SymPaths: Symbolic Execution Meets Partial Order Reduction

Frank de Boer[1], Marcello Bonsangue[2], Einar Broch Johnsen[3], Violet Ka I Pun[3,4], S. Lizeth Tapia Tarifa[3], and Lars Tveito[3]

[1] CWI, Amsterdam, the Netherlands
f.s.de.boer@cwi.nl
[2] Leiden University, Leiden, the Netherlands
m.m.bonsangue@liacs.leidenuniv.nl
[3] Department of Informatics, University of Oslo, Oslo, Norway
{einarj,violet,sltarifa,larstvei}@ifi.uio.no
[4] Western Norway University of Applied Sciences, Bergen, Norway
Violet.Ka.I.Pun@hvl.no

Abstract. Symbolic execution is an important technique for software analysis, which enables systematic model exploration by following all possible execution paths for a given program. For multithreaded shared variable programs, this technique leads to a state space explosion. Partial order reduction is a technique which allows equivalent execution paths to be recognized, reducing the state space explosion problem. This paper provides formal justifications for these techniques in a multithreaded setting by proving the correctness and completeness of symbolic execution for multithreaded shared variable programs, with and without the use of partial order reduction. We then show how these formal justifications carry over to prove the soundness and relative completeness of a proof system for such multithreaded shared variable programs in dynamic logic, such that partial order reduction can be used to simplify the proof construction by mitigating the state space explosion.

1 Introduction

Symbolic execution [1] is an important technique for software analysis. It is especially used for testing, but also for debugging and deductive verification. In fact, the KeY verification system [2] is based on symbolic execution. Symbolic execution is intuitively very appealing, because one symbolic execution may correspond to a large, possibly infinite, class of normal (concrete) executions by representing the input values to a program by logical variables. Symbolic execution has made tremendous progress in recent years; recent surveys [3, 4] cover improvements in the effectiveness as well as the achieved code coverage of symbolic execution tools such as, e.g., DART [5], EXE [6], KLEE [7]. As already observed in [8], the formal justification for the technique in terms of correctness has received much less attention. In contrast, [8] formally defines the correctness of symbolic execution by relating a symbolic transition system to a concrete structural operational semantics.

Symbolic execution has mainly been applied to sequential languages, where different executions can be captured by different path conditions. By considering executions with different path conditions, a tool can systematically explore the entire symbolic execution graph of a sequential program. In contrast, concurrent languages give rise to non-determinism during execution, which is another source of state space explosion when using symbolic execution for such model exploration. In this paper, we study symbolic execution for multithreaded shared variable programs in terms of a symbolic transition system. In order to capture the non-determinism, we combine path conditions with scheduling traces, which reflect the order in which different threads have been selected for execution, into so-called symbolic paths (or *sympaths*). We prove the correctness and completeness of the resulting system by extending the work of [8] to a multithreaded setting. To address the state space explosion resulting from the different possible scheduling decisions, we show how *partial order reduction* (POR) [9] can be introduced into the symbolic execution framework. POR allows symbolic states to be pruned from the overall search space during model exploration, by merging symbolic states with equivalent symbolic paths. We extend the symbolic transition system with a pruning operation for search states by means of POR, and prove the correctness and completeness of the resulting system.

Finally, we consider a proof system for multithreaded shared variable programs using dynamic logic [10], in the spirit of symbolic execution in the KeY system. We show how soundness and relative completeness of this proof system can be obtained by extending the soundness and correctness proofs for the symbolic transition system. We further show how POR can be applied in this setting to reduce the state space explosion during the verification in the proof system, and again prove soundness and relative completeness by extending the corresponding soundness and correctness proofs for the symbolic transition system with POR.

The main contributions of this paper can be summarized as follows:

- A formal model of symbolic execution for multithreaded shared variable programs, with a formal justification in terms of correctness and completeness;
- Pruning techniques for the symbolic execution model using POR, with a formal justification in terms of correctness and completeness;
- A Maude [11] implementation of the two symbolic execution models, allowing their comparison in terms of the number of reduction steps to explore the full state space and the number of final symbolic states; and
- A dynamic logic proof system for multithreaded shared variable programs with pruning techniques using POR, with a formal justification in terms of soundness and relative completeness proofs.

2 Background

2.1 Symbolic Execution

In the development of a formal model of the symbolic execution of multithreaded shared variable programs, we follow the general approach of [8]. We define the

symbolic execution of such programs by means of a transition system for generating the atomic computation steps between symbolic configurations. A symbolic configuration consists of the program to be executed, a substitution which symbolically represents the state, i.e., the assignment of symbolic values to the shared program variables, and a path condition. In general, a substitution assigns expressions of the programming language to the program variables.

We then provide a formal justification of this symbolic transition system by relating it to a transition system for generating the atomic computation steps between concrete configurations. A concrete configuration consists of the multithreaded program that remains to be executed and the assignment of values to the shared program variables. The formal justification involves a proof of correctness and completeness. Correctness establishes that executing the program in any initial state that satisfies the path condition leads to the corresponding execution path. On the other hand, completeness amounts to showing that all concrete executions are covered by the symbolic transition system.

In this paper, we extend the approach of [8] to multithreaded shared variable programs. At the core of this extension lies the symbolic representation of scheduling information in the path condition.

2.2 Partial Order Reduction

Partial order reduction (POR) is a technique to reduce the size of the state space when exploring the different executions of a parallel program by exploiting the commutativity of concurrently executed *independent* transitions [9, 12]. This commutativity relation between transitions is lifted to an equivalence relation $\sim$ on traces over these transitions. Given a trace tr reflecting an interleaved execution of a number of parallel threads, the set $[tr]$ of traces equivalent to tr according to the equivalence relation $\sim$, is supposed to preserve the sequential order of transitions for the individual threads. Thus, all equivalent traces have the same length and contain the same events. Let $s \xrightarrow{tr} s'$ denote that the state s' can be reached from a state s by applying the transition steps according to the order given by a trace tr. The pruning of states based on traces is justified during model exploration when the traces are sufficiently expressive to make sure that equivalent traces lead to equal states; i.e., the following must be a theorem [9]:

Theorem 1. *If* $s_0 \xrightarrow{tr_1} s_1$, $s_0 \xrightarrow{tr_2} s_2$ *and* $tr_2 \in [tr_1]$, *then* $s_1 = s_2$.

Observe that, given a trace tr, the elements of $[tr]$ can be enumerated by successively permuting adjacent commuting events. An additional problem is to identify syntactic criteria to approximate this semantic notion of equivalence. This can be done by identifying transitions that correspond to *interference-free* statements [13]; e.g., two transitions are independent if their corresponding statements do not affect each others' program variables.

In program analysis, POR can be used to explore the different equivalence classes of executions, rather than exploring every execution path. Assuming that we can decide whether two traces are in the same equivalence class, we can stop

$$
\begin{aligned}
Pr \in Prog &::= P \ldots P \\
P \in Proc &::= \texttt{proc}\ \{s\} \\
s \in Stm &::= s; s \mid x := e \mid \texttt{if}\ e\ \{s\}\ \texttt{else}\ \{s\} \mid \texttt{while}\ e\ \{s\} \\
e \in Exp &::= x \mid v \mid op(e, \ldots, e) \\
op \in Ops &::= ==\mid \wedge \mid \vee \mid + \mid - \mid < \mid \leq \mid ... \\
v \in Val &::= \texttt{True} \mid \texttt{False} \mid 0 \mid 1 \mid 2 \mid \ldots
\end{aligned}
$$

Fig. 1: The syntax of the basic programming language *PL*.

the analysis of the current execution path if we know that its trace is equivalent to the trace of an execution that we have already explored.

3 Combining Symbolic Execution and POR

We formalize how symbolic execution and partial order reduction can be combined in the context of a basic calculus of parallel execution.

3.1 A Basic Calculus of Parallel Processes

To formalize the main concepts of symbolic execution with partial order reduction, we consider a basic programming language *PL* with a given set of types, including `Bool` and `Int`, and a set *Var* of program variables, with the typical element x. The syntax of *PL* is defined in Fig. 1, where s denotes a sequence of statements. A program *Pr* consists of a list of processes P of the form $\texttt{proc}\ \{s\}$, where the statement s of the process will be executed in parallel with the statements of other processes. Statements s contain sequential compositions $s_1; s_2$, assignments $x := e$, conditionals and `while`-statements. Expressions e consist of program variables x, values v, and operators *op* applied to expressions. The operators include standard operators on the types `Bool` and `Int`, and values v the usual values of these types. A *predicate* is an expression of type `Bool`; the set of predicates is denoted by *BExp*, with typical element b. We assume that programs are well-typed, so operators are recursively applied to the correct number of correctly typed subexpressions.

3.2 Concrete Semantics

A valuation ϵ is a (mathematical) function $Var \to Val$ which assigns to each program variable $x \in Var$ a value $v \in Val$. For any expression e, let $\epsilon(e)$ denote its value with respect to the valuation ϵ (defined by induction on the structure of e). We now describe a transition system for the concrete execution of *PL* programs, based on a transition relation between concrete states. These states are given by the grammar in Fig. 2.

Definition 1 (Concrete States). *A* concrete state *cs is a term* $(\!|\epsilon, \Sigma|\!)$*, where* ϵ *denotes a valuation and* Σ *denotes a thread pool.*

$$
\begin{aligned}
cs \in \mathit{ConState} &::= (\!|\epsilon, \Sigma|\!) \\
\tau \in \mathit{Thread} &::= \iota(s) \\
\epsilon \in \mathit{Valuation} &::= \varepsilon \mid \epsilon[x \mapsto v] \\
\Sigma \in \mathit{ThreadPool} &::= \{\tau\} \mid \Sigma \cup \Sigma \\
V \in \mathit{VarSet} &::= \{x\} \mid V \cup V
\end{aligned}
$$

Fig. 2: Runtime syntax for the concrete semantics, where ι is a thread identifier, v a value and s a statement.

Fig. 3 defines the concrete semantics for PL by means of a transition relation $\rightarrow_c$ between concrete states. The rules Conc-Assign, Conc-Cond1, Conc-Cond2, Conc-While1 and Conc-While2 respectively describe the concrete executions of assignments, conditional and `while`-statements by a single thread ι. The concrete execution of the assignment in a thread ι, as in rule Conc-Assign, results on an update in ϵ, where the new value for variable x is the value obtained by the valuation $\epsilon(e)$ of the expression e. For a conditional statement in a thread ι, we apply one of the rules Conc-Cond1 or Conc-Cond2, depending on the valuation $\epsilon(b)$ of Boolean expression b. If the valuation is `True`, then we reduce the conditional statement to $s1$ by applying rule Conc-Cond1, otherwise we reduce it to $s2$ by applying rule Conc-Cond2. The `while`-statement is similar to the conditional statement, applying rules Conc-While1 or Conc-While2, respectively. We let $T(\Sigma)$ denote the set of identifiers for the active threads in thread pool Σ (i.e, the threads in the thread pool which have not yet terminated).

$$
\frac{v = \epsilon(e)}{\begin{array}{c}(\!|\epsilon, \{\iota(x := e; s)\} \cup \Sigma|\!) \\ \rightarrow_c (\!|\epsilon[x \mapsto v], \iota(s)\} \cup \Sigma|\!)\end{array}}\ (\textsc{Conc-Assign})
$$

$$
\frac{\epsilon(b) = \texttt{True}}{\begin{array}{c}(\!|\epsilon, \{\iota(\texttt{if } b \ \{s_1\} \ \texttt{else} \ \{s_2\}; s)\} \cup \Sigma|\!) \\ \rightarrow_c (\!|\epsilon, \{\iota(s_1; s)\} \cup \Sigma|\!)\end{array}}\ (\textsc{Conc-Cond1})
\qquad
\frac{\epsilon(b) = \texttt{False}}{\begin{array}{c}(\!|\epsilon, \{\iota(\texttt{if } b \ \{s_2\} \ \texttt{else} \ \{s_2\}; s)\} \cup \Sigma|\!) \\ \rightarrow_c (\!|\epsilon, \{\iota(s_2; s)\} \cup \Sigma|\!)\end{array}}\ (\textsc{Conc-Cond2})
$$

$$
\frac{\epsilon(b) = \texttt{True}}{\begin{array}{c}(\!|\epsilon, \{\iota(\texttt{while } b \ \{s_1\}; s_2)\} \cup \Sigma|\!) \\ \rightarrow_c (\!|\epsilon, \{\iota(s_1; \texttt{while } b \ \{s_1\}; s_2)\} \cup \Sigma|\!)\end{array}}\ (\textsc{Conc-While1})
\qquad
\frac{\epsilon(b) = \texttt{False}}{\begin{array}{c}(\!|\epsilon, \{\iota(\texttt{while } b \ \{s_1\}; s_2)\}\Sigma|\!) \\ \rightarrow_c (\!|\epsilon, \{\iota(s_2)\} \cup \Sigma|\!)\end{array}}\ (\textsc{Conc-While2})
$$

Fig. 3: A concrete semantics for PL.

Initial and final concrete states. Given a program $Pr = \texttt{proc}\ \{s_1\} \ldots \texttt{proc}\ \{s_n\}$, let $\mathit{init}(Pr)$ denote the set of threads $\{\iota_1(s_1), \ldots, \iota_n(s_n)\}$ such that $\iota_1, \ldots, \iota_n$ are distinct thread identifiers. The *initial state* cs_0 for the concrete execution of Pr is given by the concrete state $(\!|\epsilon, \mathit{init}(Pr)|\!)$, where ϵ assigns some initial values to the program variables in Pr (i.e., $\epsilon(x) = v_0$, for all $x \in \mathit{VarSet}$). The execution of a program terminates when the concrete state $(\!|\epsilon, \Sigma|\!)$ is such that $T(\Sigma) = \emptyset$ (technically, we may add a termination marker as a statement in the runtime syntax). For notational convenience, we denote such final concrete states by $(\!|\epsilon, \emptyset|\!)$.

A concrete execution consists of a sequence of concrete states $cs_0, \ldots, cs_n$ such that cs_0 is some initial concrete state of the form $(\!|\epsilon, \mathit{init}(Pr)|\!)$, and for $0 \leq i < n$ there exist concrete transitions $cs_i \rightarrow_c cs_{i+1}$.

Proposition 1 (Reachability for concrete executions). *A concrete state cs is reachable if and only if there exists a concrete execution $cs_0, \ldots, cs_n$ such that $cs_n = cs$.*

This proposition follows by a straightforward induction on the length of the concrete execution. □

3.3 Symbolic Semantics

We now introduce the machinery for the symbolic execution of *PL*. Let $\mathit{vars}(e)$ denote an inductively defined function which returns the set of program variables in an expression e. Abstracting from its possible KeY implementation as a sequence of symbolic updates, we define a symbolic substitution σ as a (mathematical) function $\mathit{Var} \rightarrow \mathit{Exp}$ which assigns to each program variable $x \in \mathit{Var}$ an expression $e \in \mathit{Exp}$ with the following constructors: ε denotes the empty substitution, and $\sigma[x \mapsto e]$ the substitution mapping x to e and y to $\sigma(y)$ for every other variable y (i.e., $y \neq x$). As usual, composition $\sigma_1 \circ \sigma_2$ is defined recursively over these constructors for substitutions σ_1 and σ_2. By $e\sigma$ we denote the application of the substitution σ to the expression e, defined inductively by

$$x\sigma = \sigma(x)$$
$$\mathit{op}(e_1, \ldots, e_n)\sigma = \mathit{op}(e_1\sigma, \ldots, e_n\sigma)\ .$$

In the sequel, we will also use the alternative function application $\sigma(e)$ which denotes the homeomorphic extension of σ to expressions.

Let sequences ϕ over a set X be constructed from the empty sequence ε, the elements in X as singleton sequences, and by a concatenation operator $\phi_1 \cdot \phi_2$ applied to sequences ϕ_1 and ϕ_2.

We can now define *symbolic paths*, which combine the accumulated Boolean conditions from symbolic execution (Sect. 2.1) with the scheduling traces used for partial order reduction (Sect. 2.2), as follows:

Definition 2 (Symbolic Paths). *A* symbolic path ϕ *is a sequence of events $\iota\langle e, V_1, V_2\rangle$, where ι is a thread identifier, e a predicate and $V_1, V_2 \subseteq \mathit{VarSet}$.*

$$
\begin{aligned}
cn \in \mathit{Configuration} &::= ss \mid cn\ cn \\
ss \in \mathit{SymState} &::= (\!|\sigma, \phi, \Sigma|\!) \\
\tau \in \mathit{Thread} &::= \iota(s) \\
\sigma \in \mathit{Subst} &::= \varepsilon \mid \sigma[x \mapsto e] \\
\phi \in \mathit{SymPath} &::= \varepsilon \mid \iota\langle e, V, V\rangle \mid \phi \cdot \phi \\
\Sigma \in \mathit{ThreadPool} &::= \{\tau\} \mid \Sigma \cup \Sigma \\
V \in \mathit{VarSet} &::= \{x\} \mid V \cup V
\end{aligned}
$$

Fig. 4: Runtime syntax for the symbolic semantics, where ι is a thread identifier, e an expression and s a statement.

The predicates e will be used to capture path conditions, and the sets V_1 and V_2 the write and read effects associated with a program statement. Let $\lceil\phi\rceil$ denote the *path condition* corresponding to a symbolic path ϕ, given as the conjunction of the predicates in the events of ϕ. For a substitution σ and a symbolic path ϕ, let $\sigma(\phi)$ denote the symbolic path obtained by the pointwise application of σ to all expressions in the events of ϕ.

Next we describe a transition system for the symbolic execution of *PL* based on a transition relation between symbolic configurations, which are *sets* of *symbolic states*. These configurations are given by the grammar in Fig. 4.

Definition 3 (Symbolic state). *A* symbolic state *ss is a term* $(\!|\sigma, \phi, \Sigma|\!)$*, where*

- σ *is a symbolic substitution,*
- ϕ *is a symbolic path, and*
- Σ *is a thread pool.*

A configuration cn consists of a set of symbolic states. Given a substitution σ' and a configuration cn, let $\sigma'(cn)$ denote the pointwise application of substitution σ' to each symbolic state in cn, such that $\sigma'((\!|\sigma, \phi, \Sigma|\!)) = (\!|\sigma' \circ \sigma, \sigma'(\phi), \Sigma|\!)$.

Fig. 5 defines the symbolic semantics for *PL*, by means of a (symbolic) transition relation $\rightarrow_s$ between configurations of symbolic states. As usual, $cn \rightarrow_s^* cn'$ denotes that cn' is reachable from cn by a finite number of transitions of $\rightarrow_s$. Transitions may be labelled by the identifier of the thread which is reduced; thus, $\xrightarrow{\iota}_s$ describes an atomic execution step by a specific thread ι. The rules Assign, Cond and While respectively describe the symbolic executions of assignments, conditional and `while`-statements by a single thread. An assignment in thread ι is captured by the rule Assign, where the substitution σ is updated with the symbolic value $e\sigma$ for x and the symbolic path ϕ is extended with the event $\iota\langle \mathtt{True}, \{x\}, \mathit{vars}(e)\rangle$, indicating that an assignment has as path condition `True`, that there is a write to the variable x and there are reads from the variables in the expression e. The symbolic execution of both conditional and `while`-statements generates *two* symbolic states which correspond to the two different outcomes of the evaluation of the Boolean condition. A conditional statement in thread ι is captured by rule Cond, which reduces the statement to either s_1 or s_2, depending on the predicates $b\sigma$ and $\neg b\sigma$. This rule abstracts the two conditional

$$\frac{\{\iota_1, \ldots, \iota_n\} = T(\Sigma) \quad (\!|\sigma, \phi, \Sigma|\!) \xrightarrow{\iota_i}_s cn_i \quad 1 \leq i \leq n}{(\!|\sigma, \phi, \Sigma|\!) \rightarrow_s cn_1 \ldots cn_n}\ (\textsc{Schedule})$$

$$\frac{cn_1 \rightarrow_s cn_1'}{cn_1\ cn_2 \rightarrow_s cn_1'\ cn_2}\ (\textsc{Context})$$

$$\frac{\phi' = \phi{\cdot}\iota\langle \texttt{True}, \{x\}, vars(e)\rangle}{\begin{array}{c}(\!|\sigma, \phi, \{\iota(x := e; s)\} \cup \Sigma|\!) \\ \xrightarrow{\iota}_s (\!|\sigma[x \mapsto e\sigma], \phi', \{\iota(s)\} \cup \Sigma|\!)\end{array}}\ (\textsc{Assign})$$

$$\frac{\begin{array}{c}\phi_1 = \phi{\cdot}\iota\langle b\sigma, \emptyset, vars(b)\rangle \\ \phi_2 = \phi{\cdot}\iota\langle \neg b\sigma, \emptyset, vars(b)\rangle\end{array}}{\begin{array}{c}(\!|\sigma, \phi, \{\iota(\texttt{if}\ b\ \{s_1\}\ \texttt{else}\ \{s_2\}; s)\} \cup \Sigma|\!) \\ \xrightarrow{\iota}_s (\!|\sigma, \phi_1, \{\iota(s_1; s)\} \cup \Sigma|\!) \\ (\!|\sigma, \phi_2, \{\iota(s_2; s)\} \cup \Sigma|\!)\end{array}}\ (\textsc{Cond})$$

$$\frac{\begin{array}{c}\phi_1 = \phi{\cdot}\iota\langle b\sigma, \emptyset, vars(b)\rangle \\ \phi_2 = \phi{\cdot}\iota\langle \neg b\sigma, \emptyset, vars(b)\rangle\end{array}}{\begin{array}{c}(\!|\sigma, \phi, \{\iota(\texttt{while}\ b\ \{s_1\}; s_2)\} \cup \Sigma|\!) \\ \xrightarrow{\iota}_s (\!|\sigma, \phi_1, \{\iota(s_1; \texttt{while}\ b\ \{s_1\}; s_2)\} \cup \Sigma|\!) \\ (\!|\sigma, \phi_2, \{\iota(s_2)\} \cup \Sigma|\!)\end{array}}\ (\textsc{While})$$

Fig. 5: A symbolic semantics for PL with symbolic paths.

rules of the concrete semantics by resulting in two possible symbolic states. The `while`-statement is handled analogously in rule While.

The rule Schedule introduces branching by generating *all* symbolic states reachable in one step from a symbolic state with thread pool Σ, by applying a transition rule for each thread ι_i in Σ and by collecting the corresponding resulting configurations cn_i. The rule Context *lifts* transitions starting from a single symbolic state to transitions between configurations themselves (i.e., configurations should be interpreted modulo the reordering of the states). Remark that the Context rule introduces branching because different symbolic states can be chosen, but this branching is trivially confluent.

Example 1 (Symbolic execution). This example illustrates the use of the symbolic transition system. Consider a program with two threads:

$$\texttt{proc}\ \{x := 0; x := y + 1; s\} \qquad \texttt{proc}\ \{\texttt{if}\ (x = 0)\ \{s_1\}\ \texttt{else}\ \{s_2\}\},$$

where s, s_1 and s_2 are arbitrary statements (possibly sequentially composed and nested). We construct an initial configuration with one symbolic state

$$(\!|\sigma, \varepsilon, \{\iota_1(x := 0; x := y + 1; s), \iota_2(\texttt{if}\ (x = 0)\ \{s_1\}\ \texttt{else}\ \{s_2\})\}|\!),$$

where $\sigma = [x \mapsto x_0, y \mapsto y_0]$ and the two threads are identified by ι_1 and ι_2 respectively. With only one symbolic state, we apply the rule Schedule, which reduces thread ι_1 by means of rule Assign and thread ι_2 by means of rule Cond,

and obtain a configuration with three symbolic states:

$$(\!|\sigma[x \mapsto 0], \iota_1\langle \texttt{True}, \{x\}, \emptyset\rangle, \{\iota_1(x := y + 1; s), \iota_2(\texttt{if } (x = 0)\ \{s_1\}\ \texttt{else}\ \{s_2\})\}|\!)$$
$$(\!|\sigma, \iota_2\langle x_0 = 0, \emptyset, \{x\}\rangle, \{\iota_1(x := 0; x := y + 1; s), \iota_2(s_1)\}|\!)$$
$$(\!|\sigma, \iota_2\langle x_0 \neq 0, \emptyset, \{x\}\rangle, \{\iota_1(x := 0; x := y + 1; s), \iota_2(s_2)\}|\!)$$

The execution can now progress in any of the three symbolic states, where one is arbitrarily chosen by the rule Context. Let us select the first symbolic state. Since there are two threads which can be scheduled in this symbolic state and the Cond executes both branches of ι_2, application of the rule Schedule results in a configuration with five symbolic states:

$$(\!|(\sigma[x \mapsto 0])[x \mapsto y_0 + 1], \iota_1\langle \texttt{True}, \{x\}, \emptyset\rangle \cdot \iota_1\langle \texttt{True}, \{x\}, \{y\}\rangle,$$
$$\{\iota_1(s), \iota_2(\texttt{if } (x = 0)\ \{s_1\}\ \texttt{else}\ \{s_2\})\}|\!)$$
$$(\!|\sigma[x \mapsto 0], \iota_1\langle \texttt{True}, \{x\}, \emptyset\rangle \cdot \iota_2\langle x_0 = 0, \emptyset, \{x\}\rangle, \{\iota_1(x := y + 1; s), \iota_2(s_1)\}|\!)$$
$$(\!|\sigma[x \mapsto 0], \iota_1\langle \texttt{True}, \{x\}, \emptyset\rangle \cdot \iota_2\langle x_0 \neq 0, \emptyset, \{x\}\rangle, \{\iota_1(x := y + 1; s), \iota_2(s_2)\}|\!)$$
$$(\!|\sigma, \iota_2\langle x_0 = 0, \emptyset, \{x\}\rangle, \{\iota_1(x := 0; x := y + 1; s), \iota_2(s_1)\}|\!)$$
$$(\!|\sigma, \iota_2\langle x_0 \neq 0, \emptyset, \{x\}\rangle, \{\iota_1(x := 0; x := y + 1; s), \iota_2(s_2)\}|\!)$$

Thus, the application of the rules Context and Schedule alternate, the former introduces branching in the execution tree since any symbolic state may be selected and the latter introduces additional symbolic states in a configuration. This alternation continues until the threads identified by ι_1 and ι_2 have both terminated.

Symbolic transitions can be adapted by means of a symbolic substitution:

Proposition 2 (Framing). *For any $\sigma \in Subst$, $ss \in SymState$, and $cn \in Configuration$, we have that if $ss \to_s cn$ then $\sigma(ss) \to_s \sigma(cn)$.*

This proposition follows by a straightforward induction on the length of the symbolic execution. □

Initial and final symbolic states. Given a program $Pr = \texttt{proc}\ \{s_1\} \ldots \texttt{proc}\ \{s_n\}$, the *initial state* ss_0 for the symbolic execution of Pr is given by a symbolic configuration with a single symbolic state $(\!|\sigma, \varepsilon, init(Pr)|\!)$, where $\sigma(x) = x$, for all $x \in VarSet$. The execution of a program terminates when $T(\Sigma) = \emptyset$ for all symbolic states $(\!|\sigma, \phi, \Sigma|\!)$ in the configuration cn (technically, we may add a termination marker as a statement in the runtime syntax). For notational convenience, we denote such final symbolic states by $(\!|\sigma, \phi, \emptyset|\!)$.

For any computation $ss_0 \to_s^* cn$, starting from the initial symbolic state ss_0, the symbolic states $ss' \in cn$ represent symbolic states which are *reachable* by a particular scheduling of the threads and a particular symbolic evaluation of the Boolean conditions $\lceil \phi \rceil$ of the conditional and `while`-statements, as recorded by the symbolic path ϕ in ss'. This is captured by the following proposition which relates the above notion of reachability and the following notion of a *symbolic*

execution: A symbolic execution consists of a sequence $ss_0, \ldots, ss_n$ of symbolic states such that ss_0 denotes the initial symbolic state (as defined above), and for $0 \leq i < n$ there exist symbolic configurations cn_{i+1} and transitions $ss_i \rightarrow_s cn_{i+1}$ such that $ss_{i+1} \in cn_{i+1}$. Note that the reachability of ss_n only states the existence of configurations cn_i, for $0 \leq i \leq n$, such that $cn_0 = ss_0$, $cn_i \rightarrow_s cn_{i+1}$, $0 \leq i < n$, and $ss_n \in cn_n$.

Proposition 3 (Reachability for symbolic executions). *A symbolic state ss is reachable if and only if there exists a symbolic execution $ss_0, \ldots, ss_n$ such that $ss_n = ss$.*

This proposition follows by a straightforward induction on the length of the symbolic execution. □

3.4 Correctness

We prove correctness with respect to the concrete semantics from Sec. 3.2. Correctness then roughly consists of showing that for every symbolic execution there exists a corresponding concrete execution. We can now introduce the following formulation of the correctness theorem, where the valuation $\epsilon \circ \sigma$, which consists of a composition of the valuation ϵ and the substitution σ, is defined by $\epsilon \circ \sigma(x) = \epsilon(\sigma(x))$.

Theorem 2 (Correctness). *For every symbolic execution $ss_0, \ldots, ss_n$ such that $\epsilon_0(\lceil \phi_n \rceil) = \text{true}$, where ϵ_0 is a valuation and ϕ_n the symbolic path of ss_n, there exists a concrete computation $cs_0, \ldots, cs_n$ such that, for $0 \leq i \leq n$, the states ss_i and cs_i have the same thread pool, and $\epsilon_i = \epsilon_0 \circ \sigma_i$, where ϵ_i denotes the valuation of cs_i and σ_i denotes the substitution recorded by ss_i.*

Proof. The proof proceeds by induction on the length of the symbolic computation and a case analysis of the last transition. As such, it consists of a straightforward extension of the correctness proof of the symbolic execution of the underlying sequential programming language (consisting of assignments, conditional and `while`-statements, and their sequential composition), as given in [8]. □

Corollary 1. *Let $(\!|\sigma_1, \phi_1, \Sigma_1|\!) \rightarrow_s^* (\!|\sigma_n, \phi_n, \Sigma_n|\!)$ cn be a symbolic transition sequence and ϵ a valuation such that $\epsilon(\lceil \phi_n \rceil)$. Then there exists a concrete transition sequence $(\!|\epsilon_1, \Sigma_1|\!) \rightarrow_c^* (\!|\epsilon_n, \Sigma_n|\!)$ such that $\epsilon_i = \epsilon \circ \sigma_i$ for all $1 \leq i \leq n$.*

This corollary follows directly from Theorem 2 and Proposition 2. □

The converse of Theorem 2 states that for every concrete execution there exists a symbolic one.

Theorem 3 (Completeness). *For any concrete computation $cs_0, \ldots, cs_n$, there exists a symbolic execution $ss_0, \ldots, ss_n$ such that for $0 \leq i \leq n$ the states ss_i and cs_i have the same thread pool, and $\epsilon_i = \epsilon_0 \circ \sigma_i$, where ϵ_i denotes the valuation of cs_i and σ_i denotes the substitution recorded by ss_i.*

The proof of the theorem proceeds by a straightforward induction on the length of the concrete computation and a case analysis of the last transition. □

Corollary 2. *Let* $(\!|\epsilon_1, \Sigma_1|\!) \to_c^* (\!|\epsilon_n, \Sigma_n|\!)$ *be a concrete transition sequence and* ϵ *a valuation such that* $\epsilon_1 = \epsilon \circ \sigma_1$*. Then there exists a symbolic transition sequence* $(\!|\sigma_1, \phi_1, \Sigma_1|\!) \to_s^* (\!|\sigma_n, \phi_n, \Sigma_n|\!)$ *cn such that* $\epsilon_n = \epsilon \circ \sigma_n$ *for all* $1 \leq i \leq n$*.*

This proposition follows directly from Theorem 3 and Proposition 2. □

3.5 Partial Order Reduction

In order to provide a syntactic characterization of POR in the transition system of Fig. 5, we first define an equivalence-preserving permutation as an equivalence relation between symbolic paths. For this purpose, we use a syntactic criterion for interference freedom [14, 15], based on the disjointness of read- and write-variables between the two events [13].

Definition 4 (Interference Freedom & Path Equivalence). *Given two events* $ev_1 = \iota_1\langle e_1, W_1, R_1\rangle$ *and* $ev_2 = \iota_2\langle e_2, W_2, R_2\rangle$*. The* interference freedom *of the events* ev_1 *and* ev_2*, denoted* $ev_1 \sim ev_2$*, is defined as follows:*

$$ev_1 \sim ev_2 \iff \iota_1 \neq \iota_2 \text{ and } (R_1 \cap W_2) = (W_1 \cap R_2) = (W_1 \cap W_2) = \emptyset.$$

Let ϕ_1 *and* ϕ_2 *be symbolic paths. Denoted by* $\sim$ *the smallest equivalence relation on symbolic paths such that*

$$ev_1 \sim ev_2 \text{ implies } \phi_1 \cdot ev_1 \cdot ev_2 \cdot \phi_2 \sim \phi_1 \cdot ev_2 \cdot ev_1 \cdot \phi_2$$

We have the following semantical justification of the above definition.

Proposition 4 (Confluence). *Let*

$$(\!|\sigma, \phi, \{\tau_1, \tau_2\} \cup \Sigma|\!) \xrightarrow{\iota_1}_s (\!|\sigma_1, \phi \cdot ev_1, \{\tau_1', \tau_2\} \cup \Sigma|\!)\ cn_1$$

and

$$(\!|\sigma, \phi, \{\tau_1, \tau_2\} \cup \Sigma|\!) \xrightarrow{\iota_2}_s (\!|\sigma_2, \phi \cdot ev_2, \{\tau_1, \tau_2'\} \cup \Sigma|\!)\ cn_2$$

be two symbolic transitions such that $ev_1 \sim ev_2$*. Then there exists a substitution* σ' *such that*

$$(\!|\sigma_1, \phi \cdot ev_1, \{\tau_1', \tau_2\} \cup \Sigma|\!) \xrightarrow{\iota_2}_s (\!|\sigma', \phi \cdot ev_1 \cdot ev_2, \{\tau_1', \tau_2'\} \cup \Sigma|\!)\ cn_2$$

and

$$(\!|\sigma_2, \phi \cdot ev_2, \{\tau_1, \tau_2'\} \cup \Sigma|\!) \xrightarrow{\iota_1}_s (\!|\sigma', \phi \cdot ev_2 \cdot ev_1, \{\tau_1', \tau_2'\} \cup \Sigma|\!)\ cn_1$$

Proof. It suffices to consider the following cases of the events ev_1 and ev_2. The case where $ev_1 = \iota_1\langle b_1\sigma, \emptyset, vars(b_1)\rangle$ and $ev_2 = \iota_2\langle b_2\sigma, \emptyset, vars(b_2)\rangle$ follows immediately from the symbolic semantics of conditional and `while`-statements. Next,

let $ev_1 = \iota_1\langle \mathtt{True}, \{x\}, \mathit{vars}(e)\rangle$ and $ev_2 = \iota_2\langle b\sigma, \emptyset, \mathit{vars}(b)\rangle$. By the symbolic semantics of assignments, it follows that $\sigma_1 = \sigma[x \mapsto \sigma(e)]$. Since $ev_1 \sim ev_2$, we have that $x \notin \mathit{vars}(b)$, and thus $b\sigma$ (syntactically) equals $b\sigma_1$. As the last case, let $ev_1 = \iota_1\langle \mathtt{True}, \{x_1\}, \mathit{vars}(e_1)\rangle$ and $ev_2 = \iota_2\langle \mathtt{True}, \{x_2\}, \mathit{vars}(e_2)\rangle$. Furthermore, let $\sigma_1 = \sigma[x_1 \mapsto \sigma(e_1)]$ and $\sigma_2 = \sigma[x_2 \mapsto \sigma(e_2)]$. Since $ev_1 \sim ev_2$, we have that x_1 and x_2 are distinct variables, $x_1 \notin \mathit{vars}(e_2)$, and $x_2 \notin \mathit{vars}(e_1)$. It follows that for $i \neq j \in \{1, 2\}$, $e_i\sigma_j$ equals the expression $e_i\sigma$, and $\sigma_i[x_j \mapsto \sigma_i(e_j)]$ equals the *simultaneous* substitution $\sigma[x_1, x_2 \mapsto e_1\sigma, e_2\sigma]$. □

The following corollary of Proposition 4 states that equivalent symbolic paths uniquely determine the substitution and thread pool of a reachable symbolic state.

Corollary 3 (Determinism). *For any reachable symbolic states* $(\!|\sigma, \phi, \Sigma|\!)$ *and* $(\!|\sigma', \phi', \Sigma'|\!)$, $\phi \sim \phi'$ *implies* $\sigma = \sigma'$ *and* $\Sigma = \Sigma'$.

The equivalence relation $\sim$ is then trivially extended to symbolic states as follows: if $\phi \sim \phi'$ then $(\!|\sigma, \phi, \Sigma|\!) \sim (\!|\sigma, \phi', \Sigma|\!)$. We can now apply a partial order reduction to the symbolic transition system by simply lifting it to the equivalence classes of symbolic states. Let $[cn]$ denote the equivalence classes of the symbolic states of the symbolic configuration cn. In addition, let γ be a selection function such that $[cn]_\gamma$ denotes the symbolic configuration which results from selecting from each of the equivalence classes of the symbolic states of the symbolic configuration cn a (canonical) representative. We then have the following partial order reduction rule.

$$\frac{[cn]_\gamma \rightarrow_s cn'}{cn \rightarrow_{por_\gamma} cn'} \ (\mathrm{POR}_\gamma)$$

Correctness and completeness of the partial order reduction rule with respect to the symbolic transition system follows from the above corollary which ensures that the partial order reduction rule is independent of the choice of the representatives of the equivalence classes of symbolic states. This is formalized by the following proposition.

Proposition 5 (Bisimulation). *The transition systems* $\rightarrow_s$ *and* $\rightarrow_{\mathrm{por}_\gamma}$ *are bisimilar with respect to the equivalence relation* $\sim$*: Let* cn_1 *and* cn_2 *be two symbolic configurations such that* $[cn_1] = [cn_2]$*. Then for any transition* $cn_1 \rightarrow_s cn_1'$*, there exists a transition* $cn_2 \rightarrow_{\mathrm{por}_\gamma} cn_2'$ *such that* $[cn_1'] = [cn_2']$*, and vice versa.*

Proof. It suffices to prove that $ss \sim ss'$ and $ss \xrightarrow{\iota}_s cn$ implies that there exists a cn' such that $ss' \xrightarrow{\iota}_s cn'$ and $[cn] = [cn']$, which follows from a straightforward case analysis of the transition $ss \xrightarrow{\iota}_s cn$. □

A symbolic state ss is POR_γ reachable if there exists a symbolic configuration cn such that $ss_0 \rightarrow^*_{por_\gamma} cn$ and $ss \in cn$ (where ss_0 denotes the initial symbolic state, as defined above).

Corollary 4 (Correctness of POR). *For any symbolic state ss that is POR_γ reachable, there exists an equivalent symbolic state $ss' \sim ss$ which is $\rightarrow_s$ reachable (i.e., there exists a symbolic configuration cn such that $ss_0 \rightarrow_s^* cn$ and $ss' \in cn$).*

Proof. Induction on the length of the computation $ss_0 \rightarrow_{por_\gamma}^* cn$, where $ss' \in cn$, using the above proposition. □

Corollary 5. *For any POR computation $(\!|\sigma, \phi, \Sigma|\!) \rightarrow_{por_\gamma}^* (\!|\sigma', \phi', \Sigma'|\!)$ cn, there exists a symbolic computation $(\!|\sigma, \phi, \Sigma|\!) \rightarrow_s^* (\!|\sigma', \phi', \Sigma'|\!)$ cn.*

Proof. The proof follows directly from Corollary 4 and Proposition 2. □

Corollary 6 (Completeness of POR). *For any symbolic state ss that is $\rightarrow_s$ reachable, there exists an equivalent symbolic state $ss' \sim ss$ which is POR_γ reachable.*

Proof. Induction on the length of the computation $ss_0 \rightarrow_s^* cn$, where $ss' \in cn$, using the above proposition. □

Corollary 7. *For any symbolic computation $(\!|\sigma, \phi, \Sigma|\!) \rightarrow_s^* (\!|\sigma', \phi', \Sigma'|\!)$ cn, there exists a POR computation $(\!|\sigma, \phi, \Sigma|\!) \rightarrow_{por_\gamma}^* (\!|\sigma', \phi', \Sigma'|\!)$ cn.*

Proof. The proof follows directly from Corollary 6 and Proposition 2. □

Note that Corollary 7 holds because we are free in the choice of γ. In Sect. 5, we fix a particular selection function, meaning all symbolic *states*, but not all symbolic *computations*, will be reachable.

4 Dynamic Logic for Multithreaded Programs

This section demonstrates how the integration of symbolic execution and partial order reduction presented in Sect. 3 can be put to use to define a dynamic logic proof system for multithreaded programs.

4.1 Dynamic Logic

The formulas of dynamic logic are defined inductively, starting from the set *BExp* of Boolean expressions.

Definition 5 (DL Formulas). *Given a set BExp of Boolean expressions over variables $x \in VarSet$, symbolic substitutions $\sigma \in Subst$ and thread pools $\Sigma \in ThreadPool$, the dynamic logic formulas are defined inductively as the smallest set $\boldsymbol{DL}$ such that*

1. $\Psi \in \boldsymbol{DL}$ *if* $\Psi \in BExp$
2. $\neg\Psi_1, \Psi_1 \wedge \Psi_2, \Psi_1 \vee \Psi_2, \Psi_1 \rightarrow \Psi_2, \Psi_1 \leftrightarrow \Psi_2 \in \boldsymbol{DL}$ *if* $\Psi_1, \Psi_2 \in \boldsymbol{DL}$
3. $\exists x \cdot \Psi, \forall x \cdot \Psi \in \boldsymbol{DL}$ *if* $x \in VarSet$ *and* $\Psi \in \boldsymbol{DL}$
4. $[\Sigma]\Psi \in \boldsymbol{DL}$ *if* $\Psi \in \boldsymbol{DL}$

5. $\{\sigma\}\Psi \in \boldsymbol{DL}$ *if* $\sigma \in \mathit{Subst}$ *and* $\Psi \in \boldsymbol{DL}$

The set **DL** extends the set **FOL** of first-order logic formulas with two *modalities* $[\Sigma]\Psi$ and $\{\sigma\}\Psi$, which we refer to as the box-modality and the update-modality, respectively. (Note that the update modality here features a symbolic substitution σ, which differs slightly from the symbolic updates in KeY. Symbolic substitutions may be implemented using symbolic updates.) A formula $\Psi \in \mathbf{DL}$ is called *first-order* if it does not contain any modality (i.e., $\Psi \in \mathbf{FOL}$). For simplicity, we may write $[Pr]\Psi$ to express $[\mathit{init}(Pr)]\Psi$, where $Pr \in \mathit{Prog}$.

We write $\epsilon \models \Psi$ to denote that a formula Ψ is valid in a valuation ϵ (or that ϵ satisfies Ψ). The satisfiability of a DL formula can be defined as follows (e.g., [10]):

Definition 6 (Satisfiability of DL Formulas). *Let* $\Psi \in \boldsymbol{DL}$, $e \in \mathit{BExp}$ *and* ϵ *a valuation. Satisfiability is defined inductively as follows:*

$$
\begin{aligned}
&\epsilon \models e \iff \epsilon(e) = \top \\
&\epsilon \models \neg\Psi \iff \epsilon \not\models \Psi \\
&\epsilon \models \Psi_1 \wedge \Psi_2 \iff \epsilon \models \Psi_1 \text{ and } \epsilon \models \Psi_2 \\
&\epsilon \models \Psi_1 \vee \Psi_2 \iff \epsilon \models \Psi_1 \text{ or } \epsilon \models \Psi_2 \\
&\epsilon \models \exists x \cdot \Psi \iff \epsilon[x \mapsto \epsilon(e)] \models \Psi \text{ for some } e \in \mathit{Exp} \\
&\epsilon \models \forall x \cdot \Psi \iff \epsilon[x \mapsto \epsilon(e)] \models \Psi \text{ for all } e \in \mathit{Exp} \\
&\epsilon \models [\Sigma]\Psi \iff \epsilon' \models \Psi \text{ for all } \epsilon' \text{ such that } (\!|\epsilon, \Sigma|\!) \to_c^* (\!|\epsilon', \emptyset|\!) \\
&\epsilon \models \{\sigma\}\Psi \iff \epsilon' \models \Psi \text{ and } \epsilon' = \epsilon \circ \sigma
\end{aligned}
$$

Operators for implication and equivalence can be defined as usual in terms of negation and disjunction. This definition of the box-modality $[\Sigma]\Psi$ corresponds to *partial correctness* in the sense that the formula only needs to hold for all execution paths that lead to a final state; in particular, $[\Sigma]\Psi$ is true if the execution of Σ never terminates. The update modality $\{\sigma\}\Psi$ corresponds to executing a sequence of assignments. Observe that $\{\sigma\}\{\sigma'\}\Psi \leftrightarrow \{\sigma \circ \sigma'\}\Psi$.

4.2 A DL Sequent Calculus for Multithreaded Programs

We first define a sequent calculus for proving DL formulas over multithreaded Pr programs, and then transform it into a transition system similar to $\to_s$.

A *sequent* of the form $\Gamma \Rightarrow \Psi$, where Γ and Ψ are sets of formulas, expresses that the conjunction of the formulas in the antecedent Γ implies the disjunction of the formulas in the succedent Ψ; a sequent $\Gamma \Rightarrow \Psi$ is valid, denoted $\models \Gamma \Rightarrow \Psi$ if and only if its corresponding implication is valid.

The sequent calculus is given by the rules in Fig. 6. To prove a sequent, one constructs a proof tree with the sequent as an open goal at the root by repeatedly applying rules to open proof goals until all open goals have been closed. The application of a rule to an open goal closes the goal and extends the tree with new open goals corresponding to the premisses of the rule. An axiom closes an open goal. A particular feature of this calculus is that sequents are labelled, such that the thread which is symbolically executed in the rules

DL-Assign, DL-Cond and DL-While is determined by the label. Rule DL-Schedule forces all threads to be explored in the construction of the proof tree. We omit standard rules for first-order logic (e.g., [2, Ch. 2]), and assume that we can determine whether sequents of the form $\Gamma \Rightarrow \Psi$ hold, when Γ, Ψ consist of formulas in **FOL**. Consequently, we can decide whether a branch can be closed by checking if the formulas in the antecedent Γ are inconsistent or by means of the rule DL-Reduce, which reduces the sequent to first-order formulas by applying the substitution σ to Ψ (assuming Ψ consists of formulas in **FOL**, otherwise one of the presented rules applies).

To show that a DL formula Ψ is true for a program Pr, we need to construct a proof by applying the rules of the sequent calculus, starting from the sequent $\Gamma \Rightarrow \{\sigma\}[\mathit{init}(Pr)]\Psi$, where $x\sigma = x$ for all $x \in \mathit{VarSet}$. Observe that due to the interleaving of threads, we may need to prove many equivalent sequents in different nodes of the proof tree. These sequents differ only in the order in which steps from different non-interfering threads have been selected in the branches leading to the nodes.

In order to eliminate such redundant nodes during the proof construction, we now consider a path-sensitive reformulation of this calculus. For this purpose, we introduce *path-sensitive sequents*, a variation of the sequents above, in the form $\Gamma, \phi \Rightarrow \Psi$, where ϕ is a symbolic path as defined in Sect. 3. The correspondence with the standard sequents is obtained by lifting the symbolic path to, e.g., $\Gamma, \lceil\phi\rceil \Rightarrow \Psi$ for the sequent above. We consider a transition relation $\rightarrow_{dl}$ between sets Ω of path-sensitive sequents that need to be proven, corresponding to the open leaf nodes of the proof tree constructed by the sequent calculus above. The rules of the transition relation $\rightarrow_{dl}$ are presented in Fig. 7. Each transition in the proof system takes a sequent from Ω and replaces it with some new sequents. As before, we omit the standard rules for first-order logic; we remove sequents that correspond to closed leaf nodes in the proof trees, such that a closed proof tree in the proof calculus here corresponds to an empty set of sequents. Thus, a sequent $\Gamma, \phi \Rightarrow \Psi$ has a proof if $\Gamma, \phi \Rightarrow \Psi \rightarrow_{dl}^{*} \emptyset$ (or if the formulas in $\Gamma, \lceil\phi\rceil$ are inconsistent). Observe that as a consequence of the representation, we have a linear sequence of proof steps rather than a proof tree.

The rules of the transition system in Fig. 7 correspond closely to the sequent calculus of Fig. 6. For each rule of the sequent calculus, the corresponding transition rule is obtained by letting the left-hand-side of the transition be the conclusion of the proof rule and the right-hand-side its set of premises. The remaining transition rule DLT-Context corresponds in the transition system to the selection of open nodes in the proof tree for the next construction step (this step is not formalized in the sequent calculus). Observe that the transition system $\rightarrow_{dl}$ is *locally confluent* as the result of applying DLT-Context to different sequents in a set Ω is unaffected by the order of these rule applications.

4.3 Soundness and Relative Completeness for the DL Proof System

Our goal is to establish that the DL proof system for multithreaded programs is sound and relative complete. For this purpose, we take the formulation as

$$\frac{\Gamma \Rightarrow \{\sigma\}\Psi}{\Gamma \Rightarrow \{\sigma\}[\emptyset]\Psi}\ \text{(DL-Reduce)}$$

$$\frac{\Gamma \Rightarrow \{\sigma[x \mapsto \sigma(e)]\}[\{\iota(s)\} \cup \Sigma]\Psi}{\Gamma \overset{\iota}{\Rightarrow} \{\sigma\}[\{\iota(x := e; s)\} \cup \Sigma]\Psi}\ \text{(DL-Assign)}$$

$$\frac{\{\iota_1, \ldots, \iota_n\} = T(\Sigma) \quad \Gamma \overset{\iota_1}{\Longrightarrow} \{\sigma\}[\Sigma]\Psi \ \cdots\ \Gamma \overset{\iota_n}{\Longrightarrow} \{\sigma\}[\Sigma]\Psi}{\Gamma \Rightarrow \{\sigma\}[\Sigma]\Psi}\ \text{(DL-Schedule)}$$

$$\frac{\begin{array}{c}\Gamma, b\sigma \Rightarrow \{\sigma\}[\{\iota(s_1; s)\} \cup \Sigma]\Psi \\ \Gamma, \neg b\sigma \Rightarrow \{\sigma\}[\{\iota(s_2; s)\} \cup \Sigma]\Psi\end{array}}{\begin{array}{l}\Gamma \overset{\iota}{\Rightarrow} \{\sigma\}[\{\iota(\texttt{if}\ b\ \{s_1\} \\ \qquad \texttt{else}\ \{s_2\}; s)\} \cup \Sigma]\Psi\end{array}}\ \text{(DL-Cond)}$$

$$\frac{\begin{array}{c}\Gamma, b\sigma \Rightarrow \{\sigma\}[\{\iota(s_1; \\ \qquad \texttt{while}\ b\ \{s_1\}; s_2)\} \cup \Sigma]\Psi \\ \Gamma, \neg b\sigma \Rightarrow \{\sigma\}[\{\iota(s_2)\} \cup \Sigma]\Psi\end{array}}{\Gamma \overset{\iota}{\Rightarrow} \{\sigma\}[\{\iota(\texttt{while}\ b\ \{s_1\}; s_2)\} \cup \Sigma]\Psi}\ \text{(DL-While)}$$

Fig. 6: A proof calculus for dynamic logic.

$$\frac{\Omega_1 \to_{dl} \Omega_1'}{\Omega_1\ \Omega_2 \to_{dl} \Omega_1'\ \Omega_2}\ \text{(DLT-Context)}$$

$$\frac{\Gamma, \phi \Rightarrow \{\sigma\}[\emptyset]\Psi}{\to_{dl} \Gamma, \lceil\phi\rceil \Rightarrow \{\sigma\}\Psi}\ \text{(DLT-Reduce)}$$

$$\frac{\phi' = \phi{\cdot}\iota\langle \texttt{True}, \{x\}, \mathit{vars}(e)\rangle}{\begin{array}{l}\Gamma, \phi \overset{\iota}{\Rightarrow} \{\sigma\}[\{\iota(x := e; s)\} \cup \Sigma]\Psi \\ \quad \to_{dl} \Gamma, \phi' \Rightarrow \{\sigma[x \mapsto \sigma(e)]\} \\ \qquad\qquad [\{\iota(s)\} \cup \Sigma]\Psi\end{array}}\ \text{(DLT-Assign)}$$

$$\frac{\{\iota_1, \ldots, \iota_n\} = T(\Sigma) \quad \Gamma, \phi \overset{\iota_i}{\Longrightarrow} \{\sigma\}[\Sigma]\Psi \to_{dl} \Omega_i \quad 1 \le i \le n}{\Gamma, \phi \Rightarrow \{\sigma\}[\Sigma]\Psi \to_{dl} \Omega_1 \ldots \Omega_n}\ \text{(DLT-Schedule)}$$

$$\frac{\begin{array}{c}\phi_1 = \phi{\cdot}\iota\langle b\sigma, \emptyset, \mathit{vars}(b)\rangle \\ \phi_2 = \phi{\cdot}\iota\langle \neg b\sigma, \emptyset, \mathit{vars}(b)\rangle\end{array}}{\begin{array}{l}\Gamma, \phi \overset{\iota}{\Rightarrow} \{\sigma\}[\{\iota(\texttt{if}\ b\ \{s_1\} \\ \qquad \texttt{else}\ \{s_2\}; s)\} \cup \Sigma]\Psi \\ \quad \to_{dl} \Gamma, \phi_1 \Rightarrow \{\sigma\}[\{\iota(s_1; s)\} \cup \Sigma]\Psi \\ \qquad \Gamma, \phi_2 \Rightarrow \{\sigma\}[\{\iota(s_2; s)\} \cup \Sigma]\Psi\end{array}}\ \text{(DLT-Cond)}$$

$$\frac{\begin{array}{c}\phi_1 = \phi{\cdot}\iota\langle b\sigma, \emptyset, \mathit{vars}(b)\rangle \\ \phi_2 = \phi{\cdot}\iota\langle \neg b\sigma, \emptyset, \mathit{vars}(b)\rangle\end{array}}{\begin{array}{l}\Gamma, \phi \overset{\iota}{\Rightarrow} \{\sigma\}[\{\iota(\texttt{while}\ b\ \{s_1\}; s_2)\} \cup \Sigma]\Psi \\ \quad \to_{dl} \Gamma, \phi_1 \Rightarrow \{\sigma\}[\{\iota(s_1; \\ \qquad \texttt{while}\ b\ \{s_1\}; s_2)\} \cup \Sigma]\Psi \\ \qquad \Gamma, \phi_2 \Rightarrow \{\sigma\}[\{\iota(s_2)\} \cup \Sigma]\Psi\end{array}}\ \text{(DLT-While)}$$

Fig. 7: A transition system with symbolic paths for dynamic logic.

a transition system (Fig. 7) as our starting point, and connect it to the symbolic execution framework defined in Sect. 3. In fact, the symbolic execution achieved by the transition relation $\rightarrow_{dl}$ corresponds exactly to that achieved by the transition relation $\rightarrow_s$, as expressed by the following lemma.

Lemma 1. *$\Gamma, \phi \Rightarrow \{\sigma\}[\Sigma]\Psi \rightarrow_{dl} \{\Gamma, \phi' \Rightarrow \{\sigma'\}[\Sigma']\Psi\} \cup \Omega$ if and only if $(\!|\{\sigma, \phi, \Sigma|\!) \rightarrow_s (\!|\{\sigma', \phi', \Sigma'|\!)$ cn.*

Proof. Follows by induction over the transition rules. For each rule in the transition system (Fig. 7), select the corresponding rule for symbolic execution, and vice versa. □

Lemma 1 allows soundness to be shown in terms of the soundness of the symbolic execution framework and the soundness of the underlying proof system for FOL.

Theorem 4 (Soundness). *Let $\Sigma \in$ ThreadPool, $\sigma \in$ Subst, $\phi \in$ SymPath and $\Gamma, \Psi \in$ **DL**. If $\Gamma, \phi \Rightarrow \{\sigma\}[\Sigma]\Psi \rightarrow_{dl}^* \emptyset$, then $\models \Gamma, \phi \Rightarrow \{\sigma\}[\Sigma]\Psi$.*

Proof. Without loss of generality, we may assume that Ψ consists of first-order formulas. Assume $\Gamma, \phi \Rightarrow \{\sigma\}[\Sigma]\Psi \rightarrow_{dl}^* \emptyset$. Then, due to the confluence of the transition system $\rightarrow_{dl}$, there is a derivation $\Gamma, \phi \Rightarrow \{\sigma\}[\Sigma]\Psi \rightarrow_{dl}^* \Omega$ such that $\Sigma' = \emptyset$ for all $\Gamma, \phi' \Rightarrow \{\sigma'\}[\Sigma']\Psi \in \Omega$. Then by applying DLT-Reduce, we get $\Gamma, \lceil\phi'\rceil \Rightarrow \{\sigma'\}\Psi \rightarrow_{dl}^* \emptyset$. This last reduction uses the proof system for FOL, which by assumption gives $\models \Gamma, \lceil\phi'\rceil \Rightarrow \{\sigma'\}\Psi$ for any concrete valuation.

Let ϵ be a valuation such that $\epsilon \models \Gamma, \lceil\phi'\rceil$, and by Def. 6, we have $\epsilon \models \{\sigma'\}\Psi$. By Def. 6 again, we have $\epsilon_2 \models \Psi$ where $\epsilon_2 = \epsilon \circ \sigma'$

By Lemma 1 and the assumption $\Gamma, \phi \Rightarrow \{\sigma\}[\Sigma]\Psi \rightarrow_{dl}^* \emptyset$, we get $(\!|\sigma, \phi, \Sigma|\!) \rightarrow_s^* (\!|\sigma', \phi', \emptyset|\!)$ cn. Given this and $\epsilon \models \lceil\phi'\rceil$ (implied by $\epsilon \models \Gamma, \lceil\phi'\rceil$), then by Corollary 1 we get $(\!|\{\epsilon_1, \Sigma|\!) \rightarrow_c^* (\!|\{\epsilon_2, \emptyset|\!)$ such that $\epsilon_1 = \epsilon \circ \sigma$ and $\epsilon_2 = \epsilon \circ \sigma'$. This together with $\epsilon_2 \models \Psi$ and Def. 6 gives $\epsilon_1 \models [\Sigma]\Psi$; consequently by Def. 6 again, we get $\epsilon \models \{\sigma\}[\Sigma]\Psi$. Finally, $\lceil\phi'\rceil$ implying $\lceil\phi\rceil$ gives $\epsilon \models \Gamma, \phi \Rightarrow \{\sigma\}[\Sigma]\Psi$. □

The relative completeness of the proof system can be shown in terms of the correctness of the symbolic execution framework and the relative completeness of the underlying proof system for FOL in a similar way.

Theorem 5 (Relative Completeness). *Let $\Sigma \in$ ThreadPool, $\sigma \in$ Subst, $\phi \in$ SymPath and $\Gamma, \Psi \in$ **DL**. If $\models \Gamma, \phi \Rightarrow \{\sigma\}[\Sigma]\Psi$, then $\Gamma, \phi \Rightarrow \{\sigma\}[\Sigma]\Psi \rightarrow_{dl}^* \emptyset$.*

Proof. Since we aim to show *relative completeness*, we assume the completeness of the underlying first-order logic proof system; i.e., $\models \psi$ implies $\psi \rightarrow_{dl}^* \emptyset$ for any first-order formula ψ. Moreover, without loss of generality, we may assume that Ψ consists of only first-order formulas.

Assume $\epsilon \models \Gamma, \phi \Rightarrow \{\sigma\}[\Sigma]\Psi$ for some ϵ, and assume further $\epsilon \models \Gamma \wedge \lceil\phi\rceil$. This gives $\epsilon \models \{\sigma\}[\Sigma]\Psi$. By Def. 6, we have $\epsilon' \models [\Sigma]\Psi$ where $\epsilon' = \epsilon \circ \sigma$. Then by Def. 6 again, we have $\epsilon'' \models \Psi$ for all valuations ϵ'' such that $(\!|\epsilon', \Sigma|\!) \rightarrow_c^* (\!|\epsilon'', \emptyset|\!)$.

Given this together with $\epsilon' = \epsilon \circ \sigma$ and by Corollary 2, there exist a symbolic computation $(\!|\sigma, \phi, \Sigma|\!) \rightarrow_s^* (\!|\sigma', \phi', \emptyset|\!)$ *cn* for some σ' and ϕ' such that $\epsilon'' = \epsilon \circ \sigma'$.

Since $\epsilon'' \models \epsilon \circ \sigma'$ gives $\epsilon \models \{\sigma'\}\Psi$, then by assuming $\epsilon \models \Gamma \wedge \lceil \phi' \rceil$, we get $\models \Gamma \wedge \lceil \phi' \rceil \Rightarrow \{\sigma'\}\Psi$. By Lemma 1, we have $\Gamma, \phi \Rightarrow \{\sigma\}[\Sigma]\Psi \rightarrow_{dl}^* \{\Gamma, \phi' \Rightarrow \{\sigma'\}[\emptyset]\Psi\} \cup \Omega$, where $\Gamma, \phi' \Rightarrow \{\sigma'\}[\emptyset]\Psi$ can be reduced to $\Gamma, \lceil \phi' \rceil \Rightarrow \{\sigma'\}\Psi$ by rule DLT-Reduce. Finally by the completeness of FOL that gives us $\Gamma, \phi' \Rightarrow \{\sigma'\}\Psi \rightarrow_{dl}^* \emptyset$, the theorem follows. □

Remark that the rule DLT-While here is based on unfolding the `while`-loop to emphasize its similarity with the rule While of Sect. 3.3. The rule could be lifted to reasoning about infinite unfoldings of the `while`-loop using a loop-invariant in the usual way; the proof of relative completeness would then require an assumption about a sufficiently strong invariant to prove reachability for all finite approximations of the infinite unfoldings.

4.4 Partial Order Reduction in Proof Search

We now lift the equivalence relation $\sim$ of Def. 4 to DL sequents as follows: if $\phi \sim \phi'$ then $\Gamma, \phi \Rightarrow \{\sigma\}[\Sigma]\Psi \sim \Gamma, \phi' \Rightarrow \{\sigma\}[\Sigma]\Psi$. Let $[\Omega]$ denote the equivalence classes of sets Ω of path-sensitive sequents. Let γ be a selection function such that $[\Omega]_\gamma$ denotes the path-sensitive sequent which results from selecting from each of the equivalence classes of path-sensitive sequents of the set Ω a (canonical) representative. This allows us to define the following partial order reduction rule for the proof construction in DL:

$$\frac{[\Omega]_\gamma \rightarrow_{dl} \Omega'}{[\Omega] \rightarrow_{dlpor_\gamma} [\Omega']} \text{(DL-POR)}$$

The transition system $\rightarrow_{dlpor_\gamma}$ is locally confluent as the result of applying (DLT-Context) to different sequents in a set $[\Omega]_\gamma$ is not affected by the order of these rule applications.

Remark that in the proof system we have formalized as $\rightarrow_{dlpor_\gamma}$, the state of the proof search is represented as a set Ω of open proof goals. The same ideas could also be formulated in a proof system which represents the proof using a more standard graph-like proof structure, in which each sequent in Ω corresponds to an open proof goal in a leaf node of the graph. If the symbolic paths are kept on a canonical form in the sequents (see Sect. 5), the rule DL-POR would correspond to a rule which merges nodes with different occurrences of the same proof goal, turning the proof into a directed acyclic graph.

Correctness and relative completeness of $\rightarrow_{dlpor_\gamma}$ can be shown following the proofs for Theorems 4 and 5 above. In fact, the symbolic execution achieved by the transition relation $\rightarrow_{dlpor_\gamma}$ corresponds exactly to that of the transition relation $\rightarrow_{por_{\gamma'}}$, as expressed by the following lemma.

Lemma 2. *$\Gamma, \phi \Rightarrow \{\sigma\}[\Sigma]\Psi \rightarrow_{dlpor_\gamma} \{\Gamma, \phi' \Rightarrow \{\sigma'\}[\Sigma']\Psi\} \cup \Omega$ if and only if $(\!|\{\sigma, \phi, \Sigma\}|\!) \rightarrow_{por_{\gamma'}} (\!|\{\sigma', \phi', \Sigma'\}|\!)$ cn.*

Proof. Follows by induction over the transition rules. For each rule in the transition system for $\rightarrow_{dlpor_\gamma}$, select the corresponding rule for$\rightarrow_{por_{\gamma'}}$, and vice versa. □

Lemma 2 allows soundness and relative completeness to be shown for $\rightarrow_{dlpor_\gamma}$, following the proof structure of Theorems 4 and 5 above.

Theorem 6 (Soundness). *Let $\Sigma \in ThreadPool$, $\sigma \in Subst$, $\phi \in SymPath$ and $\Gamma, \Psi \in \boldsymbol{DL}$. If $\Gamma, \phi \Rightarrow \{\sigma\}[\Sigma]\Psi \rightarrow^*_{dlpor_\gamma} \emptyset$, then $\models \Gamma, \phi \Rightarrow \{\sigma\}[\Sigma]\Psi$.*

Proof (Sketch). Since $\rightarrow_{dlpor_\gamma}$ is locally confluent, we can adapt the proof structure of Theorem 4. The proof then follows from Corollary 5 and Lemma 2. □

Theorem 7 (Relative Completeness). *Let $\Sigma \in ThreadPool$, $\sigma \in Subst$, $\phi \in SymPath$ and $\Gamma, \Psi \in \boldsymbol{DL}$. If $\models \Gamma, \phi \Rightarrow \{\sigma\}[\Sigma]\Psi$, then $\Gamma, \phi \Rightarrow \{\sigma\}[\Sigma]\Psi \rightarrow^*_{dlpor_\gamma} \emptyset$.*

Proof (Sketch). We adapt the proof structure of Theorem 5. The proof then follows from Corollary 7 and Lemma 2. □

5 Implementation

We have developed a prototype implementation of the transition systems $\rightarrow_s$ and $\rightarrow_{por_\gamma}$ from Sect. 3, in order to report on experimental results and validate the approach. The prototype is developed in Maude [11], which allows writing executable models in rewriting logic [16]. We do not cover the full details of the implementation[5] here, but focus on how it differs from the transition system presented in Sect. 3.

Given a program *Pr*, our implementation executes the program symbolically. The program terminates when all threads have terminated, resulting in a configuration of final symbolic states. In addition, we output statistics about how many symbolic states are in a configuration and the number of applications of the rule SCHEDULE needed to reach a configuration from the initial configuration, in order to compare the two systems $\rightarrow_s$ and $\rightarrow_{por_\gamma}$.

The implementation stays very close to the transition systems defined in Sect. 3. One notable difference is that while the rule POR_γ achieves partial order reduction by lifting the symbolic transition system to equivalence classes and by fixing a selection function γ to obtain the canonical representative, the implementation transforms all symbolic states to a canonical form. We achieve this by defining an ordering between *interference-free* events, and by choosing the smallest lexicographic representative. The ordering of interference-free events depends on the thread identifiers, which we assume to have a total order; in the implementation, these are represented as natural numbers.

Another difference is that the while statement in Sect. 3 is possibly unbounded, which can lead to an infinite number of paths in a symbolic setting. In order to get terminating executions for the examples in the prototype, the `while`-loop has been given an optional bound.

[5] The implementation is available at `https://github.com/larstvei/sympaths`.

Example 2. The following is a very simple program with three threads, all of which perform a single assignment.

$$\texttt{proc}\ \{y := x\} \qquad \texttt{proc}\ \{z := x\} \qquad \texttt{proc}\ \{x := 2\}$$

There are six final search states for this program in $\rightarrow_s$, with the following symbolic paths:

$$\begin{array}{l}
\iota_0\langle \texttt{True}, \{y\}, \{x\}\rangle \cdot \iota_1\langle \texttt{True}, \{z\}, \{x\}\rangle \cdot \iota_2\langle \texttt{True}, \{x\}, \emptyset\rangle \\
\iota_1\langle \texttt{True}, \{z\}, \{x\}\rangle \cdot \iota_0\langle \texttt{True}, \{y\}, \{x\}\rangle \cdot \iota_2\langle \texttt{True}, \{x\}, \emptyset\rangle \\
\iota_0\langle \texttt{True}, \{y\}, \{x\}\rangle \cdot \iota_2\langle \texttt{True}, \{x\}, \emptyset\rangle \cdot \iota_1\langle \texttt{True}, \{z\}, \{x\}\rangle \\
\iota_1\langle \texttt{True}, \{z\}, \{x\}\rangle \cdot \iota_2\langle \texttt{True}, \{x\}, \emptyset\rangle \cdot \iota_0\langle \texttt{True}, \{y\}, \{x\}\rangle \\
\iota_2\langle \texttt{True}, \{x\}, \emptyset\rangle \cdot \iota_0\langle \texttt{True}, \{y\}, \{x\}\rangle \cdot \iota_1\langle \texttt{True}, \{z\}, \{x\}\rangle \\
\iota_2\langle \texttt{True}, \{x\}, \emptyset\rangle \cdot \iota_1\langle \texttt{True}, \{z\}, \{x\}\rangle \cdot \iota_0\langle \texttt{True}, \{y\}, \{x\}\rangle
\end{array}$$

According to Definition 4, we have $\iota_1\langle \texttt{True}, \{z\}, \{x\}\rangle \sim \iota_0\langle \texttt{True}, \{y\}, \{x\}\rangle$. With partial order reduction in $\rightarrow_{por_\gamma}$, symbolic states that contain symbolic paths that only differ in the order of independent events can be *pruned* from the search. Executing the program in our implementation of in $\rightarrow_{por_\gamma}$ gives the following four symbolic paths, which cannot be reduced further.

$$\begin{array}{l}
\iota_0\langle \texttt{True}, \{y\}, \{x\}\rangle \cdot \iota_1\langle \texttt{True}, \{z\}, \{x\}\rangle \cdot \iota_2\langle \texttt{True}, \{x\}, \emptyset\rangle \\
\iota_0\langle \texttt{True}, \{y\}, \{x\}\rangle \cdot \iota_2\langle \texttt{True}, \{x\}, \emptyset\rangle \cdot \iota_1\langle \texttt{True}, \{z\}, \{x\}\rangle \\
\iota_1\langle \texttt{True}, \{z\}, \{x\}\rangle \cdot \iota_2\langle \texttt{True}, \{x\}, \emptyset\rangle \cdot \iota_0\langle \texttt{True}, \{y\}, \{x\}\rangle \\
\iota_2\langle \texttt{True}, \{x\}, \emptyset\rangle \cdot \iota_0\langle \texttt{True}, \{y\}, \{x\}\rangle \cdot \iota_1\langle \texttt{True}, \{z\}, \{x\}\rangle
\end{array}$$

Example 3. The program in Fig. 8 consists of three threads, where the first two threads iterate in a `while`-loop, and the last thread assigns a variable to `p`, depending on the values of `i`, `m`, `j` and `n`. In the program, the variables `x` and `y` are shared between two out of three threads, while the other variables are local to a thread. To make the program terminating, we use *bounded* `while`-loops to specify the number of iterations of the loops that should be performed. Since there is no synchronization, the program has a high number of concurrent interleavings.

```
proc {
 while (i < m)[1] {
  x := x * i;
  i := i + 1
 }
}
```

```
proc {
 while (j < n)[1] {
  y := y * j;
  j := j + 1
 }
}
```

```
proc {
 if (i == m and j == n) {
  p := x * y
 } else {
  p := -1
 }
}
```

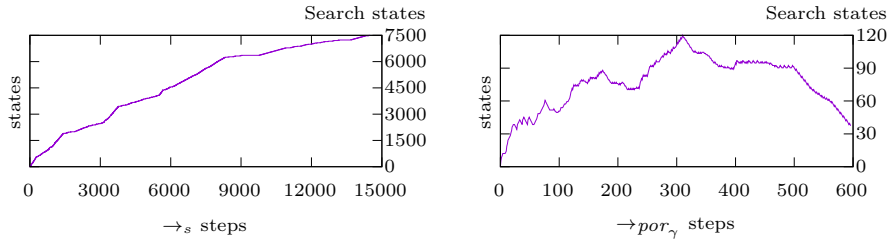
Fig. 8: A concurrent program

The search space for this program grows very quickly. If we consider two instances of the program, with bounds $1, 1$ and $2, 1$ for the two `while`-loops,

respectively, the following table shows the number of symbolic search *states* upon termination and the number of *steps* in terms of applications of the scheduling rule for executions with and without partial order reduction. In the table, the leftmost column shows the bounds of the two `while`-loops.

	$\rightarrow_s$		$\rightarrow_{por_\gamma}$	
Bounds	States	Steps	States	Steps
1,1	6744	14495	35	596
2,1	58944	132884	97	1562

Fig. 9 shows how many symbolic search states are visited during the search. As expected, when no partial order reduction is applied, the number of states grows monotonically. However, with partial order reduction, the number of states peaks at 110 states in the middle of the search, but many of these are later found to be equivalent, leading to a final number of 35 states.



(a) Search without POR.

(b) Search with POR.

Fig. 9: The number of symbolic search states for the program with bounds $1, 1$.

6 Language Extensions

In this section, we briefly describe some possible extensions to the programming language.

6.1 Aliasing

In this paper, we considered a multithreaded shared variable programming language that only includes assignments to simple variables. Assignments involving arrays or fields of objects give rise to aliasing. In [8], a formal model is described for the symbolic execution of such assignments in a sequential context. Extending this model to multithreaded programs further requires a definition of the equivalence relation between symbolic paths that takes aliasing into account, in order to apply partial order reduction. More specifically, the commutativity of two events will depend on the aliasing information encoded by the symbolic path preceding these two events.

6.2 Dynamically Spawned Threads

It is straightforward to extend the language and the symbolic transition system $\rightarrow_s$ with dynamic thread creation. A new thread can be introduced by adding a syntactic construct `new(s)` to Pr. Since threads in the symbolic states are represented by a thread pool, the dynamically created thread needs to both be inserted into the thread pool and maintain the uniqueness of thread identifiers. Let a predicate $\text{fresh}(\iota, \Sigma)$ express that a thread identifier ι does not occur among the identifiers of the threads already in the thread pool Σ. It should be noted that the thread identifies are only generated at run-time and cannot be manipulated by the program. As such they are different from the object identities in KeY. The introduction of dynamically created threads can be handled by extending the symbolic transition system $\rightarrow_s$ with the following additional rule:

$$\frac{\text{(NewThread)} \quad \text{fresh}(\iota', \Sigma)}{\begin{array}{c}(\!|\sigma, \phi, \{\iota(\texttt{new}(s'); s)\} \cup \Sigma|\!) \\ \xrightarrow{\iota}_s (\!|\sigma, \phi, \{\iota(s), \iota'(s')\} \cup \Sigma|\!)\end{array}}$$

The proofs of correctness and completeness are straightforward extensions of the proofs of Theorems 2 and 3.

6.3 Synchronization

The programming language discussed in this paper has no means of synchronization between threads. Synchronization can be achieved through simple mechanisms like locks or channels, or the more general mechanism of guarding statements with a condition on shared memory (e.g., `await` $(x < 5)$ in the syntax of ABS [17]). We can extend the language with guards by adding an event with the condition of the guard to the symbolic path whenever a guarded thread is scheduled. In the context of partial order reduction, equivalence between paths remains the same, as the conditions imposed by guards are no different than the conditions of conditional or `while`-statements.

6.4 Object Orientation

Above we already discussed the symbolic execution of assignments to fields of objects. In [8] a formal model of the symbolic execution of a basic sequential objet-oriented language is given which includes method calls. Of particular interest here is that the underlying symbolic representation of the heap abstracts from concrete object identifiers (or "locations"). In contrast, in the KeY system the symbolic execution of field assignments is defined with respect to an explicit heap variable. Such an explicit heap variable introduces a mismatch between the abstraction level of the programming language and its symbolic execution. On the other hand, in [18] we already formalized symbolic execution for abstract object creation in dynamic logic which, as in [8], does not involve an explicit

heap variable. As such this formalization provides a promising basis for further extensions to object-oriented multithreaded programs, using the partial order reduction techniques as described in this paper.

7 Related Work

Symbolic execution of concurrent programs is limited by the state space explosion due to all possible interleavings of concurrent events [19]. As such, the number of execution paths and of branching points where path conditions need to be evaluated, grows exponentially in the number of concurrent events within each thread. Many techniques have been proposed to mitigate state explosion [3], for example by using heuristics [20], abstractions [21], or by either merging [22, 23] or pruning states [9, 24, 25]. We studied the latter approach in this paper.

Similar to [24, 26, 27], we assume the execution of a program symbolically on a virtual machine that dynamically computes a dependency relation between states. Executions that have the same causal structure can be pruned, as they are equivalent with respect to finding errors in the given program. Our approach differs from prior works on combining partial order reduction techniques with symbolic execution. The main difference is that we provide a formal framework for studying the correctness and completeness of the symbolic and the reduced semantics with respect to that with concrete values. In this respect, we extend the work in [8] to concurrency.

Although it does not play a role in the programming language we considered in this paper, in general, the computation of the dependency relation may involve alias analysis. To avoid imprecision and extra computational burden, it could be interesting to investigate the combination of the current results with the syntax-based method proposed in [8] for checking aliases.

Our second contribution is a sound and relative complete dynamic logic for concurrent programs using partial order reduction. Several variations of dynamic logic exist for sequential programs, but only few calculi extend dynamic logic to concurrency. Most notably, concurrent dynamic trace logic [28] extends dynamic logic with symbolic execution rules for concurrent interleavings and dynamic thread creation based on the rely/guarantee methodology, the dynamic logic for concurrent Java [29] has been implemented in the KeY system [2], and the dynamic logic for Creol [30] supports compositional verification of an object-oriented modeling language for concurrent distributed applications [31]. The latter is very similar to our dynamic logic, because its semantics is based on traces, reformulating the work in [32] in the context of dynamic logic. The main difference with our approach is in the use of partial order reduction, so that verification is not only compositional but also effective. Other related dynamic logics for concurrency are Peleg's Concurrent Dynamic Logic [33], its extension with channels and shared variable communication [34], and the dynamic logic for communicating concurrent systems [35]. The former can only treat concurrent systems with no communication, while the latter can describe communications

and concurrency using a CCS-based language with Kleene-star for recursion. Neither of these logics, however, focus on partial order reduction.

8 Conclusion

The use of symbolic execution for multithreaded languages may lead to a state space explosion during model exploration. In this paper, we have studied how symbolic execution can be combined with partial order reduction, a technique for pruning symbolic states during model exploration by pruning states with equivalent symbolic paths. Our study is *foundational*: we develop a formal model of symbolic execution for multithreaded programs and provide a formal justification for the correctness of this model. We show how the formal model can be extended to incorporate partial order reduction, and provide a formal justification of correctness for the extended model. We then show how a proof system in dynamic logic for the considered multithreaded programming language corresponds to the formal model of symbolic execution such that correctness proofs for the latter carry over to prove soundness and relative completeness for the proof system. These proofs, which curiously establish the soundness and relative completeness of the proof system with partial order reduction by means of bisimulation, provide a formal justification for pruning nodes during proof construction, thereby simplifying the proof construction for multithreaded programs.

In this paper, our focus has been on the formal justification of the basic mechanisms of symbolic execution combined with partial order reduction for multithreaded programs. This work leaves many possible extensions unexplored. The current formulation of symbolic execution (and the corresponding proof system) will generate many unfeasible paths. One interesting extension of our work is the further pruning of search states by evaluating the consistency of symbolic paths, which opens for many different evaluation strategies (e.g., [3]). Another interesting extension is the treatment of aliases for the dependency relation underlying partial order reduction, which leads to imprecision and extra computational burden. It would also be interesting to investigate the formal justification of more aggressive pruning techniques, such as prefix-pruning over the equivalence relation of the partial order reduction.

Acknowledgement. We thank the reviewers for their valuable comments.

References

1. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7) (1976) 385–394
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M., eds.: Deductive Software Verification - The KeY Book - From Theory to Practice. Volume 10001 of Lecture Notes in Computer Science. Springer (2016)
3. Baldoni, R., Coppa, E., D'Elia, D.C., Demetrescu, C., Finocchi, I.: A survey of symbolic execution techniques. ACM Comput. Surv. **51**(3) (2018) 50:1–50:39

4. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM **56**(2) (2013) 82–90
5. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In Sarkar, V., Hall, M.W., eds.: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05), ACM (2005) 213–223
6. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: EXE: automatically generating inputs of death. In Juels, A., Wright, R.N., di Vimercati, S.D.C., eds.: Proc. 13th ACM Conference on Computer and Communications Security (CCS'06), ACM (2006) 322–335
7. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Draves, R., van Renesse, R., eds.: Proc. 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI'08), USENIX Association (2008) 209–224
8. de Boer, F.S., Bonsangue, M.M.: On the nature of symbolic execution. In ter Beek, M.H., McIver, A., Oliveira, J.N., eds.: Proc. 3rd World Congress on Formal Methods (FM 2019). Volume 11800 of Lecture Notes in Computer Science., Springer (2019) 64–80
9. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem. Volume 1032 of Lecture Notes in Computer Science. Springer (1996)
10. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. Foundations of Computing. MIT Press (October 2000)
11. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L., eds.: 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)
12. Clarke, E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press (2001)
13. Andrews, G.R.: Concurrent programming - principles and practice. Benjamin/Cummings (1991)
14. Owicki, S.S., Gries, D.: An axiomatic proof technique for parallel programs I. Acta Inf. **6** (1976) 319–340
15. Apt, K.R., de Boer, F.S., Olderog, E.: Verification of Sequential and Concurrent Programs. 3 edn. Texts in Computer Science. Springer (2009)
16. Meseguer, J., Rosu, G.: The rewriting logic semantics project: A progress report. Inf. Comput. **231** (2013) 38–69
17. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A core language for abstract behavioral specification. In Aichernig, B., de Boer, F.S., Bonsangue, M.M., eds.: Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010). Volume 6957 of Lecture Notes in Computer Science., Springer (2011) 142–164
18. de Gouw, S., de Boer, F.S., Ahrendt, W., Bubel, R.: Integrating deductive verification and symbolic execution for abstract object creation in dynamic logic. Software and Systems Modeling **15**(4) (2016) 1117–1140
19. Steele, G.L.: Making asynchronous parallelism safe for the world. In: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 1990), ACM (1989) 218–231
20. Li, Y., Su, Z., Wang, L., Li, X.: Steering symbolic execution to less traveled paths. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA 2013), ACM (2013) 19–32

21. Guo, S., Kusano, M., Wang, C., Yang, Z., Gupta, A.: Assertion guided symbolic execution of multithreaded programs. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015), ACM (2015) 854–865
22. Kuznetsov, V., Kinder, J., Bucur, S., Candea, G.: Efficient state merging in symbolic execution. In Vitek, J., Lin, H., Tip, F., eds.: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12), ACM (2012) 193–204
23. Scheurer, D., Hähnle, R., Bubel, R.: A general lattice model for merging symbolic execution branches. In Ogata, K., Lawford, M., Liu, S., eds.: Proceedings of the 18th International Conference on Formal Engineering Methods (ICFEM 2016). Volume 10009 of LNCS., Springer (2016) 57–73
24. Boonstoppel, P., Cadar, C., Engler, D.: RWset: Attacking path explosion in constraint-based test generation. In Ramakrishnan, C. R.and Rehof, J., ed.: Tools and Algorithms for the Construction and Analysis of Systems, Springer (2008) 351–366
25. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005), ACM (2005) 110–121
26. Khurshid, S., Pasareanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003), Springer-Verlag (2003) 553–568
27. Visser, W., Pasareanu, C.S., Khurshid, S.: Test input generation with Java PathFinder. SIGSOFT Softw. Eng. Notes **29**(4) (2004) 97–107
28. Bruns, D.: Deductive verification of concurrent programs. Technical Report 3, Karlsruher Institut für Technologie (KIT) (2015)
29. Beckert, B., Klebanov, V.: A dynamic logic for deductive verification of concurrent programs. In: Proc. Fifth IEEE Intl. Conf. on Software Engineering and Formal Methods (SEFM 2007), IEEE Computer Society (2007) 141–150
30. Ahrendt, W., Dylla, M.: A system for compositional verification of asynchronous objects. Sci. Comput. Program. **77**(12) (2012) 1289–1309
31. Johnsen, E.B., Owe, O., Yu, I.C.: Creol: A type-safe object-oriented model for distributed concurrent systems. Theoretical Computer Science **365**(1) (2006) 23–66
32. Dovland, J., Johnsen, E.B., Owe, O.: Observable behavior of dynamic systems: Component reasoning for concurrent objects. Electr. Notes Theor. Comput. Sci. **203**(3) (2008) 19–34
33. Peleg, D.: Concurrent dynamic logic. In: Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing (STOC 1985), ACM (1985) 232–239
34. Peleg, D.: Communication in concurrent dynamic logic. J. Comput. Syst. Sci. **35**(1) (1987) 23–58
35. Benevides, M.R.F., Schechter, L.M.: Propositional dynamic logics for communicating concurrent programs with CCS's parallel operator. J. Log. Comput. **24**(4) (2014) 919–951