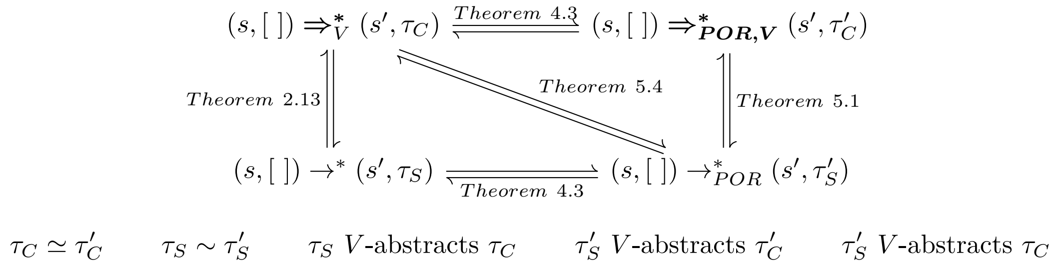


Figure 4 Overview of the correctness and completeness results

5 Composition of SE and POR

In this section we show that the bisimulation results of Section 2 and 4 compose naturally. We use this composition to fill in the remaining edges of Fig. 1, resulting in Fig. 4. This leads to the main result: a bisimulation relation between direct concrete semantics and symbolic POR semantics. Importantly, this allows reasoning about program analysis using *both* SE and POR with the symbolic trace abstracting the concrete trace.

The results are parametric in abstraction and trace equivalence in the following sense. Any equivalence relation on traces which is contained in ours — that is, whose equivalent traces have equivalent final states and path conditions — can be used to perform partial order reduction. Additionally, any symbolic abstraction satisfying Theorem 3.5 can be used for the symbolic execution. The result is a complete and correct *symbolic partial order reduction* where completeness and correctness follows from the respective completeness and correctness results of SE and POR semantics.

First we relate symbolic and concrete POR by combining Theorem 2.13 and Theorem 4.3.

▶ **Theorem 5.1** (POR-POR Bisimulation). *For initial traces τ_S, τ_C such that τ_S V-abstracts τ_C:*

- *If $(s,\tau_C) \Rightarrow_{\boldsymbol{POR,V}} (s',\tau'_C)$, then there exists $(s,\tau_S) \rightarrow_{POR} (s',\tau'_S)$ such that τ'_S V-abstracts τ'_C*
- *If $(s,\tau_S) \rightarrow_{POR} (s',\tau'_S)$ and $V \models pc(\tau'_S)$, then there exists $(s,\tau_C) \Rightarrow_{\boldsymbol{POR,V}} (s',\tau'_C)$ and $\tau'_C \Downarrow_V = V \circ (\tau'_S \Downarrow)$*

From this bisimulation, correctness and completeness relations are obtained by induction. These results are analogous to the direct relationships in Section 2, which shows that the correctness and completeness of symbolic execution is maintained through partial order reduction. In particular we may work with representatives of an *equivalence class* of traces rather than one single trace — which may greatly reduce the state space — and then perform symbolic execution in this new setting.

▶ **Corollary 5.2** (Trace POR Correctness). *If $(s,\tau_S) \rightarrow^*_{POR} (s',\tau'_S)$, τ_S V-abstracts τ_C, and $V \models pc(\tau'_S)$, then there exists a concrete trace τ'_C s.t $(s,\tau_C) \Rightarrow^*_{\boldsymbol{POR,V}} (s',\tau'_C)$ and $\tau'_C \Downarrow_V = \tau_C \Downarrow_V \circ(\tau'_S \Downarrow)$*

▶ **Corollary 5.3** (Trace POR Completeness). *If $(s,\tau_C) \Rightarrow^*_{\boldsymbol{POR,V}} (s',\tau'_C)$ and τ_S V-abstracts τ_C, there exist τ'_S s.t $(s,\tau_S) \rightarrow^*_{POR} (s',\tau'_S)$ and τ'_S V-abstracts τ'_C.*

We are now ready to state our main result, filling in the diagonal and connecting concrete semantics directly to PO-reduced symbolic semantics. Formally, Theorem 2.13 and

418 Theorem 4.3 can be combined to obtain bisimulation of the basic concrete semantics and
419 PO-reduced symbolic semantics.

420 ▶ **Theorem 5.4** (Total Bisimulation). *For initial traces τ_S, τ_C such that τ_S V-abstracts τ_C:*
421 - *If $(s, \tau_C) \Rightarrow_V (s', \tau'_C)$, then there exists $(s, \tau_S) \rightarrow_{POR} (s', \tau'_S)$ such that τ'_S V-abstracts*
422 *τ'_C*
423 - *If $(s, \tau_S) \rightarrow_{POR} (s', \tau'_S)$ and $V \models pc(\tau'_S)$, then there exists $(s, \tau_C) \Rightarrow_V (s', \tau'_C)$ and*
424 *$\tau'_C \Downarrow_V = V \circ (\tau'_S \Downarrow)$*

425 ▶ **Corollary 5.5** (Total Correctness). *If $(s, \tau_0) \rightarrow^*_{POR} (s', \tau)$, τ_0 V-abstracts τ'_0 and $V \models pc(\tau)$,*
426 *then there exists τ' such that $(s, \tau'_0) \Rightarrow^*_V (s', \tau')$ and τ V-abstracts τ'.*

427 ▶ **Corollary 5.6** (Total Completeness). *If $(s, \tau_0) \Rightarrow^*_V (s', \tau)$ and τ'_0 V-abstracts τ_0, there*
428 *exist τ' s.t $(s, \tau'_0) \rightarrow^*_{POR} (s', \tau')$ and τ' V-abstracts τ.*

429 Figure 4 shows all four reduction systems — symbolic and concrete, with and without
430 POR. Each double arrow denotes a notion of bisimulation, and we obtain the properties
431 shown: both symbolic and concrete traces are equivalent across POR, and V-abstraction is
432 maintained across the symbolic/concrete divide as well as their composition. Additionally we
433 show the relationships between the four traces — the symbolic traces abstract their concrete
434 counterparts, and the POR traces are equivalent — although by Theorem 3.5 it suffices to
435 know the equivalences and one of the abstractions.

436 5.1 Discussion

437 The bisimulations compose naturally. As an example, consider Theorem 5.4 which is obtained
438 by composing the symbolic/concrete bisimulation of Theorem 2.13 and the direct/reduced
439 bisimulation of Theorem 4.3. Starting with a concrete execution with trace τ_C we first obtain
440 a symbolic execution with trace τ_S such that τ_S V-abstracts τ_C. Then the POR-bisimulation
441 of Theorem 4.3 gives a symbolic POR-computation with an equivalent trace τ_S. Since trace
442 equivalence is a congruence for abstraction (Theorem 3.5) and τ_C is equivalent to itself, this
443 final trace also abstracts τ_C.
444 The ease of this composition is not unexpected, since both abstraction and trace equiva-
445 lence were explicitly formulated to preserve the relevant parts of the program state. The result
446 is that any partial order reduction which picks equivalent traces in this sense preserves the
447 correctness and completeness properties of the symbolic execution. Explicitly, if the notion
448 of trace equivalence is contained in ours and the symbolic abstraction can be transported
449 along this equivalence in the sense of Theorem 3.5 then the techniques can be composed.

450 5.2 Mechanization

451 In this section we cover some of the details of the mechanization in Coq.
452 The basic building blocks of program state are simple. Both substitutions and valuations
453 are implemented as total maps from strings, parameterized by a result type. Updates,
454 notation and several useful lemmas about maps can be proven generically and the notation
455 mirrors that of Pierce et al. [25]. Similarly traces are an inductive type, parametric in the
456 type of events. In essence they are lists, but extended to the right for convenience, with the
457 expected operations and properties.
458 Trace *equivalence* is defined as a relation. Then we show necessary properties of this
459 relation, in particular Lemma 3.4 and Theorem 3.5 which are used in proofs. Additionally, we
460 implement an equivalence by permuting independent events and show that it satisfies the same

properties if the independence relation does. This part is parametric in the independence relation and serves as an example of a POR relation. The example at the end of Section 3 is an instance with interference freedom as the independence relation.

Expressions (both arithmetic and Boolean) and statements are inductive types. As an example, the type of statements is given by:

```
Inductive Stmt : Type :=
| SAsgn (x:Var) (e:Aexpr)
| SPar (s1 s2:Stmt)
| SIf (b:Bexpr) (s1 s2:Stmt)
...
```

To give semantics to this language, we define a *head reduction* relation and a type of contexts. The head reduction describes the single step reductions for each atomic and how it transforms the current trace. For example an assignment reduces to `skip` and appends the assignment to the current trace. Here `<{_}>` encloses language statements and `Asgn__S x e` represents the symbolic event $(x := e)$.

```
Variant head_red__S: (trace__S * Stmt) → (trace__S * Stmt) → Prop :=
| head_red_asgn__S: ∀t x e,
   head_red__S (t, <{ x := e }>) (t :: Asgn__S x e, SSkip)
   ...
```

Note that `Variant` is a version of `Inductive` that does not include recursive constructors.

Contexts are implemented as functions `Stmt → Stmt` along with an inductive relation `is_context: (Stmt → Stmt) → Prop` — an approach inspired by Xavier Leroy [21]. This approach allows us to define transition relation semantics parametric in both the type of contexts and the head reduction relation. The following generalizes the ∗-IN-CONTEXT rules for any type of state `X`. In our case, `X` will be a type of traces, but note that `X` appears on the left — this makes the rule amenable to states represented by product types due to the way parentheses associate.

```
Variant context_red
  (is_cont: (Stmt → Stmt) → Prop) (head_red: relation (X * Stmt))
  : relation (X * Stmt) :=
| ctx_red_intro: ∀C x x' s s',
  head_red (x, s) (x', s') → is_context C →
  context_red is_cont head_red (x, C s) (x', C s').
```

Having used `context_red` with the appropriate `is_context` and `head_red` we obtain the full transition relation by stepwise reflexive-transitive closure to the right (`clos_refl_trans_n1`) from the `Relations` library.

The proofs are performed in two steps. Induction on the transition relation leaves us with either a reflexive step or an induction hypothesis and some sequence followed by a step. Then unfolding and dependent destruction (from `Program.Equality`) can be used on the step to unpack `ctx_red_intro` and split on the head reduction rule while remembering the ultimate and penultimate traces.

6 Related Work

We focus on a simple formal model that permits reasoning about symbolic execution and partial order reduction. De Boer and Bonsangue [5] lay the foundations of our work — a symbolic execution model based on transition systems and symbolic substitutions which may be composed with concrete valuations. They do not consider parallelism, but do apply

513 their model to languages with other features including recursive function calls and dynamic
514 object creation. They also explore a kind of trace semantics for the latter extension, but it
515 differs from the semantics considered herein. Extending the current work with more language
516 features, including procedure calls and synchronization tools would be interesting.

517 SymPaths [6] explores the use of POR for SE in a manner very similar to ours, but
518 does not explicitly compose the correctness and completeness of SE and POR, nor treat
519 the relationship to partial order reduction in the non-symbolic case. Additionally, their
520 treatment of trace equivalence focuses on one specific independence relation while we take a
521 more abstract view.

522 Other formal approaches to symbolic execution have also been considered in the literature.
523 Steinhöfel [31] focuses on the semantics of the SE system and uses a concretization function to
524 relate sets of symbolic and concrete states. The Gillian platform [14,23] and related work [28]
525 uses separation logic to construct a SE system that is parametric in the target memory model.
526 Rosu et al. [22,27,30] develop reachability logic to present symbolic execution parameterized
527 by the semantics of the target language. These all present alternative approaches to the left
528 edge of Figure 4.

529 There are also other approaches to partial order reduction. In particular, dynamic or
530 stateless POR (DPOR) [1,13,16,26] avoids exploring equivalent future traces by identifying
531 backtracking points. Additionally the *unfolding* approach explores partial orders more directly
532 as a tree-like event structure [26]. Unfolding has been fruitfully combined with symbolic
533 execution in practice [29].

534 ## 7 Conclusion

535 POR and SE are fundamental abstraction techniques in program analysis. SE is particu-
536 larly useful as a state abstraction technique for sequential programs, while POR addresses
537 equivalent interleavings in the execution of concurrent programs. In this paper, we study
538 the foundations of both techniques based on transition systems and trace semantics, in the
539 context of a core imperative language with parallelism. The formalization provides a unified
540 view of concrete and symbolic semantics with and without partial order reduction. We
541 further formalize correctness and completeness relations for both POR and SE, and compose
542 these relations to study how SE and POR can be combined while preserving correctness
543 and completeness. Our work shows that the framework of correctness and completeness
544 relations between symbolic and concrete transition systems, introduced by de Boer and
545 Bonsangue, extends to parallelism and trace semantics, and provides a natural setting to
546 study formalizations of abstraction techniques for SE, such as POR.

547 In addition, our formal development of correctness and completeness relations of SE and
548 POR has been fully mechanized using Coq[2]. We believe the mechanization of this framework
549 in Coq can be useful to the community to study further formalizations of abstraction
550 techniques for symbolic execution and their correctness. In particular, in future work, we
551 plan to extend the framework developed in this paper to understand relations between
552 concrete SE frameworks typically used for software testing [9], such as Klee [8], in which
553 states are described using symbolic stores as in this paper, and abstract SE frameworks
554 typically used for deductive verification, such as KeY [2], in which states are described using
555 predicates.

[2] Provided as supplementary material at `https://github.com/Aqissiaq/symex-formally-formalized` and `https://zenodo.org/record/8070170`

References

1. Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In Suresh Jagannathan and Peter Sewell, editors, *Proc. 41st Annual Symposium on Principles of Programming Languages (POPL'14)*, pages 373–384. ACM, 2014. doi:10.1145/2535838.2535845.
2. Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*, volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016. doi:10.1007/978-3-319-49812-6.
3. Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.
4. Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*. Springer, 2013.
5. Frank S de Boer and Marcello Bonsangue. Symbolic execution formally explained. *Formal Aspects of Computing*, 33(4):617–636, 2021.
6. Frank S de Boer, Marcello Bonsangue, Einar Broch Johnsen, Violet Ka I Pun, S Lizeth Tapia Tarifa, and Lars Tveito. SymPaths: Symbolic execution meets partial order reduction. In *Deductive Software Verification: Future Perspectives*, pages 313–338. Springer, 2020.
7. Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT - a formal system for testing and debugging programs by symbolic execution. In Martin L. Shooman and Raymond T. Yeh, editors, *Proc. International Conference on Reliable Software 1975*, pages 234–245. ACM, 1975. doi:10.1145/800027.808445.
8. Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *Proc. 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008)*, pages 209–224. USENIX Association, 2008. URL: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.
9. Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013. doi:10.1145/2408776.2408795.
10. Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007. doi:10.1007/978-3-540-71999-1.
11. Crystal Chang Din, Reiner Hähnle, Ludovic Henrio, Einar Broch Johnsen, Violet Ka I Pun, and Silvia Lizeth Tapia Tarifa. LAGC semantics of concurrent programming languages. *arXiv preprint arXiv:2202.12195*, 2022.
12. Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992. doi:10.1016/0304-3975(92)90014-7.
13. Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proc. 32nd Symposium on Principles of Programming Languages (POPL'05)*, pages 110—-121. ACM, 2005. doi:10.1145/1040305.1040315.
14. José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. Gillian, part i: a multi-language platform for symbolic execution. In *Proc. 41st ACM Conference on Programming Language Design and Implementation (PLDI'20)*, pages 927–942, 2020.
15. Patrice Godefroid. Using partial orders to improve automatic verification methods. In Edmund M. Clarke and Robert P. Kurshan, editors, *Computer-Aided Verification*, pages 176–185. Springer, 1991.
16. Patrice Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*. Springer, 1996.

607 **17** Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of
608 deadlock freedom and safety properties. *Formal Methods in System Design*, 2:149–164, 1993.
609 **18** Shmuel Katz and Zohar Manna. Towards automatic debugging of programs. *ACM SIGPLAN*
610 *Notices*, 10(6):143–155, 1975.
611 **19** James C King. Symbolic execution and program testing. *Communications of the ACM*,
612 19(7):385–394, 1976.
613 **20** Åsmund Aqissiaq Arild Kløvstad. Compositional correctness and completeness for symbolic
614 partial order reduction: Concur23, June 2023. doi:10.5281/zenodo.8070170.
615 **21** Xavier Leroy. Mechanized semantics. Course materials, 2020. URL: https://github.com/
616 xavierleroy/cdf-mech-sem.
617 **22** Dorel Lucanu, Vlad Rusu, and Andrei Arusoaie. A generic framework for symbolic execution:
618 A coinductive approach. *Journal of Symbolic Computation*, 80:125–163, 2017. SI: Program
619 Verification. doi:10.1016/j.jsc.2016.07.012.
620 **23** Petar Maksimović, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. Gillian,
621 part ii: Real-world verification for JavaScript and C. In Alexandra Silva and K. Rustan M.
622 Leino, editors, *Computer Aided Verification*, pages 827–850. Springer, 2021.
623 **24** Antoni Mazurkiewicz. Concurrent program schemes and their interpretations. *DAIMI Report*
624 *Series*, 6(78), Jul. 1977. URL: https://tidsskrift.dk/daimipb/article/view/7691, doi:
625 10.7146/dpb.v6i78.7691.
626 **25** Benjamin C Pierce, A Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael
627 Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, A Tolmach, and B Yorgey. Software foundations,
628 volume 2: Programming language foundations. 2017.
629 **26** César Rodríguez, Marcelo Sousa, Subodh Sharma, and Daniel Kroening. Unfolding-based
630 partial order reduction. In Luca Aceto and David de Frutos-Escrig, editors, *26th International*
631 *Conference on Concurrency Theory, CONCUR 2015*, volume 42 of *LIPIcs*, pages 456–469.
632 Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.CONCUR.2015.
633 456.
634 **27** Grigore Rosu, Andrei Stefanescu, Stefan Ciobâcá, and Brandon M. Moore. One-path reach-
635 ability logic. In *Proc. 28th Annual ACM/IEEE Symposium on Logic in Computer Science*
636 *(LICS'13)*, pages 358–367, 2013. doi:10.1109/LICS.2013.42.
637 **28** José Fragoso Santos, Petar Maksimović, Théotime Grohens, Julian Dolby, and Philippa
638 Gardner. Symbolic execution for JavaScript. In *Proc. 20th International Symposium on*
639 *Principles and Practice of Declarative Programming*, PPDP '18. ACM, 2018. doi:10.1145/
640 3236950.3236956.
641 **29** Daniel Schemmel, Julian Büning, César Rodríguez, David Laprell, and Klaus Wehrle. Symbolic
642 partial-order execution for testing multi-threaded programs. In *International Conference on*
643 *Computer Aided Verification*, pages 376–400. Springer, 2020.
644 **30** Andrei Ştefănescu, Ştefan Ciobâcă, Radu Mereuta, Brandon M. Moore, Traian Florin Şer-
645 bănută, and Grigore Roşu. All-path reachability logic. In Gilles Dowek, editor, *Rewriting and*
646 *Typed Lambda Calculi*, pages 425–440. Springer, 2014.
647 **31** Dominic Steinhöfel. *Abstract execution: automatically proving infinitely many programs.* PhD
648 thesis, Technische Universität Darmstadt, 2020.
649 **32** Dominic Steinhöfel and Reiner Hähnle. The trace modality. In *Dynamic Logic. New Trends*
650 *and Applications*, pages 124–140. Springer, 2020. doi:10.1007/978-3-030-38808-9_8.
651 **33** The Coq Development Team. The Coq proof assistant, September 2022. doi:10.5281/zenodo.
652 7313584.

653 # A Auxiliary Lemmas

654 ▶ **Lemma A.1** (Substitution [5]). $V \circ \sigma(e) = V(e\sigma)$

655 ▶ **Corollary A.2** (Soundness of Assignment [5]). $V \circ (\sigma[x := a\sigma]) = V'[x := V'(a)]$*, where*
656 $V' = V \circ \sigma$

B Selected Proofs

B.1 Theorem 2.13: Bisimulation of concrete and symbolic semantics

For any initial valuation V and initial traces τ_0, τ_0' such that τ_0 V-abstracts τ_0':

- if there is a concrete step $(s, \tau_0) \Rightarrow_V (s', \tau)$, then there exists a symbolic step $(s, \tau_0') \rightarrow (s', \tau')$ such that τ' V-abstracts τ, and
- if there is a symbolic step $(s, \tau_0') \rightarrow (s', \tau')$ and $V \models pc(\tau')$, then there exists a concrete step $(s, \tau_0) \Rightarrow_V (s', \tau)$ such that $\tau \Downarrow_V = V \circ \tau' \Downarrow$

Proof. Both directions are proven by case analysis and applying the corresponding rule (ie. the matching one from Figure 3). The `skip`-cases follow immediately, but the others take a bit more work.

For the conditional steps (choice and iteration), the concrete-to-symbolic direction requires splitting on the evaluation of the guard and picking the right symbolic branch. Then Lemma A.1 ensures that the resulting trace is the correct one. The symbolic-to-concrete direction requires showing that if the true (resp. false) branch is chosen, then the guard evaluates to true (resp. false) — which, again, holds by Lemma A.1.

Finally, both directions of the assignment step follow directly from Corollary A.2. ◀

B.2 Lemma 3.4: Equivalent steps

For equivalent traces $\tau \sim \tau'$, if $(s, \tau) \rightarrow (s', \tau_1)$ then there exists τ_2 so that $(s, \tau') \rightarrow (s', \tau_2)$ and $\tau_1 \sim \tau_2$

Proof. First note that

$$\text{If } \tau \sim \tau', \text{ then } \tau :: e \sim \tau' :: e \text{ for any event } e \tag{1}$$

Then we proceed by case analysis of the reduction step. In the `skip` cases, we pick $\tau_2 = \tau'$ and the equivalence follows by assumption. In the other cases, τ' is extended by some event, but then the result is equivalent by 1. ◀

B.3 Theorem 3.5: Abstraction Congruence

For equivalent symbolic traces $\tau_S \sim \tau_S'$ and concrete traces $\tau_C \simeq \tau_C'$, if τ_S V-abstracts τ_C then τ_S' V-abstracts τ_C'.

Proof. Since τ_S V-abstracts τ_C, V satisfies the path condition of τ_S and $\tau_C \Downarrow_V$ is the composition of V and $\tau_S \Downarrow$. But since it is trace equivalent, τ_S' has equivalent path condition, so it must also be satisfied. Furthermore its final substitution is the same as τ_S and likewise the final valuation of τ_C' is equal to that of τ_C, so $\tau_C' \Downarrow_V = V \circ \tau_S' \Downarrow$ as required. ◀

B.4 Theorem 4.3: POR Bisimulation

For equivalent initial traces $\tau_0 \sim \tau_0'$:

- If $(s, \tau_0) \rightarrow_{POR} (s', \tau)$ then there exists $(s, \tau_0') \rightarrow (s', \tau')$ such that $\tau \sim \tau'$, and
- If $(s, \tau_0) \rightarrow (s', \tau)$ then there exists $(s, \tau_0') \rightarrow_{POR} (s', \tau')$ such that $\tau \sim \tau'$

For equivalent initial traces $\tau_0 \simeq \tau_0'$ and initial valuation V:

- If $(s, \tau_0) \Rightarrow_{\boldsymbol{POR,V}} (s', \tau)$ then there exists $(s, \tau_0') \Rightarrow_V (s', \tau')$ such that $\tau \simeq \tau'$, and
- If $(s, \tau_0) \Rightarrow_V (s', \tau)$ then there exists $(s, \tau_0') \Rightarrow_{\boldsymbol{POR,V}} (s', \tau')$ such that $\tau \simeq \tau'$

Proof. Both results follow directly from unpacking contexts and applying Lemma 3.4. ◀

B.5 Theorem 5.1: POR-POR Bisimulation

For initial traces τ_S, τ_C such that τ_S V-abstracts τ_C:

- If $(s, \tau_C) \Rightarrow_{\boldsymbol{POR,V}} (s', \tau'_C)$, then there exists $(s, \tau_S) \rightarrow_{POR} (s', \tau'_S)$ such that τ'_S V-abstracts τ'_C
- If $(s, \tau_S) \rightarrow_{POR} (s', \tau'_S)$ and $V \models pc(\tau'_S)$, then there exists $(s, \tau_C) \Rightarrow_{\boldsymbol{POR,V}} (s', \tau'_C)$ and $\tau'_C \Downarrow_V = V \circ (\tau'_S \Downarrow)$

Proof. Let us consider the first (completeness) direction. By the concrete version of Theorem 4.3 we obtain the concrete step $(s, \tau) \Rightarrow_V (s', \tau''_C)$ with $\tau''_C \simeq \tau'_C$. Then, by Theorem 2.13 we have a symbolic step $(s, \tau) \rightarrow (s', \tau''_S)$ such that τ''_S V-abstracts τ''_C. Finally, the symbolic half of Theorem 4.3 provides the desired $(s, \tau_S) \rightarrow_{POR} (s', \tau'_S)$ with $\tau'_S \sim \tau''_S$ and abstraction congruence (Theorem 3.5) shows τ'_S V-abstracts τ'_C. The other direction is analoguous, using the other half of each bisimulation. ◀

B.6 Theorem 5.4: Total Bisimulation

For initial traces τ_S, τ_C such that τ_S V-abstracts τ_C:

- If $(s, \tau_C) \Rightarrow_V (s', \tau'_C)$, then there exists $(s, \tau_S) \rightarrow_{POR} (s', \tau'_S)$ such that τ'_S V-abstracts τ'_C
- If $(s, \tau_S) \rightarrow_{POR} (s', \tau'_S)$ and $V \models pc(\tau'_S)$, then there exists $(s, \tau_C) \Rightarrow_V (s', \tau'_C)$ and $\tau'_C \Downarrow_V = V \circ (\tau'_S \Downarrow)$

Proof. The proof follows a similar structure to the above, composing the concrete/symbolic (Theorem 2.13) and symbolic POR (Theorem 4.3) bisimulations. ◀