

Inseguendo Fagiani Selvatici: Partial Order Reduction for Guarded Command Languages

Frank S. de Boer


CWI Amsterdam, The Netherlands
f.s.de.boer@cwi.nl

Einar Broch Johnsen 

Department of Informatics, University of Oslo, Norway
einarj@ifi.uio.no

Rudolf Schlatte

Department of Informatics, University of Oslo, Norway
rudi@ifi.uio.no

Silvia Lizeth Tapia Tarifa 

Department of Informatics, University of Oslo, Norway
sltarifa@ifi.uio.no

Lars Tveito

Department of Informatics, University of Oslo, Norway
larstvei@ifi.uio.no

Abstract

This paper presents a method for testing whether objects in actor languages and active object languages exhibit locally deterministic behavior. We investigate such a method for a class of guarded command programs, abstracting from object-oriented features like method calls but focusing on cooperative scheduling of dynamically spawned processes executing in parallel. The proposed method can answer questions such as whether all permutations of an execution trace are equivalent, by generating candidate traces for testing which may lead to different final states. To prune the set of candidate traces, we employ partial order reduction. To further reduce the set, we introduce an analysis technique to decide whether a generated trace is schedulable. Schedulability cannot be decided for guarded commands using standard dependence and interference relations because guard enabledness is non-monotonic. To solve this problem, we use concolic execution to produce linearized symbolic traces of the executed program, which allows a weakest precondition computation to decide on the satisfiability of guards.

2012 ACM Subject Classification Software and its engineering → Automated static analysis; Software and its engineering → Software testing and debugging; Software and its engineering → Semantics; Theory of computation → Semantics and reasoning

Keywords and phrases Testing, Symbolic Traces, Guarded Commands, Partial Order Reduction

Digital Object Identifier 10.4230/OASICS.Gabbrielli.2020.10

1 Introduction

Let us open this paper with the allegory of the pheasant-chasing wine-maker:

A vineyard is a place where wild pheasants are gobbling up the grapes and where wine-makers chase these pheasants off the land. During this Sisyphean undertaking, a theoretically inclined wine-maker may wonder: “will the order in which I chase the pheasants affect the yield at season’s end?” Overwhelmed by the existential dimensions of this question, the wine-maker could but observe the unfolding of the feast.



© Frank S. de Boer, Einar Broch Johnsen, Rudolf Schlatte, S. Lizeth Tapia Tarifa, and Lars Tveito; licensed under Creative Commons License CC-BY

Recent Developments in the Design and Implementation of Programming Languages.

Editors: Frank S. de Boer and Jacopo Mauro; Article No. 10; pp. 10:1–10:18

OpenAccess Series in Informatics



OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Leaving aside its existential dimensions, the astute reader may observe that the problem of the pheasant-chasing wine-maker bears a remarkable similarity to the problem of testing whether dynamically spawned processes operating on a shared state space exhibit deterministic behavior in the sense that the final state is independent of the local scheduling decisions. We hope that we, by studying a particular instance of the latter problem, can also shed some light on the former.

Different forms of dynamically spawned processes have been studied extensively by Gabbrielli [12–14]. The problem that we address in this paper is to test whether in an *imperative* context cooperatively scheduled tasks executing on a *shared state space* exhibit *deterministic* behavior. This problem lies at the heart of the development of a testing theory for a single active object, or a single actor in an actor system. For a general overview of different programming languages and paradigms, we refer to Gabbrielli’s course notes [28].

Actor systems [2] are inherently prone to social distancing, based on a strong sense of isolation and asynchronous communication: by design, one actor cannot directly affect the local state of another actor, it only sends messages. Active objects [10] extend these attractive features of actors to an object-oriented setting with asynchronous method calls and futures [22, 26]. As a consequence, both actor and active object systems are *almost* confluent: if there are no communication races and local scheduling is deterministic, asynchronously communicating objects have been shown to have deterministic behavior [17].

Active object languages such as ABS [33] and Encore [11] allow methods to be cooperatively scheduled: a task executing on an object may choose to suspend itself and allow other tasks to be scheduled, such that the original task can only be rescheduled once an associated Boolean condition holds. This extension makes the behavior highly non-deterministic because suspended tasks that are enabled can be arbitrarily selected by the scheduler when an object is idle. The authors have previously shown that even for cooperatively scheduled active objects, it is sufficient to control the local behavior of each actor to ensure global confluence [9, 42] and developed an axiomatic semantics of trace reachability for active objects [23]. However, neither line of work addresses the problem of testing determinacy for active objects with cooperatively scheduled tasks.

In this paper, we study the problem of testing whether an object with cooperatively scheduled tasks locally exhibits deterministic behavior in the sense that the final state of the object is independent of the local scheduling of tasks. The cooperatively scheduled tasks of a single object can be abstracted in terms of a guarded command language over shared state. The paper develops a behavioral theory of guarded commands in a slight variation of Dijkstra’s guarded command language GCL [24]. The problem of deterministic behavior can be formulated as determining whether all feasible schedulings of tasks (also called processes in the sequel) will produce the same final state, or as testing whether, given a trace of a guarded command program, are there permutations of the trace that are executable but not observationally equivalent to the original trace? We tackle this problem by means of concolic execution [15, 16, 31], such that the concrete run produces a linearized, symbolic trace of the executed program. We then combine techniques for weakest-precondition calculation [7, 25] and for partial order reduction [19, 29] to compute, for a given symbolic trace, all executable traces which only differ in the interleaving of the individual local computations of the tasks (threads) of the given trace and which may result in a different final state.

2 Partial Order Reduction

Partial order reduction (POR) is a technique to reduce the size of the search space when exploring the different executions of a parallel program by exploiting the commutativity of concurrently executed *independent* transitions [19, 29]. This commutativity relation between transitions is lifted to an equivalence relation \sim on traces over these transitions. Given a trace θ reflecting an interleaved execution of a number of parallel processes, we denote by $[\theta]$ the set of traces equivalent to θ according to the equivalence relation \sim ; the traces in this set agree on the sequential order of transitions for the individual processes, but the processes may be interleaved in different ways. Thus, all equivalent traces have the same length and contain the same labeled transition steps. Let $s \xrightarrow{\theta} s'$ denote that a state s' can be reached from a state s by a sequence of labeled transition steps, where the trace θ is the corresponding sequence of labels representing scheduling events for the different transition steps. We use the following notation for traces: ε denotes the empty trace and $\theta \cdot \tau$ the composition of a trace θ and an event τ . With a slight abuse of notation, we will write $\theta_1 \cdot \theta_2$ for the composition of traces θ_1 and θ_2 and $\tau \cdot \theta$ for the trace which starts with event τ and continues as trace θ .

The pruning of the search based on traces during model exploration can be justified when the traces are sufficiently expressive to make sure that equivalent traces lead to equal states; i.e., the following must be a theorem [29]:

► **Theorem 1.** *If $s_0 \xrightarrow{\theta_1} s_1$, $s_0 \xrightarrow{\theta_2} s_2$ and $\theta_2 \in [\theta_1]$, then $s_1 = s_2$.*

Observe that, given a trace θ , the elements of $[\theta]$ can be enumerated by successively permuting adjacent commuting (i.e., independent) transition steps. 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 [6]; e.g., two transitions are independent if their corresponding statements do not affect each others' program variables.

► **Example 2** (Independent processes). Consider a program

$$x \mapsto 1, y \mapsto 2, \{ x := x+1 \parallel y := 3 \}$$

with two parallel statements $x := x+1$ and $y := 3$ and a shared state that is initialized with program variables x with value 1 and $y = 2$. Assume that the two statements are executed by processes ι_1 and ι_2 , respectively, and, for simplicity, that the execution of each assignment is atomic. The final result of this program is a state in which x has value 2 and y has value 3. However, there are two traces θ_1 and θ_2 of this program, reflecting that either of the two processes ι_1 and ι_2 can be scheduled first without affecting the outcome of the program. Thus, θ_1 and θ_2 belong to the same equivalence class.

POR can be used to explore the different equivalence classes of executions, without exploring every execution path of each equivalence class. If we can decide whether two traces are in the same equivalence class, we can stop the analysis of a candidate execution path if we know that its trace is equivalent to the trace of an execution that has already been explored.

► **Example 3** (Interfering processes). Consider a program

$$x \mapsto 1, \{ x := x+1 \parallel x := 3 \}$$

which describes two processes $x := x+1$ and $x := 3$ executing interleaved on a shared state where program variable x has value 1. This program may have two outcomes: in the final state, x will have as value either 3 or 4, depending on the scheduling of the processes. Let us assume that the two statements are executed by processes ι_1 and ι_2 , respectively. Then these two executions can be represented by execution traces θ_1 and θ_2 in which the scheduling of transition steps (which we can represent as events) for ι_1 and ι_2 occur in different orders. Since the outcome of executing the program depends on the selected trace, θ_1 and θ_2 do not belong to the same equivalence class.

The interference of two scheduling events can be approximated by syntactic criteria. A common approach is to decorate the scheduling events with the sets of read and written variables in the executed atomic blocks, such that a standard notion of interference [6] can be applied (i.e., write-variables in one process interfere with both read- and write-variables in the other process). In this case, scheduling events τ will have the format $\iota\langle W, R \rangle$, where ι is a process identifier and W and R are the write- and read-sets of the underlying transition. Thus, we get the traces $\iota_1\langle\{x\}, \{x\}\rangle \cdot \iota_2\langle\{x\}, \emptyset\rangle$ and $\iota_2\langle\{x\}, \emptyset\rangle \cdot \iota_1\langle\{x\}, \{x\}\rangle$ for the executions of Example 3. With events on this format, we can syntactically approximate non-interference by comparing write- and read-sets:

$$\iota_i\langle W_i, R_i \rangle \sim \iota_j\langle W_j, R_j \rangle \iff W_i \cap (R_j \cup W_j) = \emptyset \wedge (W_i \cup R_i) \cap W_j = \emptyset.$$

By Theorem 1, two traces $\theta_1 \cdot \tau_1 \cdot \tau_2 \cdot \theta_2$ and $\theta_1 \cdot \tau_2 \cdot \tau_1 \cdot \theta_2$ are equivalent and lead to the same final state s_1 from a given initial state s_0 if the events τ_1 and τ_2 are non-interfering, as captured by $\tau_1 \sim \tau_2$. In this case, there is no need to execute both traces. In contrast, if $\tau_1 \not\sim \tau_2$, the two traces may be in different equivalence classes and can lead to different final states from s_0 . However, we cannot in general know that the events in $\tau_1 \cdot \theta_2$ can be executed after $\theta_1 \cdot \tau_2$. For example, with dynamically spawned processes, the execution associated with τ_1 may create the process scheduled by τ_2 . This dependency between events can be captured by a so-called *must happen before* relation, which is a transitive relation over the events of the traces: if two events τ_1 and τ_2 associated with different processes are in a causal ordering, they cannot be permuted even if $\tau_1 \sim \tau_2$. Thus, there is a clear resemblance between equivalence classes in our setting and Mazurkiewicz traces [35].

POR can be used to systematically generate traces which correspond to all possible behaviors of a program up to trace equivalence without executing all the traces of the program, for example for the purpose of systematic testing [4]. The basic idea is to ensure that all equivalence classes are visited by at least one execution. Given a trace θ that corresponds to some execution, and $\theta_1 \cdot \tau_1 \cdot \tau_2 \cdot \theta_2 \in [\theta]$ (where $[\theta]$ is an equivalence class) such that $\tau_1 \not\sim \tau_2$, we know that the any trace that extends $\theta_1 \cdot \tau_2$ is a candidate trace for a different equivalence class than $[\theta]$. An algorithm can successively run executions which extend a given trace prefix (e.g., $\theta_1 \cdot \tau_2$), such that all equivalence classes will eventually be visited. The best known such algorithm is perhaps DPOR [27], which is usually implemented by directly manipulating the data structures and the scheduler of the runtime system of a targeted language.

3 GCL: A Language with Guarded Commands

This section presents a guarded command language (hereafter GCL), which is a slight simplifying variation on Dijkstra's original language [24]. The syntax of GCL is given in Figure 1. A program $Prog$ consists of a state σ and a guarded statement g . The state σ binds program variables x to values v . Guarded statements g are **skip**, sequential composition

$$\begin{aligned}
Prog &::= \sigma, g \\
\sigma \in State &::= \epsilon \mid \sigma[x \mapsto v] \\
g \in GrdStm &::= \mathbf{skip} \mid e \triangleright s \mid s \triangleleft e \triangleright s \mid g; g \\
s \in Stm &::= x := e \mid \mathbf{spawn}(g) \\
e \in Exp &::= \mathbf{True} \mid \mathbf{False} \mid x \mid v \mid op(e, \dots, e)
\end{aligned}$$

■ **Figure 1** The syntax of *GCL*.

$g; g$ and the two statements $e \triangleright s$ and $s_1 \triangleleft e \triangleright s_2$, where e is an expression and s, s_1, s_2 are simple statements. Simple statements s are assignments $x := e$ and $\mathbf{spawn}(g)$. The guarded statement $e \triangleright s$ allows s to be executed when the guard e holds. The guarded statement $s_1 \triangleleft e \triangleright s_2$ will execute s_1 when e holds, otherwise s_2 . The guarded statement \mathbf{skip} is used to denote both internal actions and termination, i.e., we identify g with $g; \mathbf{skip}$ and define a solitary $\iota(\mathbf{skip})$ (explained below) to represent a terminated process. Expressions e include basic propositions \mathbf{True} and \mathbf{False} , program variables x , values v (such as Boolean values \mathbf{true} and \mathbf{false} , and numbers), and operations over expressions (such as addition of numbers and logical operators over Booleans). Expressions are assumed to be well-typed. Their syntax is standard and not further detailed in Figure 1. The GCL language is kept intentionally simple, but see Section 6 for some straightforward extensions such as loops, nested guarded statements and procedure calls.

Runtime syntax

The execution of GCL programs is organized around a set of processes in the form of guarded commands. Processes are executed in an interleaved way. Any enabled process may be selected to execute at any scheduling point, which makes GCL highly non-deterministic. The runtime syntax extends Fig. 1 as follows:

$$\begin{aligned}
rs \in RuntimeState &::= \sigma, P \\
P \in ProcessSet &::= \emptyset \mid \{\iota(g)\} \mid P \cup P
\end{aligned}$$

A runtime configuration rs is a tuple which consists of a state σ and a set P of processes. A state σ assigns values to the program's shared variables. A state update $\sigma[x \mapsto v]$ denotes the state resulting from assigning the value v to the variable x . By $\sigma(e)$ we denote the value resulting from the evaluation of the expression e in state σ . Note that evaluating expressions is free of side effects. Processes are written $\iota(g)$ and consist of a process identifier ι and a guarded statement g (which can be a compound statement $g; g$). All processes in a runtime configuration are required to have unique process identifiers. The initial process, which is not created by a \mathbf{spawn} statement, is assigned the process identifier ι_0 .

The operational semantics of GCL is defined as a transition system $rs \rightarrow rs'$ between runtime configurations rs and rs' , shown in Fig. 2. Rule ASSIGN updates the global state with the effect of an assignment under the assumption that the guard is true. Rule SPAWN creates a new process. The process identifier ι' of the spawned process is non-deterministically chosen (uniqueness is guaranteed by the INTERLEAVING rule and the above requirement that all its processes in a runtime configuration have unique process identifiers). Rule SKIP can always reduce since its implicit guard is taken to be \mathbf{True} . Each of the CHOICE rules schedules the enabled guarded statement (note that the premises of both rules result from the execution of the enabled statement). Rule INTERLEAVING nondeterministically chooses a process to execute, which will trigger the execution of one of the other rules, depending on the guarded statement g .

$$\begin{array}{c}
\text{(ASSIGN)} \\
\frac{\sigma(e) = \text{True} \quad \sigma' = \sigma[x \mapsto \sigma(e')]}{\sigma, \{\iota(e \triangleright x := e'; g)\} \rightarrow \sigma', \{\iota(g)\}} \\
\\
\text{(SPAWN)} \\
\frac{\sigma(e) = \text{True}}{\sigma, \{\iota(e \triangleright \text{spawn}(g'); g)\} \rightarrow \sigma, \{\iota(g), \iota'(g')\}} \\
\\
\text{(SKIP)} \\
\sigma, \{\iota(\text{skip}; g)\} \rightarrow \sigma, \{\iota(g)\} \\
\\
\text{(CHOICE1)} \\
\frac{\sigma, \{\iota(e \triangleright s_1; g)\} \rightarrow \sigma', P}{\sigma, \{\iota(s_1 \triangleleft e \triangleright s_2; g)\} \rightarrow \sigma', P} \\
\\
\text{(CHOICE2)} \\
\frac{\sigma, \{\iota(\neg e \triangleright s_2; g)\} \rightarrow \sigma', P}{\sigma, \{\iota(s_1 \triangleleft e \triangleright s_2; g)\} \rightarrow \sigma', P} \\
\\
\text{(INTERLEAVING)} \\
\frac{\sigma, \{\iota(g)\} \rightarrow \sigma', P'}{\sigma, P \cup \{\iota(g)\} \rightarrow \sigma', P \cup P'}
\end{array}$$

■ **Figure 2** Operational semantics of GCL.

The *initial runtime configuration* of a program σ, g is given by $\sigma, \{\iota_0(g)\}$ with the initial process identifier ι_0 . A successful execution of a program from an initial runtime configuration $\sigma, \{\iota_0(g)\}$ is a sequence of transitions which ends in a *terminal* configuration where all processes are of the form $\iota(\text{skip})$. An execution *deadlocks* if it reaches a non-terminal configuration in which no rule is applicable. This can happen when the guards of all the initial statements evaluate to **False**. We denote by \rightarrow^* the transitive closure of the transition relation \rightarrow and by $\sigma, g \rightarrow^* \sigma'$ the existence of a successful execution of the program σ, g with initial state σ and final state σ' .

4 A Concolic Semantics for GCL

A concolic semantics for GCL can be defined by lifting the non-deterministic operational semantics of Fig. 2 to a labeled transition system in which the labeled transitions of each rule of the operational semantics capture the symbolic execution step corresponding to the concrete transition. The labeled transition relation \xrightarrow{l} is given in Fig. 3; apart from the labeling the rules are the same as their non-labeled versions in Fig. 2.

► **Definition 4** (Labels). *A label l takes one of the following forms:*

$$l ::= \tau \mid \iota(e \triangleright x := e') \mid \iota(e \triangleright \text{spawn}(\iota'))$$

Here, τ denotes the empty label and, in the other labels, ι corresponds to the identifier of the process that was executed.

The guarded statements in labels are non-branching (i.e., a guarded assignment or spawn). However, in case of the execution of a spawn instruction we record in the label the identifier of the new process (encoded by $\text{spawn}(\iota')$, where ι' denotes the new process identifier).

The *trace* θ generated by an execution in this transition system is the sequence of (non-empty) labels of the corresponding transition steps. The trace records a *symbolic linearization* of the executed program; i.e., the trace ignores the branching points in the control flow of the source program. Thus, there may be many traces which correspond to the different executions of a GCL program.

► **Example 5** (Traces). Consider a GCL program with an initial process ι_0 that spawns two processes ι_1 and ι_2 , where ι_1 doubles the value of a counter x if a guard **flag** is true, and ι_2 increments the value of x by one and negates the value of **flag** four times, then increments

$$\begin{array}{c}
\text{(ASSIGN)} \\
\frac{\sigma(e) = \text{True} \quad \sigma' = \sigma[x \mapsto \sigma(e')]}{\sigma, \{\iota(e \triangleright x := e'; g)\} \xrightarrow{\iota(e \triangleright x := e')} \sigma', \{\iota(g)\}}
\end{array}
\qquad
\begin{array}{c}
\text{(CHOICE1)} \\
\frac{\sigma, \{\iota(e \triangleright s_1; g)\} \xrightarrow{l} \sigma', P}{\sigma, \{\iota(s_1 \triangleleft e \triangleright s_2; g)\} \xrightarrow{l} \sigma', P}
\end{array}$$

$$\begin{array}{c}
\text{(SPAWN)} \\
\frac{\sigma(e) = \text{True}}{\sigma, \{\iota(e \triangleright \text{spawn}(g'); g)\} \xrightarrow{\iota(e \triangleright \text{spawn}(g'))} \sigma, \{\iota(g), \iota'(g')\}}
\end{array}
\qquad
\begin{array}{c}
\text{(CHOICE2)} \\
\frac{\sigma, \{\iota(\neg e \triangleright s_2; g)\} \xrightarrow{l} \sigma', P}{\sigma, \{\iota(s_1 \triangleleft e \triangleright s_2; g)\} \xrightarrow{l} \sigma', P}
\end{array}$$

$$\begin{array}{c}
\text{(SKIP)} \\
\sigma, \{\iota(\text{skip}; g)\} \xrightarrow{\tau} \sigma, \{\iota(g)\}
\end{array}
\qquad
\begin{array}{c}
\text{(INTERLEAVING)} \\
\frac{\sigma, \{\iota(g)\} \xrightarrow{l} \sigma', P'}{\sigma, P \cup \{\iota(g)\} \xrightarrow{l} \sigma', P \cup P'}
\end{array}$$

■ **Figure 3** Concolic operational semantics for GCL.

an unrelated variable y . Initially, we let the counters x and y have value 0 and flag has value true . In the surface syntax of GCL, the program looks like this:

```

x ↦ 0, y ↦ 0, flag ↦ true,
True ▷ spawn(x == 0 ∨ flag ▷ x := 2 * x);           // t1
True ▷ spawn(x := x + 2 < flag ▷ x := x - 1; True ▷ flag := ¬flag; // t2
           x := x + 2 < flag ▷ x := x - 1; True ▷ flag := ¬flag;
           x := x + 2 < flag ▷ x := x - 1; True ▷ flag := ¬flag;
           x := x + 2 < flag ▷ x := x - 1; True ▷ flag := ¬flag;
           True ▷ y := y + 1)

```

A possible execution of this program, resulting in a state $x \mapsto 4$, $y \mapsto 1$, $\text{flag} \mapsto \text{true}$, schedules t_2 until it completes before scheduling t_1 . The trace θ_0 , of the concolic transitions corresponding to this execution is

$$\begin{aligned}
\theta_0 : \quad & \iota_0(\text{True} \triangleright \text{spawn}(t_1)) \cdot \iota_0(\text{True} \triangleright \text{spawn}(t_2)) \\
& \cdot \iota_2(\text{flag} \triangleright x := x + 2) \cdot \iota_2(\text{True} \triangleright \text{flag} := \neg \text{flag}) \\
& \cdot \iota_2(\neg \text{flag} \triangleright x := x - 1) \cdot \iota_2(\text{True} \triangleright \text{flag} := \neg \text{flag}) \\
& \cdot \iota_2(\text{flag} \triangleright x := x + 2) \cdot \iota_2(\text{True} \triangleright \text{flag} := \neg \text{flag}) \\
& \cdot \iota_2(\neg \text{flag} \triangleright x := x - 1) \cdot \iota_2(\text{True} \triangleright \text{flag} := \neg \text{flag}) \\
& \cdot \iota_2(\text{True} \triangleright y := y + 1) \cdot \iota_1(x == 0 \vee \text{flag} \triangleright x := 2 * x)
\end{aligned}$$

Note that well-formed permutations in traces θ (e.g., permutations respecting program order in the different processes) will generate different linearizations with possible different final states due to the non-deterministic nature of GCL. Such possible well-formed permutations will be explored in the next section.

Let $e[e'/x]$ denote the substitution operation which replaces all occurrences of x in e by e' (it binds stronger than any other logical operation/connective). The path condition of a symbolic trace θ expresses all the guards of θ in terms of the initial state. We define path conditions symbolically as follows.

► **Definition 6** (Path condition). *Let θ be a trace over labels l . The path condition $\text{path}(\theta)$ is defined inductively:*

$$\begin{aligned} \text{path}(\epsilon) &= \text{True} \\ \text{path}(\tau \cdot \theta) &= \text{path}(\theta) \\ \text{path}(\iota(e \triangleright x := e') \cdot \theta) &= e \wedge \text{path}(\theta)[e'/x] \\ \text{path}(\iota(e \triangleright \text{spawn}(l')) \cdot \theta) &= e \wedge \text{path}(\theta) \end{aligned}$$

► **Example 7** (Path conditions). We compute the path conditions for the trace in Example 5.

$$\begin{aligned} \text{path}(\theta_0) &= \text{path}(\iota_0(\text{True} \triangleright \text{spawn}(\iota_1)) \cdot \iota_0(\text{True} \triangleright \text{spawn}(\iota_2)) \\ &\quad \cdot \iota_2(\text{flag} \triangleright \text{x}:=\text{x}+2) \cdot \iota_2(\text{True} \triangleright \text{flag}:=\neg\text{flag}) \\ &\quad \cdot \iota_2(\neg\text{flag} \triangleright \text{x}:=\text{x}-1) \cdot \iota_2(\text{True} \triangleright \text{flag}:=\neg\text{flag}) \\ &\quad \cdot \iota_2(\text{flag} \triangleright \text{x}:=\text{x}+2) \cdot \iota_2(\text{True} \triangleright \text{flag}:=\neg\text{flag}) \\ &\quad \cdot \iota_2(\neg\text{flag} \triangleright \text{x}:=\text{x}-1) \cdot \iota_2(\text{True} \triangleright \text{flag}:=\neg\text{flag}) \\ &\quad \cdot \iota_2(\text{True} \triangleright \text{y}:=\text{y}+1) \cdot \iota_1(\text{x}==0 \vee \text{flag} \triangleright \text{x}:=2*\text{x})) \\ &= \text{flag} == \text{true} \end{aligned}$$

We can see how the computation produces the weakest precondition for the guards to hold; thus, any state in which the initial state of `flag` has value `true` allows the execution of θ_0 .

For a symbolic trace θ , let $g(\theta)$ denote the corresponding guarded statement obtained simply by dropping the empty labels τ , removing the process identifiers ι (such that $e \triangleright \text{spawn}(\iota)$ becomes $e \triangleright \text{spawn}(\text{skip})$), and connecting the resulting sequence of guard statements via sequential composition. We have the following basic property for path conditions.

► **Theorem 8** (Formal justification of path conditions). *For any symbolic trace θ and initial state σ , if $\sigma(\text{path}(\theta)) = \text{True}$ then there exists a state σ' such that $\sigma, g(\theta) \rightarrow^* \sigma'$.*

Proof. The proof proceeds by a straightforward induction on the length of θ (assuming that for the base case $g(\epsilon) = \text{skip}$). ◀

5 Partial Order Reduction for GCL

This section describes an algorithm which, from a given run, constructs all scheduling policies which respect the local flow of control of the individual processes. In order to reduce the search space we apply a partial order reduction based on a non-interference relation between the labels of the concolic operational semantics.

5.1 Symbolic Traces and Equivalence

We first define equivalence for the symbolic traces of the concolic semantics of GCL. Let $\text{vars}(e)$ denote the program variables in an expression e . For a label l , we define the write- and read-sets, written as $W(l)$ and $R(l)$ respectively, as follows:

$$\begin{aligned} W(\iota(e_1 \triangleright x := e_2)) &= \{x\} & R(\iota(e_1 \triangleright x := e_2)) &= \text{vars}(e_1) \cup \text{vars}(e_2) \\ W(\iota(e \triangleright \text{spawn}(l'))) &= \emptyset & R(\iota(e \triangleright \text{spawn}(l'))) &= \text{vars}(e) \end{aligned}$$

Building on the general explanation of non-interference in Section 2, we can now define the non-interference relation \sim between labels for GCL as follows:

► **Definition 9** (Non-interference). For any two labels $l_1 = \iota_1(e_1 \triangleright s_1)$ and $l_2 = \iota_2(e_2 \triangleright s_2)$ we denote by $l_1 \sim l_2$ that

$$\begin{aligned} W(l_1) \cap (R(l_2) \cup W(l_2)) &= \emptyset \wedge (W(l_1) \cup R(l_1)) \cap W(l_2) = \emptyset \\ \wedge \iota_1 &\neq \iota_2 \\ \wedge s_1 &\neq \text{spawn}(\iota_2) \wedge s_2 \neq \text{spawn}(\iota_1) \end{aligned}$$

This definition of non-interference captures both the non-interference of independent transitions and the must-happen-before relation for the semantics of GCL: two events of the same process must happen in program order, a process cannot be scheduled before it is created, and events with overlapping write- and read-sets cannot be reordered.

Recall from Section 2 that an equivalence relation between events can be extended to an equivalence relation on traces over those events. We extend the non-interference relation of Definition 9 to the smallest equivalence relation between symbolic traces such that $\theta' \cdot l_1 \cdot l_2 \cdot \theta'' \sim \theta' \cdot l_2 \cdot l_1 \cdot \theta''$, if $l_1 \sim l_2$.

► **Example 10** (Trace permutations and equivalence.). Consider the following traces, which are permutations of trace θ_0 from Example 5. In the traces, the changing position of the label $\iota_1(x==0 \vee \text{flag} \triangleright x:=2*x)$ is highlighted.

$$\begin{aligned} \theta_1 : & \quad \iota_0(\text{True} \triangleright \text{spawn}(\iota_1)) \cdot \iota_0(\text{True} \triangleright \text{spawn}(\iota_2)) \\ & \quad \cdot \iota_2(\text{flag} \triangleright x:=x+2) \cdot \iota_2(\text{True} \triangleright \text{flag}:=\neg \text{flag}) \\ & \quad \cdot \iota_2(\neg \text{flag} \triangleright x:=x-1) \cdot \iota_2(\text{True} \triangleright \text{flag}:=\neg \text{flag}) \\ & \quad \cdot \iota_2(\text{flag} \triangleright x:=x+2) \cdot \iota_2(\text{True} \triangleright \text{flag}:=\neg \text{flag}) \\ & \quad \cdot \iota_2(\neg \text{flag} \triangleright x:=x-1) \cdot \iota_2(\text{True} \triangleright \text{flag}:=\neg \text{flag}) \\ & \quad \cdot \iota_1(x==0 \vee \text{flag} \triangleright x:=2*x) \cdot \iota_2(\text{True} \triangleright y:=y+1) \\ \\ \theta_2 : & \quad \iota_0(\text{True} \triangleright \text{spawn}(\iota_1)) \cdot \iota_0(\text{True} \triangleright \text{spawn}(\iota_2)) \\ & \quad \cdot \iota_2(\text{flag} \triangleright x:=x+2) \cdot \iota_2(\text{True} \triangleright \text{flag}:=\neg \text{flag}) \\ & \quad \cdot \iota_2(\neg \text{flag} \triangleright x:=x-1) \cdot \iota_2(\text{True} \triangleright \text{flag}:=\neg \text{flag}) \\ & \quad \cdot \iota_2(\text{flag} \triangleright x:=x+2) \cdot \iota_2(\text{True} \triangleright \text{flag}:=\neg \text{flag}) \\ & \quad \cdot \iota_2(\neg \text{flag} \triangleright x:=x-1) \cdot \iota_1(x==0 \vee \text{flag} \triangleright x:=2*x) \\ & \quad \cdot \iota_2(\text{True} \triangleright \text{flag}:=\neg \text{flag}) \cdot \iota_2(\text{True} \triangleright y:=y+1) \\ \\ \theta_3 : & \quad \iota_0(\text{True} \triangleright \text{spawn}(\iota_1)) \cdot \iota_0(\text{True} \triangleright \text{spawn}(\iota_2)) \\ & \quad \cdot \iota_2(\text{flag} \triangleright x:=x+2) \cdot \iota_2(\text{True} \triangleright \text{flag}:=\neg \text{flag}) \\ & \quad \cdot \iota_2(\neg \text{flag} \triangleright x:=x-1) \cdot \iota_2(\text{True} \triangleright \text{flag}:=\neg \text{flag}) \\ & \quad \cdot \iota_2(\text{flag} \triangleright x:=x+2) \cdot \iota_1(x==0 \vee \text{flag} \triangleright x:=2*x) \\ & \quad \cdot \iota_2(\text{True} \triangleright \text{flag}:=\neg \text{flag}) \cdot \iota_2(\neg \text{flag} \triangleright x:=x-1) \\ & \quad \cdot \iota_2(\text{True} \triangleright \text{flag}:=\neg \text{flag}) \cdot \iota_2(\text{True} \triangleright y:=y+1) \end{aligned}$$

Trace θ_1 is in the same equivalence class as θ_0 , denoted by $\theta_1 \in [\theta_0]$, since $\iota_1(x==0 \vee \text{flag} \triangleright x:=2*x) \sim \iota_2(\text{True} \triangleright y:=y+1)$. However, $\theta_2 \notin [\theta_0]$ since $\iota_1(x==0 \vee \text{flag} \triangleright x:=2*x) \not\sim \iota_2(\text{True} \triangleright \text{flag}:=\neg \text{flag})$, and similarly $\theta_3 \notin [\theta_0]$, $\theta_3 \notin [\theta_2]$, etc.

► **Example 11** (Path conditions continued). We compute path conditions for the traces from Example 10. By computing $path(\theta_2)$ we get the constraint

$$\begin{aligned}
path(\theta_2) &= path(\iota_0(\text{True} \triangleright \text{spawn}(\iota_1)) \cdot \iota_0(\text{True} \triangleright \text{spawn}(\iota_2))) \\
&\quad \cdot \iota_2(\text{flag} \triangleright x := x + 2) \cdot \iota_2(\text{True} \triangleright \text{flag} := \neg \text{flag}) \\
&\quad \cdot \iota_2(\neg \text{flag} \triangleright x := x - 1) \cdot \iota_2(\text{True} \triangleright \text{flag} := \neg \text{flag}) \\
&\quad \cdot \iota_2(\text{flag} \triangleright x := x + 2) \cdot \iota_2(\text{True} \triangleright \text{flag} := \neg \text{flag}) \\
&\quad \cdot \iota_2(\neg \text{flag} \triangleright x := x - 1) \cdot \iota_1(x == 0 \vee \text{flag} \triangleright x := 2 * x) \\
&\quad \cdot \iota_2(\text{True} \triangleright \text{flag} := \neg \text{flag}) \cdot \iota_2(\text{True} \triangleright y := y + 1) \\
&= \text{flag} == \text{true} \wedge x == -2
\end{aligned}$$

Thus, θ_2 is not executable from the initial program state of Example 5, but it would be executable from states satisfying this constraint (i.e., for initial states in which the value of `flag` is true and the value of `x` is -2). By computing $path(\theta_3)$, we see that θ_3 is executable from the initial state of Example 5, since its path condition reduces to `flag == true`. Observe that although θ_0 and θ_3 have the same path condition, their final states differ. The final state for θ_3 when executed from the initial program state of Example 5 will be $x \mapsto 5$, $y \mapsto 1$, `flag` \mapsto true.

To easily decide whether two traces are in the same equivalence class, we define a *canonical representation* for the traces of GCL executions. In general, a lexicographic ordering on traces can be used to select the smallest in a set of traces, assuming a total order on the elements of the traces. Traces in the same equivalence class differ only in the ordering of adjacent, commuting labels [35]. Hence, a partial order on labels that commute will suffice to define the canonical representative for the traces in an equivalence class. This partial order can for example be expressed by lifting a strict total order on the process identifiers, since commuting events must belong to different processes.

► **Definition 12** (Canonical representatives for GCL traces). *Assume a strict total order $<$ on process identifiers $\iota_0, \iota_1, \iota_2, \dots$. For any labels l_1 and l_2 with process identifiers ι_1 and ι_2 , respectively, let $l_1 < l_2$ if and only if $\iota_1 < \iota_2$. The canonical representative of a trace θ , denoted $canon(\theta)$, is defined inductively as follows:*

$$\begin{aligned}
canon(\varepsilon) &= \varepsilon \\
canon(\varepsilon \cdot l) &= \varepsilon \cdot l \\
canon(\theta \cdot l_1 \cdot l_2) &= canon(\theta \cdot l_1) \cdot l_2 && \text{if } l_1 \not\sim l_2 \text{ or } l_1 < l_2 \\
canon(\theta \cdot l_1 \cdot l_2) &= canon(canon(\theta \cdot l_2) \cdot l_1) && \text{if } l_1 \sim l_2 \text{ and } l_2 < l_1
\end{aligned}$$

If we consider the traces from Examples 5 and 10 and a strict total order $\iota_0 < \iota_1 < \iota_2$ on process identifiers, we can observe that $canon(\theta_0) = \theta_1$.

The set of processes in a runtime state of a GCL program can be derived from the trace leading to that state. We define a function which, for a given trace, returns the set of process identifiers for these processes.

► **Definition 13** (Active processes). *Let θ be a symbolic trace representing the execution of a GCL program. The set of active process identifiers $proc(\theta)$ is defined inductively as follows:*

$$\begin{aligned}
proc(\varepsilon) &= \{\iota_0\} \\
proc(\theta' \cdot e \triangleright \text{spawn}(\iota)) &= proc(\theta') \cup \{\iota\} \\
proc(\theta' \cdot l) &= proc(\theta') \text{ for } l \in \{\tau, \iota(e \triangleright x := e')\}
\end{aligned}$$

This definition is easily justified: If θ is a trace of σ , $\{\iota_0(g)\} \rightarrow^* \sigma', P$, then $P = proc(\theta)$. The proof is straightforward by induction over the length of θ .

We can now formalize what it means for a symbolic trace to be executable. Let \leq denote the prefix relation on symbolic traces. and $\theta \downarrow_\iota$ the projection of a symbolic trace θ unto all labels of of process identifier ι (i.e., labels of the form $\iota(\dots)$). For a symbolic trace θ , let $length(\theta)$ denote the length of θ and θ_n the initial prefix of θ of length n (i.e., $\theta_n \leq \theta$ and $length(\theta_n) = n$).

► **Definition 14** (Trace executability). *Let g be a GCL program, σ a state, l a label with process identifier ι , and assume that θ is a permutation of a symbolic trace of the execution $\sigma, \{\iota_0(g)\} \rightarrow^* \sigma', P$. The trace θ is executable in σ if $\sigma(path(\theta)) = \mathbf{True}$ and $\iota \in proc(\theta_{n-1})$ for all $n \leq length(\theta)$, where $\theta_n = \theta_{n-1} \cdot l$.*

Trace executability allows us to strengthen Theorem 8 by expressing that the source program can produce executable permutations of one of its traces. We say that a permutation θ' of a symbolic trace θ preserves local order if $\forall \iota \in proc(\theta) : \theta' \downarrow_\iota = \theta \downarrow_\iota$.

► **Theorem 15** (Soundness). *For any GCL program g and state σ , such that θ is a symbolic trace of $\sigma, \{\iota_0(g)\} \rightarrow^* \sigma', P$ and let θ' be a permutation of θ which preserves local order. If θ' is executable in σ then there exists a σ'' such that θ' is a symbolic trace of $\sigma, \{\iota_0(g)\} \rightarrow^* \sigma'', P$.*

Proof. The proof proceeds by induction on the length of θ (assuming that for the base case $g(\epsilon) = \mathbf{skip}$). ◀

5.2 The Fagiani Algorithm

We now present an algorithm¹ which, given an initial state σ and an initial symbolic trace θ generated by a successfully terminating concolic execution in the above labeled transition system, constructs a set I of all the canonical representatives of permutations of θ which are executable in σ ; i.e., the algorithm generates one representative for each equivalence class of the traces which are permutations of θ . For simplicity, we assume that θ is in canonical form. Considering our running example, if we start the algorithm with state σ and trace $canon(\theta_0)$ from Example 5, the algorithm will compute traces such as θ_1 , θ_2 and θ_3 from Example 10. The algorithm will detect that $canon(\theta_1) = canon(\theta_0)$ so $canon(\theta_1)$ can be discarded, that $canon(\theta_2)$ is not executable from the initial state σ of Example 5 and can therefore be discarded, and that $canon(\theta_3) \neq canon(\theta_0)$ and therefore $canon(\theta_3)$ is added to I .

The algorithm is presented in the form of pseudo code in Algorithm 1. In the pseudo code we abstract from the data structures representing states and traces. Let θ' be a symbolic trace such that for any process ι its local computation in θ' is a prefix of its local computation in the initial symbolic trace θ . We then denote by $next(\iota, \theta')$ the next instruction of process ι as defined by θ . Formally, the next instruction of a process can be defined as follows:

► **Definition 16** (Process-local next). *Let θ and θ' be symbolic traces, $\iota \in proc(\theta)$, and assume that $\theta' \downarrow_\iota \leq \theta \downarrow_\iota$. The next ι -event after θ' is defined as follows:*

$$\begin{aligned} next(\iota, \theta') &= l \text{ if } (\theta' \cdot l) \downarrow_\iota \leq \theta \downarrow_\iota \\ next(\iota, \theta') &= nil \text{ if } \theta' \downarrow_\iota = \theta \downarrow_\iota \end{aligned}$$

We use the process-local next to ensure that new traces preserve local order. In the code we also assume the inductive definitions of the path condition $path(\theta)$ (see Definition 6), the canonical representative $canon(\theta)$ (see Definition 12) and the active processes $proc(\theta)$

¹ A prototype implementation of the concolic semantics of GCL and the Fagiani algorithm is available at <https://github.com/larstvei/GCL>.

10:12 Inseguendo Fagiani Selvatici

(see Definition 13) of a symbolic trace θ , and the inductive definition of the value $\sigma(e)$ of expression e in a state σ . As before, $length(\theta)$ denotes the length of the trace θ , i.e., the number of instructions it contains, and ϵ the empty trace.

The algorithm itself iterates over the length of the initial trace θ in canonical form. Each such iteration in turn iterates over all the traces currently stored in I . Since this inner iteration updates the set I , we “freeze” the initial value of I upon each iteration of the outer for-loop. The set I will then store the newly updated canonical traces.

■ **Algorithm 1** The Fagiani Algorithm.

Input: θ_0 : a global symbolic trace in its canonical form
Input: σ : an initial state
Auxiliaries: $path, canon, proc, next$: see Definitions 6, 12, 13, 16
Result: I : A set of executable permutations of θ_0
 $I := \{\epsilon\};$
for $i := 1$ **to** $length(\theta_0)$ **do**
 $I' := I;$
 $I := \{\};$
 foreach $\theta' \in I'$ **do**
 foreach $\iota \in proc(\theta')$ **do**
 if $next(\iota, \theta') \neq nil$ **then**
 $\theta'' := canon(\theta' \cdot next(\iota, \theta'));$
 if $\theta'' \notin I \wedge \sigma(path(\theta''))$ **then** $I := I \cup \{\theta''\};$
 end
 end
 end
end

Formally, the computed set I of permutations, which respect the local order of the input trace θ , satisfies

$$I = \{canon(\theta') \mid \sigma(path(\theta')) \wedge \forall \iota \in proc(\theta') : \theta' \downarrow_{\iota} = \theta \downarrow_{\iota}\}.$$

To prove this, it suffices to show that after the i 'th iteration, $i = 1, \dots, length(\theta)$, I satisfies

$$I = \{canon(\theta') \mid length(\theta') = i \wedge \sigma(path(\theta')) \wedge \forall \iota \in proc(\theta') : \theta' \downarrow_{\iota} \leq \theta \downarrow_{\iota}\}.$$

The computed set I forms the basis for a set of test cases. We use this set for testing whether these different traces have an observable effect on the state. It is easy to see that the set I consists of executable traces, such that Theorem 15 applies. For each trace the corresponding test case simply consists of the underlying scheduling policy, which is represented by a sequence of process identifiers. The execution of such a test case then consists of an execution from the given initial state σ following the specified scheduling policy, and checking the final state.

6 Language Extensions

This section presents some conservative extensions to the language: nested guarded statements, a looping construct, named procedures, and local scopes for variables.

Nesting

Guarded statements may be nested without adding any particular complexity to the calculus. Let us consider statements with the syntax $e \triangleright g$ and $g_1 \triangleleft e \triangleright g_2$. The guarded skip $e \triangleright \mathbf{skip}$, which corresponds to an assert-statement, needs an additional label of the form $\iota(e \triangleright \mathbf{skip})$. With this extension, it has the obvious semantics of SKIP provided that the guard e holds in the current state, and with the above label. We get the following rules in the semantics:

$$\begin{array}{c}
 \text{(NESTEDGUARD)} \\
 \frac{\sigma, \{\iota((e_1 \wedge e_2) \triangleright g; g')\} \xrightarrow{l} \sigma', P}{\sigma, \{\iota(e_1 \triangleright (e_2 \triangleright g); g')\} \xrightarrow{l} \sigma', P} \\
 \\
 \begin{array}{cc}
 \text{(NESTEDCHOICE1)} & \text{(NESTEDCHOICE2)} \\
 \frac{\sigma, \{\iota((e_1 \wedge e_2) \triangleright g_1; g)\} \xrightarrow{l} \sigma', P}{\sigma, \{\iota(e_1 \triangleright (g_1 \triangleleft e_2 \triangleright g_2); g)\} \xrightarrow{l} \sigma', P} & \frac{\sigma, \{\iota(\neg(e_1 \wedge e_2) \triangleright g_2; g)\} \xrightarrow{l} \sigma', P}{\sigma, \{\iota(e_1 \triangleright (g_1 \triangleleft e_2 \triangleright g_2); g)\} \xrightarrow{l} \sigma', P}
 \end{array}
 \end{array}$$

Loops

We can add a loop construct $e \triangleright^* s$ to repeat a guarded statement zero or more times, which can be captured by the following transition rules:

$$\begin{array}{cc}
 \text{(WHILE1)} & \text{(WHILE2)} \\
 \frac{\sigma, \{\iota(e \triangleright s; e \triangleright^* s; g)\} \xrightarrow{l} \sigma', P}{\sigma, \{\iota(e \triangleright^* s; g)\} \xrightarrow{l} \sigma', P} & \frac{\neg\sigma(e)}{\sigma, \{\iota(e \triangleright^* s; g)\} \xrightarrow{\epsilon} \sigma', \{\iota(g)\}}
 \end{array}$$

Note that such a loop construct would require a straightforward generalization of our concolic testing theory to non-terminating computations by imposing a bound on the length of the computations.

Procedures

It is also straightforward to spawn new processes by procedure calls: We can add procedure definitions $\mathbf{proc} \ p \{g\}$ and add syntax $p()$ for procedure calls to the statements. If we assume given a mapping PT from procedure names p to procedure bodies g , it suffices to add the following rule to the semantics:

$$\begin{array}{c}
 \text{(PROC)} \\
 PT(p) = g' \\
 \frac{\sigma, \{\iota(e \triangleright \mathbf{spawn}(g'); g)\} \xrightarrow{l} \sigma', P}{\sigma, \{\iota(e \triangleright p(); g)\} \xrightarrow{l} \sigma', P}
 \end{array}$$

To model procedure calls locally in the context of a single process can be described by inlining the procedure body.

Local scopes

Local scopes $\{\sigma', g\}$ can also be added as guarded statements. Here, we are not interested in seeing the local variables in the labels, because they are private and do not affect other processes. For this reason, we remove local variables from the labels by applying the local

substitution *before* we (in most cases) reuse rules without local scope to create the labels. We need additional rules for scoped assignment and spawning, for leaving an empty scope and we assume that an inner scope takes precedence over an outer scope to unfold nested scopes.

$$\begin{array}{c}
\text{(LEAVESCOPE)} \\
\sigma, \{\iota(\{\sigma', \mathbf{skip}\}; g)\} \xrightarrow{\epsilon} \sigma, \{\iota(g)\} \\
\\
\text{(SCOPEDESPAWN)} \\
\frac{P' = P \cup \{\iota(\{\sigma', g'\}; g'')\}}{\sigma, \{\iota(\sigma'(e) \triangleright \mathbf{spawn}(g); \mathbf{skip})\} \xrightarrow{l} \sigma, P} \\
\\
\text{(UNFOLDSCOPE)} \\
\frac{\sigma, \{\iota(\{\sigma'', g\}; \{\sigma', g'\}; g'')\} \xrightarrow{l} \sigma''', P}{\sigma, \{\iota(\{\sigma', \{\sigma'', g\}; g'\}; g'')\} \xrightarrow{l} \sigma''', P} \\
\\
\text{(SCOPEDESSIGN1)} \\
\frac{x \in \text{dom}(\sigma) \quad \sigma, \{\iota(\sigma'(e) \triangleright x := \sigma'(e'); g)\} \xrightarrow{l} \sigma'', \{\iota(g)\}}{\sigma, \{\iota(\{\sigma', e \triangleright x := e'; g\}; g'')\} \xrightarrow{l} \sigma'', \{\iota(\{\sigma', g\}; g')\}} \\
\\
\text{(SCOPEDESSIGN2)} \\
\frac{x \notin \text{dom}(\sigma) \quad e'' = (e \wedge (e' == e')) \quad \sigma', \{\iota(\sigma(e) \triangleright x := \sigma(e'); g)\} \xrightarrow{l} \sigma'', \{\iota(g)\}}{\sigma, \{\iota(\{\sigma', e \triangleright x := e'; g\}; g')\} \xrightarrow{\iota(\sigma'(e'') \triangleright \mathbf{skip})} \sigma, \{\iota(\{\sigma'', g\}; g')\}}
\end{array}$$

7 Related Work

Parallel and distributed systems are difficult to analyze because of their inherent non-determinism. Both testing and formal verification have their limitations for these systems. Model checking, which can be situated between testing and formal verification, here suffers from state explosion [19]; in practice, model checking relies on analyzing models with a tractable state space. Software model checking techniques either adapt model checking into techniques for systematic testing of programs (e.g., [4, 18, 30, 44]) or abstract programs into models for which traditional model checking techniques apply (e.g., [8, 20, 32, 36, 38]). Our work fits into the former category.

Stateless model checking avoids state space explosion by exploring the executions of a program without explicitly storing all the program states [29], and has been implemented in tools including VeriSoft [30] and CHESS [37]. Stateless model checking can be realized by combining a runtime scheduler which controls the program execution with an algorithm which explores the different ways in which processes can be scheduled. The combinatorial explosion of different executions for parallel programs can be reduced by means of partial order reduction (POR) [19, 29, 39], which introduces an equivalence relation on executions based on Mazurkiewicz traces [35]. POR explores at least one execution in each equivalence class. Ideally, only one trace of each equivalence class needs to be explored; the precision (i.e., performance) of a particular algorithm depends on the number of execution paths visited in each equivalence class. Dynamic partial order reduction (DPOR) [1, 27, 34, 39, 40, 43] makes POR more precise by detecting and exploiting interference dynamically. DPOR assumes access to the scheduler and state of the runtime system, both to guide execution and to decide whether an action is enabled. Our work uses partial order reduction, but it does not need access to the scheduler of the runtime system.

Actor systems [2] are well-suited for systematic testing using POR because their inherent isolation of local state limits the number of races in a program. TransDPOR [40] extends DPOR to explore that the dependency relation of actor systems is transitive. SYCO [4, 5] is a testing tool for actor-based concurrency which combines the transitivity exploited by

TransDPOR with a dependency relation based on process interference [6], similar to the non-interference relation discussed in Section 2, and to consider synchronization primitives as found in active object languages [10], such as ABS [33]; i.e., they handle await-statements which synchronize on the resolution of futures. ContextDPOR [3] introduces a context-sensitive notion of non-interference between events. This is achieved by deciding on the equivalence between subsequences of the traces and the next action (resulting in so called sleep sequences). Technically, this is done by storing the state resulting from the one trace together with the sleep sequence. In contrast, our motivation for studying GCL stems from the problem of swapping events in the traces with intra-actor synchronization based on non-monotonic Boolean await-statements (in contrast to futures, which keep an enabled state after reaching it). Similar to ContextDPOR, we had to go beyond the read- and write-sets traditionally used to determine interference in order to decide at the level of traces whether the permutation of an observed trace is executable. In contrast to ContextDPOR, our work is based on a weakest-prefix calculation over symbolic traces to decide on their executability. The relationship between concrete and symbolic execution with partial-order reduction has previously been studied by the authors in [21]; that previous work focussed on soundness and correctness of the symbolic execution framework but not on weakest-precondition computation for executability as we address in this paper.

A major limitation of DPOR algorithms is that they are implemented inside the runtime system of the language. We are currently developing ExoDPOR, a stateless model checker for ABS which is implemented outside the runtime system, such that it can perform parallel stateless model checking by exogenously coordinating the runs of a number of instances of the runtime system [41]. This is enabled by extending the backend of ABS with a trace record and replay mechanism [42] and manipulating traces directly to trigger new runs. Whereas ExoDPOR can handle most of ABS (including deployment components and real-time behavior), it does not yet handle await statements with Boolean conditions (a non-monotonic guard statement). We expect the work in this paper to provide a basis to address this currently missing piece in our tool.

8 Conclusion

This paper presented a method for testing the deterministic behavior of dynamically spawned processes executing on a shared state. We have developed an algorithm which, starting from the symbolic trace of an initial run of a program, generates all traces which may result in different final states. Each trace represents an execution with a different local scheduling of the program's processes, but the traces may result in the same final state because the non-interference relation is an approximation. Therefore, the generated traces need to be tested to determine whether the program outcome is independent of the local scheduling decisions. We rely on recording traces of executions, partial order reduction to eliminate traces which are obviously equivalent to previously generated traces, and weakest precondition calculation to eliminate infeasible (non-executable) traces. The weakest precondition calculation allows the proposed method to handle the non-monotonic enabledness conditions of guarded commands without explicitly computing the different states of the program.

This proposed method can be extended in a straightforward way to generate “seed traces”, executable trace prefixes that lead to different end states than the one in the original recorded run. To formulate seed traces, the concolic operational semantics of Section 4 can simply be extended by a new label type that records the conditional in the choice expression. These seed traces can be used to implement stateless model checking for parallel systems given a controllable scheduler. We plan to implement this method as an extension of ExoDPOR [41], our stateless model checker for the active object language ABS.

References

- 1 Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Source sets: A foundation for optimal dynamic partial order reduction. *J. ACM*, 64(4):25:1–25:49, 2017. doi:10.1145/3073408.
- 2 Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, 1986.
- 3 Elvira Albert, Puri Arenas, Maria Garcia de la Banda, Miguel Gómez-Zamalloa, and Peter J. Stuckey. Context-sensitive dynamic partial order reduction. In Rupak Majumdar and Viktor Kuncak, editors, *Proc. 29th International Conference on Computer Aided Verification (CAV 2017), Part I*, volume 10426 of *Lecture Notes in Computer Science*, pages 526–543. Springer, 2017. doi:10.1007/978-3-319-63387-9_26.
- 4 Elvira Albert, Puri Arenas, and Miguel Gómez-Zamalloa. Systematic testing of actor systems. *Softw. Test. Verification Reliab.*, 28(3), 2018. doi:10.1002/stvr.1661.
- 5 Elvira Albert, Miguel Gómez-Zamalloa, and Miguel Isabel. SYCO: a systematic testing tool for concurrent objects. In Ayal Zaks and Manuel V. Hermenegildo, editors, *Proc. 25th International Conference on Compiler Construction (CC 2016)*, pages 269–270. ACM, 2016. doi:10.1145/2892208.2892236.
- 6 Gregory R. Andrews. *Concurrent programming - principles and practice*. Benjamin/Cummings, 1991.
- 7 Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Texts in Computer Science. Springer, 3 edition, 2009. doi:10.1007/978-1-84882-745-5.
- 8 Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A decade of software model checking with SLAM. *Commun. ACM*, 54(7):68–76, 2011. doi:10.1145/1965724.1965743.
- 9 Nikolaos Bezirgiannis, Frank S. de Boer, Einar Broch Johnsen, Ka I Pun, and Silvia Lizeth Tapia Tarifa. Implementing SOS with active objects: A case study of a multicore memory system. In *Proc. 22nd Intl. Conf. on Fundamental Approaches to Software Engineering (FASE 2019)*, volume 11424 of *Lecture Notes in Computer Science*, pages 332–350. Springer, 2019. doi:10.1007/978-3-030-16722-6_20.
- 10 Frank De Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. A survey of active object languages. *ACM Comput. Surv.*, 50(5):76:1–76:39, October 2017. doi:10.1145/3122848.
- 11 Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I. Pun, S. Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. Parallel objects for multicores: A glimpse at the parallel language Encore. In Marco Bernardo and Einar Broch Johnsen, editors, *Formal Methods for Multicore Programming*, volume 9104 of *Lecture Notes in Computer Science*, pages 1–56. Springer, 2015. doi:10.1007/978-3-319-18941-3_1.
- 12 Nadia Busi, Maurizio Gabbrielli, and Gianluigi Zavattaro. Replication vs. recursive definitions in channel based calculi. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP 2003)*, volume 2719 of *Lecture Notes in Computer Science*, pages 133–144. Springer, 2003. doi:10.1007/978-3-540-27836-8_28.
- 13 Nadia Busi, Maurizio Gabbrielli, and Gianluigi Zavattaro. Comparing recursion, replication, and iteration in process calculi. In *Proc. 31st International Colloquium on Automata, Languages and Programming (ICALP 2004)*, volume 3142 of *Lecture Notes in Computer Science*, pages 307–319. Springer, 2004. doi:10.1007/978-3-540-27836-8_28.
- 14 Nadia Busi, Maurizio Gabbrielli, and Gianluigi Zavattaro. On the expressive power of recursion, replication and iteration in process calculi. *Math. Struct. Comput. Sci.*, 19(6):1191–1222, 2009. doi:10.1017/S096012950999017X.
- 15 Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In Ari Juels, Rebecca N. Wright, and Sabrina De Capitani di Vimercati, editors, *Proc. 13th ACM Conf. on Computer and Communications Security (CCS'06)*, pages 322–335. ACM, 2006. doi:10.1145/1180405.1180445.

- 16 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.
- 17 Denis Caromel, Ludovic Henrio, and Bernard Serpette. Asynchronous and deterministic objects. In *Proc. 31st Symposium on Principles of Programming Languages (POPL 2004)*, pages 123–134. ACM Press, 2004. doi:10.1145/964001.964012.
- 18 Maria Christakis. *Narrowing the Gap between Verification and Systematic Testing*. PhD thesis, ETH Zurich, 2015.
- 19 Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 2001. doi:10.3233/978-1-60750-711-6-260.
- 20 James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In Carlo Ghezzi, Mehdi Jazayeri, and Alexander L. Wolf, editors, *Proc. 22nd International Conference on Software Engineering (ICSE 2000)*, pages 439–448. ACM, 2000. doi:10.1145/337180.337234.
- 21 Frank S. de Boer, Marcello Bonsangue, Einar Broch Johnsen, Ka I Pun, Silvia Lizeth Tapia Tarifa, and Lars Tveito. SymPaths: Symbolic execution meets partial order reduction. In Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, and Mattias Ulbrich, editors, *Deductive Verification - The Next 70 Years*, volume 12345 of *Lecture Notes in Computer Science*. Springer, 2020. To appear.
- 22 Frank S. de Boer, Dave Clarke, and Einar Broch Johnsen. A complete guide to the future. In *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *Lecture Notes in Computer Science*, pages 316–330. Springer, 2007. doi:10.1007/978-3-540-71316-6_22.
- 23 Frank S. de Boer and Hans-Dieter A. Hiep. Axiomatic characterization of trace reachability for concurrent objects. In Wolfgang Ahrendt and Silvia Lizeth Tapia Tarifa, editors, *Proc. 15th Intl. Conf. on Integrated Formal Methods (IFM 2019)*, volume 11918 of *Lecture Notes in Computer Science*, pages 157–174. Springer, 2019. doi:10.1007/978-3-030-34968-4_9.
- 24 Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975. doi:10.1145/360933.360975.
- 25 Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
- 26 Kiko Fernandez-Reyes, Dave Clarke, Ludovic Henrio, Einar Broch Johnsen, and Tobias Wrigstad. Godot: All the benefits of implicit and explicit futures. In Alastair F. Donaldson, editor, *Proc. 33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, volume 134 of *LIPICs*, pages 2:1–2:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICs.ECOOP.2019.2.
- 27 Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In Jens Palsberg and Martín Abadi, editors, *Proc. 32nd Symp. on Principles of Programming Languages (POPL 2005)*, pages 110–121. ACM, 2005. doi:10.1145/1040305.1040315.
- 28 Maurizio Gabbrielli and Simone Martini. *Programming Languages: Principles and Paradigms*. Springer, 2010. doi:10.1007/978-1-84882-914-5.
- 29 Patrice Godefroid. *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. doi:10.1007/3-540-60761-7.
- 30 Patrice Godefroid. Model checking for programming languages using Verisoft. In Peter Lee, Fritz Henglein, and Neil D. Jones, editors, *Proc. 24th Symp. on Principles of Programming Languages (POPL 1997)*, pages 174–186. ACM, 1997. doi:10.1145/263699.263717.
- 31 Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In Vivek Sarkar and Mary W. Hall, editors, *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*, pages 213–223. ACM, 2005. doi:10.1145/1065010.1065036.

- 32 Gerard J. Holzmann and Margaret H. Smith. A practical method for verifying event-driven software. In Barry W. Boehm, David Garlan, and Jeff Kramer, editors, *Proc. International Conference on Software Engineering (ICSE 1999)*, pages 597–607. ACM, 1999. doi:10.1145/302405.302710.
- 33 Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, and Martin Steffen. ABS: A core language for abstract behavioral specification. In Bernhard Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *Lecture Notes in Computer Science*, pages 142–164. Springer, 2011. doi:10.1007/978-3-642-25271-6_8.
- 34 Shmuel Katz and Doron A. Peled. An efficient verification method for parallel and distributed programs. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *Lecture Notes in Computer Science*, pages 489–507. Springer, 1988. doi:10.1007/BFb0013032.
- 35 Antoni W. Mazurkiewicz. Trace theory. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Advances in Petri Nets 1986*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer, 1987. doi:10.1007/3-540-17906-2_30.
- 36 Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A pragmatic approach to model checking real code. In David E. Culler and Peter Druschel, editors, *Proc. 5th Symposium on Operating System Design and Implementation (OSDI 2002)*. USENIX Association, 2002. URL: <http://www.usenix.org/events/osdi02/tech/musuvathi.html>.
- 37 Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gérard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In Richard Draves and Robbert van Renesse, editors, *Proc. 8th Symposium on Operating Systems Design and Implementation (OSDI 2008)*, pages 267–280. USENIX Association, 2008. URL: http://www.usenix.org/events/osdi08/tech/full_papers/musuvathi/musuvathi.pdf.
- 38 Kedar S. Namjoshi and Robert P. Kurshan. Syntactic program transformations for automatic abstraction. In E. Allen Emerson and A. Prasad Sistla, editors, *Proc. 12th International Conference on Computer Aided Verification (CAV 2000)*, volume 1855 of *Lecture Notes in Computer Science*, pages 435–449. Springer, 2000. doi:10.1007/10722167_33.
- 39 Doron A. Peled. All from one, one for all: on model checking using representatives. In Costas Courcoubetis, editor, *Proc. 5th International Conference on Computer Aided Verification (CAV'93)*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer, 1993. doi:10.1007/3-540-56922-7_34.
- 40 Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. TransDPOR: A novel dynamic partial-order reduction technique for testing actor programs. In Holger Giese and Grigore Rosu, editors, *Proc. Formal Techniques for Distributed Systems (FORTE/FMOODS 2012)*, volume 7273 of *Lecture Notes in Computer Science*, pages 219–234. Springer, 2012. doi:10.1007/978-3-642-30793-5_14.
- 41 Lars Tveito, Einar Broch Johnsen, and Rudolf Schlatte. ExoDPOR: Exogenous stateless model checking. Submitted for publication, 2020.
- 42 Lars Tveito, Einar Broch Johnsen, and Rudolf Schlatte. Global reproducibility through local control for distributed active objects. In Heike Wehrheim and Jordi Cabot, editors, *Proc. 23rd International Conference on Fundamental Approaches to Software Engineering (FASE 2020)*, volume 12076 of *Lecture Notes in Computer Science*, pages 140–160. Springer, 2020. doi:10.1007/978-3-030-45234-6_7.
- 43 Antti Valmari. Stubborn sets for reduced state space generation. In Grzegorz Rozenberg, editor, *Proc. 10th International Conference on Applications and Theory of Petri Nets*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer, 1989. doi:10.1007/3-540-53863-1_36.
- 44 Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003. doi:10.1023/A:1022920129859.