ASE 2017, pp 776-781

Quick Verification of Concurrent Programs by Iteratively Relaxed Scheduling

Patrick Metzler, Habib Saissi, Péter Bokor, Neeraj Suri
Technische Univeristät Darmstadt, Germany
{metzler, saissi, pbokor, suri}@deeds.informatik.tu-darmstadt.de

***Abstract*—The most prominent advantage of software verification over testing is a rigorous check of every possible software behavior. However, large state spaces of concurrent systems, due to non-deterministic scheduling, result in a slow automated verification process. Therefore, verification introduces a large delay between completion and deployment of concurrent software.**

This paper introduces a novel iterative approach to verification of concurrent programs that drastically reduces this delay. By restricting the execution of concurrent programs to a small set of *admissible schedules*, verification complexity and time is drastically reduced. Iteratively adding admissible schedules after their verification eventually restores non-deterministic scheduling. Thereby, our framework allows to find a sweet spot between a low verification delay and sufficient execution time performance. Our evaluation of a prototype implementation on well-known benchmark programs shows that after verifying only few schedules of the program, execution time overhead is competitive to existing deterministic multi-threading frameworks.

I. Introduction

Automated verification of concurrent programs with non-deterministic scheduling is known to be challenging: verification has to consider all possible schedules a concurrent program may be executed with, which may result in exponentially many states to be verified, known as state space explosion [1].

Partial order reduction (POR) is able to reduce the state space of a concurrent program by identifying equivalence classes of program executions such that only one representative of each class needs to be verified [2]–[4]. Such an equivalence class is called *Mazurkiewicz trace* or simply *trace*. However, the reduced state space may still be of exponential size [3]. Hence, the high complexity of state space exploration for concurrent systems remains and hinders a wide application of automated verification (e.g., model checking) in industrial software development.

In particular, verification introduces a considerable delay between completion and deployment of software. When development of a candidate program is complete, it may be deployed only after the verifier approved that it is correct under all possible schedules. This *verification delay* reaches large values for benchmark programs even if state-of-the-art POR is used [5], [6]. We conjecture that the verification delay is unacceptably high for large areas of industrial software development. Nevertheless, individual traces can be verified quickly as can be seen when relating verification time to the number of explored traces.

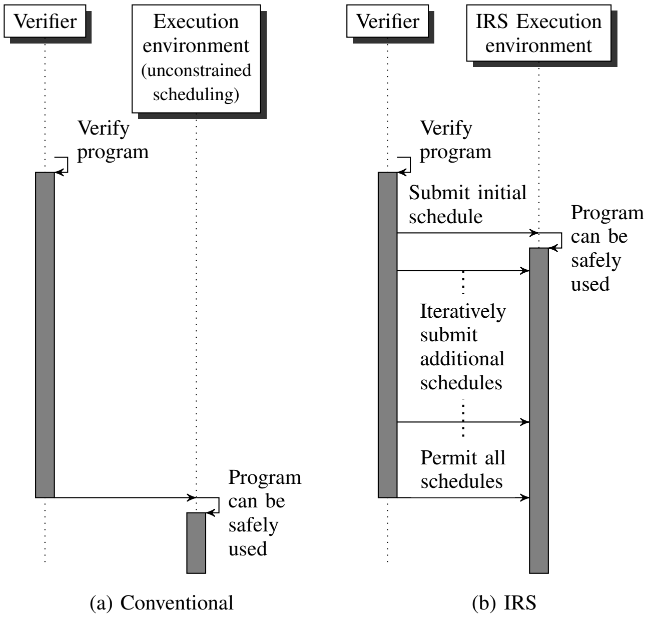
In the area of concurrency testing, deterministic multi-threading (DMT) can help to reduce the number of necessary test cases. Several techniques exist to restrict scheduling such that (1) scheduling is deterministic for a particular input and/or (2) only a reduced set of schedules may occur [7], [8]. Such DMT techniques trade potential execution time overhead (compared to executing the unmodified program) for reduced non-determinism that may simplify *testing*. However, these approaches do not allow to control the schedule that will occur in advance, which renders them unsuitable for automated *verification* of concurrent programs. We are not aware of any existing tool or concept that allows to execute selected schedules after they are successfully verified. Moreover, it is not possible to adjust the amount of non-determinism in existing DMT approaches.

We propose to make the amount of non-determinism and thereby the verification delay adjustable by using intermediate verification results and reducing non-determinism by dynamically constrained scheduling. In particular, instead of waiting for verification to complete, we propose to use intermediate verification results that guarantee program correctness for one or more schedules. By generalizing the concept of Mazurkiewicz traces [9] for symbolic model checking, we are able to use POR as a verification technique with intermediate verification results. As soon as a single trace is verified, the program may be used inside a suitable execution environment with constrained scheduling. By continuously verifying and permitting more traces, scheduling constraints are iteratively relaxed and, as shown by our evaluation, execution time overhead is reduced. Thus, execution time overhead can be traded for verification delay.

We provide the following contributions. (1) We develop a formal framework for iterative, automated verification of concurrent programs. (2) We introduce the concept of *symbolic traces* as an extension and generalization of Mazurkiewicz traces to symbolic model checking. (3) We discuss implementation issues for a suitable execution environment using our prototype implementation for LLVM programs [10]. (4) We show experimentally that only few traces need to be verified to considerably reduce execution time overhead.

II. Overview

We propose *iteratively relaxed scheduling* (IRS) for concurrent programs with non-deterministic scheduling as an


Verifier
Execution environment (unconstrained scheduling)
Verify program
Program can be safely used
Verifier
IRS Execution environment
Verify program
Submit initial schedule
Program can be safely used
Iteratively submit additional schedules
Permit all schedules

(a) Conventional (b) IRS

Figure 1: The program verification process (sequence diagram)

improvement over the conventional approach to program verification:

(1) Develop a program (or program update).
(2) Verify the program.
(3) The program can be safely used (with unconstrained scheduling).

In case verification is successful, correctness is ensured for all feasible schedules of the program. This guaranty comes at the price of a typically large verification delay because of exponentially many schedules when unconstrained (non-deterministic) scheduling is used.

Instead of waiting until the program is verified for all feasible schedules, we propose to start using the program already after a use case-specific threshold of schedules has been verified. Correctness is guaranteed by wrapping the program in an *IRS execution environment*, which permits only verified schedules by constraining scheduling. We provide details about IRS execution environments in Section III. Specifically, verifying a concurrent program with IRS proceeds as follows.

(1) Develop a program (or program update).
(2) Continuously verify individual schedules or sets of schedules.
(3) As soon as one admissible schedule is available, the program can be safely used inside an IRS execution environment.
(4) Additional verified schedules may be added during program usage to relax scheduling constraints.

The difference between conventional verification and IRS is illustrated in Figure 1. While conventionally the verification delay corresponds to the full verification time, IRS enables to adjust verification delay and the amount of non-determinism in scheduling: the longer the verification delay, the more schedules are verified and the fewer scheduling constraints are necessary to enforce that only admissible schedules may occur.

Constraining scheduling presumably introduces considerable execution time overhead. While execution time overhead may be considerable, relaxing scheduling constraints is able to quickly reduce this overhead. Our experiments show that iteratively relaxing scheduling constraints also iteratively reduces execution time overhead, which we detail in Section V.

Given a positive relationship between relaxed scheduling constraints and decreased execution time overhead, IRS may be used to exploit the sweet spot between a short verification delay and small execution time overhead. In other words, IRS enables to use *as much* non-determinism in scheduling as needed for execution time performance and *no more* non-determinism than necessary in order to limit the verification delay.

An additional advantage of IRS over conventional program verification is that programs that show both correct and erroneous schedules can be used safely, as erroneous schedules are never enabled. Such a program may be either corrected such that eventually, all schedules can be enabled, or left unchanged such that the program is used with erroneous schedules disabled. In contrast, conventional verification requires to correct the program so that the program is only available after verification is restarted and completed successfully. Please note that it is well possible that a program that is used inside an IRS execution environment has erroneous schedules. However, program usage inside the IRS environment is always safe as only correct schedules are enabled. The only limitation in such a case is that the program cannot be used safely with all schedules enabled (e.g., outside the IRS execution environment).

Several scenarios of how to use IRS are conceivable, e.g.:

(1) Safely deploy programs with large state spaces due to concurrency that are infeasible to fully verify.
(2) In case a program update introduces a bug but correct schedules can still be found, safely deploy the program with erroneous schedules disabled until the bug is fixed.
(3) For a given time budget for verification (maximum verification delay), verify as much schedules as possible until the threshold is reached and deploy the program (with the remaining schedules disabled). Verification delay is reduced.
(4) For a given budget of execution time performance (e.g., maximum execution time overhead), start verification and continuously test the execution time performance for the so far verified schedules. As soon as the program is fast enough, deploy the program (with the remaining schedules disabled). Verification delay is reduced.
(5) In addition to (4), continue verification after deployment and continuously extend the set of verified and enabled schedules. Execution time performance is increased after deployment. (It is required to update scheduling con-

straints online. With a suitable implementation, it is not necessary to update the program itself.)

Depending on the specific requirements of a use case, it may be suitable or even necessary to combine or extend these scenarios. As we detail in Section IV, it is possible to realize an IRS execution environment completely inside an application program, without modifying the operating system.

In order to execute programs safely inside an IRS execution environment (i.e., such that the specification is never violated) while preserving usability (i.e., such that full program functionality is available), it must be ensured that intermediate verification results correspond to scheduling constraints that describe how to remain inside the known-to-be-safe state space for arbitrary program inputs. Thereby, it is guaranteed that the program is safely executed regardless of the current input. While this requirement may be strong for programs with non-deterministic inputs, the same limitation applies for verification of sequential programs.

Although there exist techniques to deterministically execute concurrent programs (e.g., [7], [8], [11]) in order to support concurrency *testing*, IRS constitutes a novel approach to *verifying* concurrent programs. To our best knowledge, all existing DMT approaches depend on concrete program inputs for enforcement of a deterministic execution: no guaranty is given about which schedule is used after a change in program inputs. However, program verification requires correct behavior for every possible program input. Consequently, using an existing DMT approach for verification would require to verify a program separately for each individual input, which is typically infeasible.

III. Approach

A key requirement for a verification technique to be useful in conjunction with IRS is to yield meaningful intermediate verification results. Otherwise, safe execution of the program would have to wait until the program has been verified for all schedules and IRS would be reduced to conventional verification. Meaningful intermediate verification results either show a counter example for program correctness or guarantee correctness under certain scheduling constraints. No additional constraints should be necessary such as constraints about program inputs or execution length, as a program may not be fully operational under such constraints. Therefore, techniques such as explicit state model checking or bounded model checking are unsuitable for IRS.

Partial order reduction (POR) is a state space reduction technique suitable for symbolic model checking [12], although it is often presented for explicit-state model checking [4]–[6]. We choose POR as a verification technique to instantiate IRS and extend the notion of Mazurkiewicz traces to a novel concept of *symbolic traces* in order to support symbolic model checking with meaningful intermediate verification results. An alternative technique for reducing the complexity of non-deterministic scheduling, iterative context bounding (ICB) [13], would equally fit to produce meaningful intermediate verification results. However, to our knowledge, ICB has not been applied to symbolic model checking before.

A. System Model

A (concurrent) *program* P is a transition system $(S, S_{init}, \Sigma, \rightarrow)$ where S is a finite set of states, $S_{init} \subseteq S$ is a set of initial states (program intputs), Σ is a finite set of threads, and $\rightarrow_P \subseteq (S \times \Sigma) \rightharpoonup S$ is an acyclic transition relation (for a given state and thread, there is at most one successor state). We write $s_1 \xrightarrow{t} s_2$ to denote $(s_1, t, s_2) \in \rightarrow$.

A *partial execution* of P is an initial state (the program input for this execution) and a sequence $(s_0, u) \in S_{init} \times \Sigma^*$, where $u = t_1 \dots t_n$ such that there exist states $s_1, \dots, s_n$ with $s_0 \xrightarrow{t_1} s_1 \cdots \xrightarrow{t_n} s_n$ (s_0 may be omitted if it is clear from the context or arbitrary). In order to uniquely describe each occurrence t_i of a thread in u, it is associated with an *event* $e \in \Sigma \times \mathbb{N}$ such that $e = (t, k)$ with $k = |\{t_j : j < i \wedge t_j = t_i\}|$, i.e., e specifies the thread t_i and the number of thread occurrences of t_i that occur before position i in u.

We assume the existence of a *dependency* relation for P that induces a happens-before relation between events and a notion of Mazurkiewicz equivalence on partial executions [6], [9]. We extend the notion of Mazurkiewicz traces to symbolic traces as follows. A *symbolic trace* or simply *trace* o of P is a graph $o = (E_o, C_o, \rightarrow_o)$ that represents a partial order, the happens-before relation, of some partial execution u of P and all partial executions that are Mazurkiewicz equivalent to u. We say that u is a *linearization* of o. Let $u = t_1 \dots t_n$ be an arbitrary linearization of o such that there exist states $s_1, \dots, s_n$ with $s_0 \xrightarrow{t_1} s_1 \cdots \xrightarrow{t_n} s_n$. $E_o = \{e_1, \dots, e_n\}$ is a set of events such that e_i corresponds to t_i for $1 \leq i \leq n$. C_o is a set of sets of path constraints (that may be collected during model checking). $\rightarrow_o \subseteq E_o \times C_o \times E_o$ is an edge relation between events annotated with path constraints such that for every event $e_i \in E_o$ and every incoming edge of e_i with path constraints C, s_i satisfies C. We write *traces*(P) for a set of traces that completely cover the state space of P, i.e., it is sufficient to verify all traces in *traces*(P) in order to decide correctness of P.

B. Algorithm

Given a program P, we define an IRS execution environment that is able to (1) continuously receive representations of admissible traces from a verifier and (2) schedule the program P such that only admissible executions occur. Verifier and execution environment of P are defined by Algorithm 1.

During execution of a program, the IRS execution environment maintains that the current partial execution (s_0, u) adheres to the scheduling constraints represented by some admissible trace o, for which we write $(s_0, u) \preccurlyeq o$. We formalize this notion for a sequence $u = t_1 \dots t_n$ with $s_0 \xrightarrow{t_1} s_1 \cdots \xrightarrow{t_n} s_n$ for some states $s_0, \dots, s_n$ and o with events $E_o = \{e_1, \dots, e_m\}$ as $(s_0, u) \preccurlyeq o$ if

(1) u is empty or

Algorithm 1: IRS

```
Data: $V$ – the set of admissible traces, initially empty
Verifier:
    for each trace $o$ in $traces(P)$ do
        verify $o$;
        if $o$ is correct then
            add $o$ to the set of admissible traces $V$;
Execution environment:
    set the current partial execution $(s_0, u)$ to the empty sequence;
    while $P$ has not terminated do
        choose some thread $t$ from $admissible(V, s_0, u)$;
        execute the next event of $t$;
        append $t$ to $u$;
```

(2) there exists some $e_i \in E_o$ that represents t_1 and for all incoming edges with path constraints C, s_0 does not satisfy C and $(s_1, t_2 \ldots t_n) \preccurlyeq remove(e_i, o)$

where $remove(e_i, o)$ is o with e_i and all incoming and outgoing edges of e_i removed. Intuitively, $(s_0, u) \preccurlyeq o$ can be checked as follows: if u is empty, the condition is satisfied, as they do not contain any events that can violate any ordering given by o. If u is not empty, check whether the first element of u corresponds to an event e_i in o that has no incoming edge that satisfies the current path constraints (i.e., no event has to be scheduled before under the current program inputs). The condition is satisfied if u without its first element adheres to o with e_i and all adjacent edges removed.

For a given partial execution (s_0, u) and a set of admissible traces V, we define the set of all threads that can be executed next, without violating adherence to an admissible trace, as $admissible(V, s_0, u) := \{t \in \Sigma_P : \exists o \in V. (s_0, u \cdot t) \preccurlyeq o\}$.

When large fractions of a program's state space are to be explored, i.e., when V contains many traces, time and space complexity of checking $t \in admissible(V, s_0, u)$ and $(s_0, u) \preccurlyeq o$ may be relevant. An efficient implementation of Algorithm 1 may use, for example, a compact representation of multiple traces in a single data structure.

The execution environment representation of Algorithm 1 corresponds to the interleaving semantics of concurrent programs. It does not show explicitly when threads wait for permission to execute their next memory access. In a simple implementation, a thread waits before each memory access, which corresponds to one wait operation per loop iteration of the execution environment. For a possible implementation of an execution environment, please refer to Section IV.

Correctness In order to use IRS for program verification, it is necessary to ensure that an IRS execution environment permits only correct traces, i.e., traces that show correct program behavior. For Algorithm 1, it is clear that only executions that adhere to a admissible trace may occur by the definition of $admissible()$.

```
1 %20 = call i32 @getThreadId(%"class.indexer::WorkUnit"* %this)
2 %21 = alloca i32
3 store i32 %20, i32* %21
4 %22 = load i32, i32* %21
5 %23 = bitcast i32* %17 to i8*
6 call void @before_memory_access(i32 %22, i8* %23, i64 4, i32 1)
7 %24 = cmpxchg i32* %17, i32 0, i32 %19 seq_cst seq_cst
8 call void @after_memory_access(i32 %22)
```

Listing 1: A global memory access (**cmpxchg**) after inserting callbacks directly before and after.

Progress Besides correctness, progress is required in order to safely use a program inside an IRS execution environment with the same functionality as the unmodified program. Progress for an IRS execution environment expresses that as long as a program has not yet terminated, there exists a thread that can be scheduled next in coherence with a admissible trace, i.e., $admissible(V, s_0, u)$ is not empty. In other words, progress means that IRS does not introduce additional deadlocks into the program. Algorithm 1 provides progress, as only complete traces are added by the verifier.

IV. Implementation

We have implemented IRS in a C++ prototype that uses the LLVM compiler infrastructure [10] to automatically instrument LLVM-IR code and enforces a set of admissible traces when executing the program. This design allows to use IRS for programs that can be translated to LLVM-IR, e.g., C or C++ programs. After instrumenting a program and linking to our IRS library, the program can be safely used (provided that at least one correct schedule is known) without modifying the operating system or any other parameters of the environment. We do not see fundamental obstacles to implement IRS differently, e.g., inside a Java virtual machine (for programs that can be translated to Java bytecode) or with a customized scheduler inside the operating system.

Our implementation consists of an LLVM pass responsible for instrumentation and a library that enforces specified traces in instrumented code. The instrumentation inserts a callback to the library directly before and after memory access instructions (load, store, compare-and-swap). Only those memory accesses are instrumented that directly access a global variable or where the address of the access depends (possibly transitively) on the value of a global variable. Hence, the library sees thread executions as a sequence of events that contain exactly one global memory access. We consider two events dependent if they access the same memory location and at least one of them is a write operation.

Listing 1 shows the global memory access of the Indexer benchmark [4] and how callbacks are inserted. Identifiers have been renamed for easier readability. Only line 7 (containing the compare-and-swap instruction **cmpxchg**) is contained in the original program. All additional lines are added by our instrumentation. Before the memory access, thread ID, memory location and whether the access can modify the memory are reported by callback before_memory_access to the library, where the event is recorded. After the memory access, callback after_memory_access signals that the memory access is completed. At the beginning of the program, an additional "scheduler" thread is started, which collects recorded events and decides whether an event is currently admissible.

When a program thread enters the callback function before a memory access, it checks whether it is necessary to wait for an other thread in order to follow the set of admissible traces. Only if this is the case, synchronization with the scheduler thread is necessary. The program thread appends its current event to a queue of requests and waits on a C++ condition variable. Once the scheduler thread reads the request and the corresponding event is admissible, the program thread is signaled and continues by locking the memory location of the current access, performing the memory operation, and recording the executed event.

When testing our implementation, we found that as expected, locks and condition variables are responsible for a large portion of execution time overhead. In order to reduce the number of locks, we introduced busy-waiting in the scheduler thread, which made synchronization between program threads and the scheduler thread faster for most cases. However, in some cases, synchronization may also be much slower, which may be a disadvantage if execution time should never exceed a tight maximum. We expect further improvement by the use of more advanced lock-less synchronization.

Alternative implementation approaches that do not use an additional scheduler thread are well conceivable and we expect important insights from comparing different implementation approaches. For example, it might be overall faster to perform scheduling tasks locally in program threads instead of the scheduler thread. Even if this duplicates work, execution time might be improved by omitting synchronization.

V. Experimental Evaluation

We concentrate our experimental evaluation on supporting the claim that iteratively relaxing scheduling constraints decreases execution time overhead. Experimentally validating this claim would show that it is feasible to use IRS to adjust and find a sweet spot between verification delay and execution time overhead. As development of our prototype is only in an early stage, we do not provide a full experimental evaluation but report preliminary results for two benchmark programs Indexer [4] and Last Zero [6] that are used to evaluate POR algorithms. These programs have been chosen because model-checking them with POR is well-studied. We use the Last Zero benchmark with 15 worker threads, for which Abdulla et al. report 147456 traces and $1813s$ execution time for POR. For Indexer, we use 15 threads, where Abdulla et al. report 4096 traces and $3155s$ execution time for POR [6].

Table I shows our experimental results. Each benchmark is run without instrumentation (plain) and instrumented by our prototype (IRS). The number of admissible traces is gradually increased. Each configuration is run 1000 times. We report the median execution time and execution time overhead in comparison to the unmodified benchmark. Illustrating the interplay of execution time overhead and verification delay, we show the linearly interpolated verification delay for verification times given by Abdulla et al. as an approximation of how long a model checker would need to verify the corresponding number of traces.

Table I: Execution time overhead of IRS and verification delay

Benchmark	#Traces	Time (μs)	Overhead	Delay (s) *(interpolated)*
Last Zero (IRS)	1	623	230%	0
Last Zero (IRS)	128	506	168%	2
Last Zero (IRS)	1024	486	157%	13
Last Zero (IRS)	16384	402	113%	201
Last Zero (plain)	147456	189	0%	1813
Indexer (IRS)	1	2614	1321%	1
Indexer (IRS)	16	2352	1178%	12
Indexer (IRS)	256	859	367%	197
Indexer (IRS)	2048	761	314%	1578
Indexer (plain)	4096	184	0%	3155

For both benchmarks, execution time overhead can be reduced considerably by permitting only a small portion of all traces. For Last Zero, permitting less than 1% of all traces reduces execution time overhead from 230% to 168%. For Indexer, permitting 6% of all traces reduces execution time overhead from 1321% to 367%. However, execution time overhead is reduced less drastically when additional traces are permitted. Nevertheless, execution time overhead with only few admissible traces is competitive with the execution time overhead of up to about 700% and 300% reported for CoreDet and Dthreads [7].

VI. Related Work

Approaches that attempt to limit the amount of non-determinism in the behavior of multi-threaded programs perform deterministic multi-threading (DMT) and may be implemented using runtime systems [8], [11], [14], libraries [7], and OS modifications [15]. Liu et al. present Dthreads [7] as a replacement for the multi-threading library Pthreads. For a given input, Dthreads forces a deterministic execution by allowing threads to execute in parallel, updating separate copies of the shared state. The separate copies are then merged back in a deterministic order once a synchronization point is reached. In [11], Cui et al. propose to enforce deterministic schedules by recording initial executions and reusing the executed schedule subsequently on compatible executions with similar inputs. Another deterministic approach is Parrot [8], which combines a runtime environment with constrained scheduling with a model checker for bug-finding. Only performance-critical parts of the program need to be model-checked that are manually excluded from deterministic scheduling.

The main difference between these DMT approaches and IRS is that the former support concurrency *testing*, while our approach is suitable for program *verification* by supporting symbolic program inputs. Using existing DMT approaches for verification seems unrealistic as their scheduling constraints depend on concrete program inputs, which would require to verify all possible inputs separately. Another limitation of above described DMT approaches are fixed scheduling constraints that cannot be relaxed at runtime. In contrast, our approach allows to automatically and iteratively relax scheduling constraints at runtime, eventually leading to all schedules (that are successfully verified) being permitted. Additionally, these DMT approaches either provide no fairness in scheduling (completely deterministic execution) or provide fairness for parts of the program by completely unconstrained scheduling. IRS, in contrast, may provide controlled fairness by enabling corresponding schedules and enabling them.

Another line of work deals with limiting the number of context switches to facilitate concurrency bug detection and concurrency testing. In [16] and [17], programs are model-checked using bounded model checking (BMC) where the SMT formula is further constrained to allow only a certain number of context switches. By doing so, they reduce the number of program executions that the SAT/SMT solver has to consider. Musuvathi et al. [13] take this further by using an iterative approach to context switch bounding (ICB). They start with an initial number of context switches, and iteratively allow more to gain a higher confidence in the correctness of the program. In this approach, the model checker implements an explicit state space exploration strategy that systematically explores all possible executions as long as a number of context switches is not exceeded. While these techniques only deal with bug finding and concurrency testing, IRS proposes a complete verification deployment solution. Nevertheless, we expect that the concept of ICB can be applied to symbolic model checking as well and therefore could be used as a verification technique under IRS. To our knowledge, ICB has not yet been applied to symbolic model checking.

In addition to scheduling, a source of non-determinism are relaxed memory models in modern architectures. Relaxed memory models allow more feasible orderings than the more restricted sequential consistency model (SC) leading to more program behavior that has to be covered by the model checker. In [18], a memory monitoring approach is proposed to make sure that SC is maintained during the execution of a program. Fang et al. in [19] present an automated memory fence insertion technique to enforce SC using instrumentation at the source code level. In both cases, the program can be safely verified under the assumption that SC holds with a reduced state space. Similarly to IRS, these approaches restrict the amount of non-determinism. However, in contrast to IRS, they are not able to dynamically adapt the amount of non-determinism and are restricted to non-determinism due to relaxed memory access.

VII. Conclusion

We propose a formal framework for iteratively relaxed scheduling (IRS) as a method to make both verification delay and the amount of non-determinism in scheduling of concurrent programs *adjustable*. By enforcing scheduling constraints, multi-threaded programs can be safely used even if the program has only partially been verified. We outline several scenarios of how to use IRS to enable verification of programs with intractably-large state spaces, enable safe deployment of programs with erroneous schedules, handle verification within a given time budget, manage execution time overhead, and increase execution time performance after deployment. Our preliminary experimental results suggest that iteratively relaxing scheduling constraints gradually reduces execution time overhead.

Acknowledgments

We thank the anonymous reviewers for their constructive comments. Research supported, in part, by H2020-644579 (ESCUDO-CLOUD).

References

[1] A. Valmari, "The state explosion problem," in *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets*, ser. LNCS, vol. 1491. Springer, 1996.

[2] E. M. Clarke, O. Grumberg, M. Minea, and D. Peled, "State space reduction using partial order techniques," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 2, no. 3, 1999.

[3] P. Godefroid, *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, ser. LNCS. Springer, 1996, vol. 1032.

[4] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," in *Symposium on Principles of Programming Languages (POPL)*. ACM, 2005.

[5] G. Gueta, C. Flanagan, E. Yahav, and M. Sagiv, "Cartesian partial-order reduction," in *International SPIN Workshop*, ser. LNCS, vol. 4595. Springer, 2007.

[6] P. A. Abdulla, S. Aronis, B. Jonsson, and K. F. Sagonas, "Optimal dynamic partial order reduction," in *Symposium on Principles of Programming Languages (POPL)*. ACM, 2014.

[7] T. Liu, C. Curtsinger, and E. D. Berger, "Dthreads: efficient deterministic multithreading," in *Symposium on Operating Systems Principles (SOSP)*. ACM, 2011.

[8] H. Cui, J. Simsa, Y. Lin, H. Li, B. Blum, X. Xu, J. Yang, G. A. Gibson, and R. E. Bryant, "Parrot: a practical runtime for deterministic, stable, and reliable threads," in *Symposium on Operating Systems Principles (SOSP)*. ACM, 2013.

[9] A. W. Mazurkiewicz, "Trace theory," in *Advances in Petri Nets*, 1986.

[10] "The LLVM compiler infrastructure," http://llvm.org.

[11] H. Cui, J. Wu, J. Gallagher, H. Guo, and J. Yang, "Efficient deterministic multithreading through schedule relaxation," in *Symposium on Operating Systems Principles (SOSP)*. ACM, 2011.

[12] B. Wachter, D. Kroening, and J. Ouaknine, "Verifying multi-threaded software with impact," in *Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 2013.

[13] M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," in *Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2007.

[14] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman, "Coredet: a compiler and runtime system for deterministic multithreaded execution," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2010.

[15] A. Aviram, S. Weng, S. Hu, and B. Ford, "Efficient system-enforced deterministic parallelism," in *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2010.

[16] I. Rabinovitz and O. Grumberg, "Bounded model checking of concurrent programs," in *International Conference Computer Aided Verification (CAV)*, ser. LNCS, vol. 3576. Springer, 2005.

[17] L. C. Cordeiro and B. Fischer, "Verifying multi-threaded software using smt-based context-bounded model checking," in *International Conference on Software Engineering (ICSE)*, 2011.

[18] S. Burckhardt and M. Musuvathi, "Effective program verification for relaxed memory models," in *International Conference Computer Aided Verification (CAV)*, ser. LNCS, vol. 5123. Springer, 2008.

[19] X. Fang, J. Lee, and S. P. Midkiff, "Automatic fence insertion for shared memory multiprocessing," in *International Conference on Supercomputing (ICS)*. ACM, 2003.