Verification of Parametric System Designs

Alessandro Cimatti, Iman Narasamdya, and Marco Roveri
Fondazione Bruno Kessler, Italy.
Via Sommarive 18, I-38050, Trento, Italy
{cimatti,narasamdya,roveri}@fbk.eu

***Abstract*—System designs are often modeled as sets of threads whose activations are controlled by a domain-specific scheduler. Especially in the early design phases, the interactions between the threads and the scheduler often depend on parameters (such as the duration of thread suspensions) for which a value is not available.**

In this paper, we tackle the verification of designs with parametric scheduler-thread interaction. We propose a new method, called Semi-Symbolic Scheduler/Symbolic Threads (S3ST), to prove that a design satisfies the specified assertions for all possible values of the interaction parameters. We build on Explicit-Scheduler/Symbolic-Threads (ESST), an effective technique for verifying designs with cooperative scheduling, that is however limited to the case of non-parametric interactions. As in ESST, S3ST analyzes each thread symbolically using lazy predicate abstraction. The key difference is in the way the scheduler is dealt with. In ESST, the scheduler is directly executed, using techniques similar to explicit-state model checking. In S3ST, the scheduler is analyzed by combining concrete execution of parts of its state, with the evolution of a symbolically represented set of configurations of interaction parameters.

We have implemented S3ST in the KRATOS software model checker, and have performed an experimental evaluation on a significant set of benchmarks with parametric scheduler-thread interaction. The results clearly demonstrate the effectiveness of the new approach.

I. Introduction

System designs, in many embedded-system settings, are becoming software. Such designs are amenable for high-speed simulations before synthesizing the hardware description. The software typically consists of a set of threads that are activated by a scheduler that implements a set of domain-specific rules. Particularly relevant are multi-threaded software with *cooperative* (or *non-preemptive*) scheduling policy: a thread executes, without any interruption, until it either terminates or explicitly yields the control to the scheduler.

Especially in the early stages of the development process, system designs feature parametric interactions between threads and scheduler. For example, when a thread suspends itself, it does so by calling a suitable scheduler primitive, specifying the duration of the suspension as argument. However, such duration is not necessarily a known numerical constant, but may be modeled in the design by a parameter for which the designer has not selected a value. Thus, the verification process must show that the required properties hold for all subsequent choices of values to the interaction parameters. We also remark that the semantics of some important high-level design languages (e.g. SystemC [1]) allows the parameters of the scheduler-thread interaction primitives to range over reals, thus resulting in timed traces. Restricting the parameters to range over the integers (representing numbers of cycles of fixed duration) is only an approximation.

The problem of model checking cooperative threads has been recently tackled with *Explicit-Scheduler/Symbolic-Threads* (ESST) [2]). ESST combines explicit state techniques to analyze the scheduler with symbolic techniques, based on the lazy predicate abstraction [3], to analyze the threads. ESST orchestrates the analysis of the threads by the direct execution of the scheduler. The threads communicate with each other through shared variables, and communicate/interact with the scheduler (e.g., querying and updating scheduler states) by calling primitive functions provided by ESST. ESST is not able to verify designs with parametric scheduler-thread interactions. In fact, the ability to directly execute the scheduler during the search follows directly from the assumption that the values for the interaction parameters are statically determined.

In this paper, we overcome the limitation of ESST by proposing a new technique, called *Semi-Symbolic Scheduler/Symbolic Threads* (S3ST), that is able to deal with parametric thread-scheduler interactions.

Similar to ESST, in S3ST the threads are analyzed by means of the lazy predicate abstraction. The key difference is that the scheduler, instead of being explicitly executed, is dealt with in a semi-symbolic manner, by combining concrete execution of parts of its state, with the evolution of a symbolically represented set of configurations of interaction parameters.

The approach is based on the following steps. First, we introduce a symbolic representation of time delays for each event, and further abstract the time delays of event notifications with the relations between the symbolic representations. Such an abstraction is carried out by the symbolic analysis, and is passed to the scheduler when it is run. Second, we enable the scheduler to perform reasoning on the relations between the symbolic representations of the time delays. This reasoning determines which event notifications should be triggered at the earliest future time. This can be reduced to checking the satisfiability modulo theory (SMT) [4] of formulas that symbolically represent sets of possible time delays. Third, we enable symbolic analysis, via the lazy predicate abstraction, on the part of the scheduler that modifies the time delays. The part concerns the phase of the scheduler that accelerates the simulation time. This step is non-trivial, because the scheduler must operate on both concrete and symbolic data.

The introduction of ESST was originally motivated by the attempt to avoid performing the lazy predicate abstraction on the scheduler. In order not to lose the ESST advantages, it

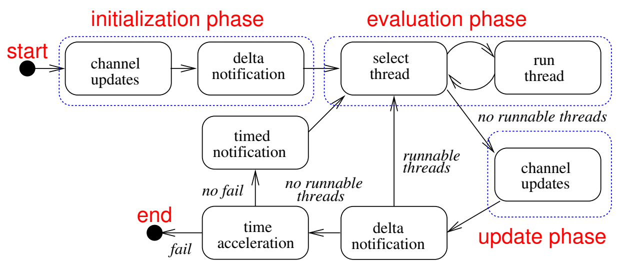

Fig. 1. The SystemC scheduler.

is necessary to control the interactions between the concrete and symbolic data during the scheduler runs. We introduce a technique for predicate filtering, that carefully determines which predicates are relevant to compute the evolution of the scheduler.

In the following we focus on parametric SystemC designs, whose parameters can determine the amount of time delays of event notifications. We have implemented the S3ST algorithm within the KRATOS software model checker [5]. We performed an experimental evaluation on a significant set of new benchmarks and benchmarks adapted from [6], [7] that stress S3ST algorithm. In the experimental evaluation we compare S3ST against the sequentialization approach [2], where the verification problem is reduced to the problem of verifying a sequential program. We also compare S3ST against ESST, by generating non-parametric threaded designs by random sampling the space of parameters. The results of experiments show the effectiveness of S3ST, not only for verification, but also for bug finding.

The paper is organized as follows. Section II provides a background on SystemC and overviews the ESST algorithm. Section III explain the inability of ESST to handle designs with parametric event-notification time delays. Section IV describes the proposed extension to the ESST algorithm. Section V describes some related work. Section VI presents the results of the experimental evaluation. Finally, Section VII concludes this paper and outlines some future work.

II. BACKGROUND

SystemC is a C++ library that consists of a core language for modeling the components of a system design and their interconnections, and a simulation kernel (or a scheduler) for fast simulations of the design. The core language models system components by means of modules (or C++ classes) and abstracts communication between modules by means of channels. SystemC provides several primitive channels such as signal, mutex, semaphore, and queue. A module can have one or more thread definitions that model the parallel behavior of the system design. The core language provides general-purpose events as synchronization mechanisms between threads.

The SystemC scheduler runs the threads during simulations. Following the SystemC semantics in [1], the scheduler consists of several phases (see Figure 1). In the *initialization phase* all channels are initialized. The scheduler then enters the *evaluation phase* where it executes all runnable threads while postponing the materialization of channel updates performed by the threads. This phase employs a cooperative scheduling policy with mutually-exclusive thread execution. When there are no more runnable threads, the scheduler goes into the *update phase* where it materializes all channel updates postponed during the evaluation phase. An evaluation phase followed by an update phase constitutes a *delta cycle*. A thread, during its execution, can perform delayed event notifications. That is, the involved events will be notified at some time in the future, including at the *delta notification*. The materializations of channel updates also often require the events associated with the updated channels to be notified at the delta notifications. In turn, all threads that are waiting for the notified events or are sensitive to the channels whose associated events are notified become runnable. If, after the delta notification, there are runnable threads, the scheduler goes back to the evaluation phase to run them. Otherwise, it accelerates the simulation time to the nearest time point where there exist events to be notified. These events are then notified at the *timed notification*. Similar to the delta notification, some waiting threads can become runnable after the timed notifications, and thus the scheduler has to go back to the evaluation phase to run them. If there are no more events to be notified at some future time, denoted in Figure 1 by failure in time acceleration, then the simulation ends.

SystemC provides several synchronization functions. For example, when a thread calls `wait(`e`)` for an event e, then the thread suspends itself and waits for the notification of e. If another thread calls e`.notify()`, then all threads waiting for the notification of e are made runnable immediately during the current delta cycle. Event notifications can be delayed. If a thread calls e`.notify(`t`)`, for a time t, then e will not be notified immediately. If t is a constant zero, then e will be notified at the delta-notification, otherwise it will be notified after the simulation time accelerates t time units. Similarly, if a thread calls `wait(`t`)`, then it suspends itself and will become runnable at the timed notification after the simulation time accelerates t time units.

Explicit-Scheduler/Symbolic-Threads (ESST) [2], [8] is an effective technique for the verification of *shared-variable* multi-threaded software with *cooperative* scheduling and *mutually-exclusive* thread executions. The threads communicate with each other through shared variables, and communicate with the scheduler (e.g., querying and updating scheduler states) through a set of *primitive functions* provided by ESST.

ESST is a counter-example guided abstraction refinement (CEGAR) [9] technique that combines explicit and symbolic model checking techniques. It analyzes each thread with the lazy predicate abstraction [3], and orchestrates the whole verification by the direct execution of the scheduler using techniques similar to explicit-state model checking. That is, ESST keeps track of the state of the scheduler explicitly, and includes the scheduler as part of the verification algorithm. For the direct execution of the scheduler, ESST needs precise scheduler states, and thus it requires the arguments passed to the primitive function calls to be constants. Both the scheduler and the set of primitive functions are left abstract, but they are required to exhibit a cooperative scheduling policy.

The ESST algorithm is based on the construction and analysis of an *abstract reachability forest* (ARF) that describes the reachable abstract states of the multi-threaded program. An ARF consists of connected abstract reachability trees (ART's), each of which is obtained by unwinding the control-flow graph (CFG) of the running thread. For a program with N threads, an ARF *node* is a tuple $(\langle l_1, \varphi_1\rangle, \ldots, \langle l_N, \varphi_N\rangle, \varphi, \mathbb{S})$, where l_i and φ_i are, respectively, the location and the region of thread i, φ is the global region, and $\mathbb{S}$ is the scheduler state. Regions are formulas describing the values of program variables, while the scheduler state maintains information about the states of the threads as a mapping from scheduler variables to concrete values.

An ARF is constructed by unwinding the CFGs of threads, and by executing the scheduler. Each ART in the ARF is constructed using the lazy predicate abstraction as for the case of sequential programs. In particular, when the operation of the unwound CFG edge involves a call to a primitive function, then ESST has a *primitive executor* that takes as inputs the scheduler state and the call to a primitive function, and returns the updated scheduler state obtained from directly executing the function call.

Given a node $(\langle l_1, \varphi_1\rangle, \ldots, \langle l_i, \varphi_i\rangle, \ldots, \langle l_N, \varphi_N\rangle, \varphi, \mathbb{S})$, such that there are no running threads indicated by $\mathbb{S}$, ESST runs the scheduler on $\mathbb{S}$. The scheduler itself is a function that takes a scheduler state $\mathbb{S}$ (with no running thread) as an input and outputs a set $\{\mathbb{S}'_1, \ldots, \mathbb{S}'_m\}$ of scheduler states representing all possible schedules such that there is only one running thread in $\mathbb{S}'_i$ for $i = 0, \ldots, m$. Each of these states forms an ARF node $(\langle l_1, \varphi_1\rangle, \ldots, \langle l_i, \varphi_i\rangle, \ldots, \langle l_N, \varphi_N\rangle, \varphi, \mathbb{S}'_{\mathrm{j}})$, that becomes the root of a new ART of the subsequent running thread. Coverage checks and refinements in ESST are similar to that of the lazy predicate abstraction. In particular, the subsumption checks are done thread-wise and require the scheduler states to coincide. We refer the reader to [2], [8] for the details on coverage checks and on the ARF refinement techniques.

To verify SystemC designs, we specialize ESST to SystemC by instantiating the ESST scheduler with the SystemC scheduler, and by defining a set of primitive functions that implement the synchronization functions of SystemC. For example, for an event e and a time t, the SystemC synchronization functions `wait(`e`)`, `wait(`t`)`, and e`.notify(`t`)` correspond, respectively, to the primitive functions `wait_event(`e`)`, `wait_time(`t`)`, and `notify_event(`e,t`)`.

III. Parametric Thread-Scheduler Interactions

We focus on parametric designs where the values for the parameters can control the interaction between the threads and the scheduler. In particular we are interested in verifying parametric SystemC designs where the values of parameters determine the amount of delays of event notifications. For example, the design can contain a call `notify_event(`e,t`)` or `wait_time(`t`)` where t is non-constant and its values depend on the value of some parameters. Subsequently, we refer to such a form of design as *SystemC designs with parametric event-notification time delays*.

To verify SystemC designs, ESST maintains information about threads and events in the scheduler state. For each thread T, the domain of the scheduler state includes the scheduler variables st_T and ev_T that keep track of, respectively, the state of T and the event whose notification is awaited by T. The variable st_T ranges over the enumerations $\{\mathit{Waiting}, \mathit{Runnable}, \mathit{Running}\}$, whose meanings are obvious. The variable ev_T ranges over the events in the design and are relevant only when st_T is $\mathit{Waiting}$. For each thread T, we implicitly introduce an event e_T whose notification is awaited by the thread when it suspends itself, e.g., by calling the timed wait function `wait_time(`t`)`, for a time t.

For each event e, the domain of scheduler states includes the scheduler variables st_e and $time_e$ that keep track of, respectively, the state and the notification time delay of e. The variable st_e ranges over the enumerations $\{\mathit{Notified}, \mathit{Delta}, \mathit{Timed}, \mathit{None}\}$, where $\mathit{Notified}$ indicates that the event is notified, Delta and Timed indicate that the event will be notified at, respectively, the delta notification and the timed notification, and None indicates that there is no notification. The value of $time_e$ is a concrete time that ranges over $\mathbb{R}^{\geq 0}$ and is relevant only when st_e is Timed.

The parameters that determine the event notification delays may range over $\mathbb{R}^{\geq 0}$. Such parameters cause state explosion. That is, to keep track of such delays, ESST requires infinitely many scheduler states, which in turn needs infinitely many ARF nodes to represent the reachable abstract states. Thus, ESST cannot handle SystemC designs of our interest.

IV. Semi-Symbolic Scheduler/Symbolic Threads

The proposed technique for verifying SystemC designs with parametric event-notification time delays is called S3ST, for *Semi-Symbolic Scheduler/Symbolic Threads*, and is based on the following ideas. First, the threads are analyzed by means of the lazy predicate abstraction technique, in order to build an ARF. Second, the primitive executor is able to handle calls to primitive functions with non-constant time arguments, by enabling the lazy predicate abstraction on the definitions of primitive functions. Third, the scheduler is modeled in such a way that it can perform reasoning on symbolic data carried by the thread and the global regions of ARF nodes. Similar to the primitive executor, the lazy predicate abstraction is enabled on the part of the scheduler that constrains and modifies the time delays, that is, the delta-notification, the timed-notification, and the time acceleration phases. Finally, we ensure that the ARF construction explores all schedules allowed by the possible combinations of event notifications.

Time-Delay Variables. To overcome the state explosion problem described in Section III, we first introduce, for each event e, a *time-delay variable* ϑ_e as a symbolic representation (or a symbolic value) of all possible time delays for the notification of e. The variable $time_e$ in the scheduler state now ranges over $\mathbb{R}^{\geq 0} \cup \{\vartheta_e\}$. For example, in analyzing a call to `wait_time(`t`)` by a thread T, for a non-constant time t, the primitive executor produces an updated scheduler state that maps st_{e_T} to $Timed$, $time_{e_T}$ to ϑ_{e_T}, st_T to $Waiting$, and ev_T to e_T.

Primitive Function Executor. A key idea of our approach is to abstract the time delays of event notifications by the relations between the time-delay variables. These relations are carried by the regions of the ARF nodes, and are analyzed by the lazy predicate abstraction. To this end, we first need to make the time-delay variables visible to the lazy predicate abstraction. Second, we require the primitive executor to provide the lazy predicate abstraction with part of the definition of the called primitive function that updates the time delays.

Let P be a threaded program with N threads, $T_1, \ldots, T_N$. We denote by $SVar$ the set of shared variables of P, by $LVar_T$ the set of local variables of the thread T in P, and by Var_P the set of all variables in P. We assume that $LVar_T \cap SVar = \emptyset$ for every thread T and $LVar_{T_i} \cap LVar_{T_j} = \emptyset$ for each two different threads T_i and T_j. To make time-delay variables visible to the lazy predicate abstraction, we consider them as being shared variables in P. That is, given a set $\{e_0, \ldots, e_m\}$ of events in P, we have $\{\vartheta_{e_0}, \ldots, \vartheta_{e_m}\} \subseteq SVar$. Besides an updated scheduler state, the primitive executor generates on-the-fly a loop-free program defining the update of the time-delay variable. This program is then analyzed by the lazy predicate abstraction.

Let $SState$ be the set of scheduler states, $PrimCall$ be the set of primitive function calls, and $LFProg_P$ be the set of loop-free programs over the variables in Var_P. For simplicity of presentation, we assume that primitive functions do not return any value. The primitive executor for P in S3ST is the function

$$\text{SEXEC} : (SState \times PrimCall) \to (SState \times LFProg_P)$$

that takes a scheduler state and a primitive function call as input, and outputs an updated scheduler state along with a loop-free program. For example, in executing `wait_time(`exp`)` called by a thread T, for some expression exp, besides outputting an updated scheduler state, as explained before, the primitive executor generates the program

$$\texttt{assume(}exp >= 0\texttt{);}\ \vartheta_{e_T} \texttt{ := } exp.$$

Note that, the time-delay variables are viewed as symbolic values by scheduler states, but as program variables by the lazy predicate abstraction.

The scheduler to determine which events to notify at the delta- and timed-notification needs to know the relations among the time-delay variables of the events with constant and non-constant delays. Thus, to enable lazy predicate abstraction to discover predicates that speak about such relations, even if the time delays are constants, the primitive executor always generates the loop-free program.

Semi-Symbolic Scheduler. The scheduler consists of the phases shown in Figure 1. Particularly, in the delta- and timed-notification the scheduler has to reason about the relations between time-delay variables to determine which events to notify. Due to the parameters that affect the time delays, there can be more than one combination of events that can be notified in those phases. Different combinations can result in the simulation time being accelerated to different earliest future times. The scheduler though must allow for the exploration of all possible combinations. Moreover, similar to the primitive executor, because the time acceleration essentially updates the time delays, the scheduler must generate on-the-fly programs representing the updates of the time-delay variables.

The S3ST scheduler is the function

$$\text{SCHED} : ARFNode \to \mathcal{P}(\mathcal{P}(SState) \times LFProg_P)$$

that takes an ARF node η as an input and returns a set $\{(S_1, P_1^{lf}), \ldots, (S_n, P_n^{lf})\}$ where S_i is a set of scheduler states and P_i^{lf} is a loop-free program. Particularly for SystemC verification, each S_i is a result of notifying a different set of events in the delta- or timed-notification. In what follows we focus on the timed-notification of the scheduler; the delta-notification can be explained in a similar way.

We denote by $\mathbb{S}[x_0 \mapsto v_0, \ldots, x_n \mapsto v_n]$ the scheduler state obtained from a scheduler state $\mathbb{S}$ by replacing the images of x_i in $\mathbb{S}$ with v_i for $i = 0, \ldots, n$. For simplicity of presentation, we assume that the relations over the time-delay variables and over the parameters are tracked by the global regions of the ARF nodes.

The procedure TIMEDNOTIFICATION shown in Algorithm 1 implements the time acceleration and the timed-notification of Figure 1. Let $(\langle l_1, \varphi_1 \rangle, \ldots, \langle l_N, \varphi_N \rangle, \varphi, \mathbb{S})$ be the input ARF node, and let TE be the set of events with $Timed$ state. TIMEDNOTIFICATION checks for every non-empty subset $E = \{e_0, \ldots, e_m\}$ of TE whether the events in E can be notified at the same earliest future time, while delaying further the notifications of others in TE. This is done by analyzing the time-delay variables and their relations carried by the global region φ. The analysis amounts to checking, by the procedure SAT, if the conjunction between φ, the equalities $\vartheta_{e_0} = \cdots = \vartheta_{e_m}$, and the inequalities $\vartheta_{e_0} > 0 \wedge \bigwedge\{\vartheta_{e_o} < \vartheta_{e'} \mid e' \in TE \setminus E\}$ is satisfiable. If it is, then the simulation time is accelerated by the procedure ACCELERATETIME, the events in E are notified, and the threads that are waiting for the notifications of the events in E are woken up by the procedure WAKEUPTHREADS, by changing the threads' states from $Waiting$ to $Runnable$.

Intuitively, the procedure TIMEDNOTIFICATION tries all possible combinations of event notifications. If the satisfiability check of the set E is successful, then it means that all events in E can be notified at the same earliest future time, while postponing the notifications of the other events in TE.

The procedure ACCELERATETIME is shown in Algorithm 2. The first for-loop sets the variable $time_e$ of the event e in TE to ϑ_e if $time_{e_0}$ has the symbolic value ϑ_{e_0}. But, note that, if $time_{e_0}$ and $time_e$ are concrete values, then the time acceleration is the same as in the ESST scheduler, i.e., it simply subtracts the value of $time_{e_0}$ from the value of $time_e$ and sets the result as the new value for $time_e$. The pseudo-code following the first for-loop generates a loop-free program P^{lf} that represents the formula checked by SAT, as well

Algorithm 1: TIMEDNOTIFICATION

```
Input  : An ARF node $(\langle l_1, \varphi_1\rangle, \ldots, \langle l_N, \varphi_N\rangle, \varphi, \mathbb{S})$.
Output: A set $R$ of pairs $(\mathbb{S}', P^{lf})$ of a scheduler state $\mathbb{S}'$ and a
        loop-free program $P^{lf}$.
$R \leftarrow \emptyset$
$TE \leftarrow \{e \mid \mathbb{S}(st_e) = Timed\}$
if $TE \neq \emptyset$ then
    for $E \in \mathcal{P}(TE)$ and $E \neq \emptyset$ do
        Let $E = \{e_0, \ldots, e_m\}$
        $Eq \leftarrow \vartheta_{e_0} = \cdots = \vartheta_{e_m}$
        $InEq \leftarrow \vartheta_{e_o} > 0 \wedge \bigwedge\{\vartheta_{e_o} < \vartheta_{e'} \mid e' \in TE \setminus E\}$
        if SAT$(\varphi \wedge Eq \wedge InEq)$ then
            $(\mathbb{S}', P^{lf}) \leftarrow$ ACCELERATETIME$(\mathbb{S}, E)$
            $\mathbb{S}' \leftarrow \mathbb{S}'[st_{e_0} \mapsto Notified, \ldots, st_{e_m} \mapsto Notified]$
            $\mathbb{S}' \leftarrow$ WAKEUPTHREADS$(\mathbb{S}')$
            $\mathbb{S}' \leftarrow \mathbb{S}'[st_{e_0} \mapsto None, \ldots, st_{e_m} \mapsto None]$
            $R \leftarrow R \bigcup \{(\mathbb{S}', P^{lf})\}$
```

Algorithm 2: ACCELERATETIME

```
Input  : A pair $(\mathbb{S}, E)$ of a scheduler state $\mathbb{S}$ and a non-empty set $E$ of
         to-be-notified events.
Output: A pair $(\mathbb{S}', P^{lf})$ of a scheduler state $\mathbb{S}'$ and a loop-free
        program $P^{lf}$.
Let $E = \{e_0, \ldots, e_m\}$
$\mathbb{S}' \leftarrow \mathbb{S}$
$TE \leftarrow \{e \mid \mathbb{S}(st_e) = Timed\}$
for $e \in TE$ do
    if $\mathbb{S}(time_{e_0}) = \vartheta_{e_0}$ then $\mathbb{S}' \leftarrow \mathbb{S}'[time_e \mapsto \vartheta_e]$
    else
        if $\mathbb{S}(time_e) \neq \vartheta_e$ then
            $t \leftarrow \mathbb{S}(time_e) - \mathbb{S}(time_{e_0})$
            $\mathbb{S}' \leftarrow \mathbb{S}'[time_e \mapsto t]$
$P^{lf} \leftarrow$ "assume($\vartheta_{e_0} > 0$);"
for $e \in TE$ do
    if $e \in E$ then $P^{lf} \leftarrow P^{lf}$ + "assume($\vartheta_e = \vartheta_{e_0}$);"
    else $P^{lf} \leftarrow P^{lf}$ + "assume($\vartheta_e > \vartheta_{e_0}$);"
    $P^{lf} \leftarrow P^{lf}$ + "$\vartheta_e$ := $\vartheta_e$ - $\vartheta_{e_0}$;"
```

as the updates of time-delay variables caused by the time acceleration. For example, if $E = \{e_0\}$ and $TE = \{e_0, e_1\}$, such that $time_{e_0}$ is mapped to the time-delay variable in the input scheduler state, then the generated loop-free program is:

```
assume($\vartheta_{e_0} > 0$);
assume($\vartheta_{e_0} = \vartheta_{e_0}$); $\vartheta_{e_0}$ := $\vartheta_{e_0}$ - $\vartheta_{e_0}$;
assume($\vartheta_{e_1} > \vartheta_{e_0}$); $\vartheta_{e_1}$ := $\vartheta_{e_1}$ - $\vartheta_{e_0}$;
```

The result of TIMEDNOTIFICATION is a set $\{(\mathbb{S}_1, P_1^{lf}), \ldots, (\mathbb{S}_n, P_n^{lf})\}$ of pairs of a scheduler state and a loop-free program. Each scheduler state $\mathbb{S}_i$ has some runnable threads that must be run in the evaluation phase. That is, for each $\mathbb{S}_i$ such that $st_{T_{i_0}}, \ldots, st_{T_{i_m}}$ are mapped to *Runnable*, the scheduler generates a set $S_i = \{\mathbb{S}_i^0, \ldots, \mathbb{S}_i^m\}$ of scheduler states where each $\mathbb{S}_i^j$ is $\mathbb{S}_i[st_{T_{i_j}} \mapsto Running]$. Finally, the scheduler returns the set $\{(S_1, P_1^{lf}), \ldots, (S_n, P_n^{lf})\}$.

ARF Construction. Similar to ESST, the S3ST algorithm is based on the construction of ARF by unwinding the CFGs of threads and by executing the scheduler. Expanding an ARF node involves computing the *abstract strongest post-condition* $SP^{\pi}_{op}(\psi)$ of a region ψ with respect to the operation *op* and the precision π. The operation *op* can be the operation labeling the unwound CFG edge or the loop-free program generated by the primitive executor or by the scheduler. The precision π is a set of predicates that are associated locally with a thread (or a location in the CFG of a thread) or associated with the global region.

We expand an ARF node $\eta = (\langle l_1, \varphi_1\rangle, \ldots, \langle l_N, \varphi_N\rangle, \varphi, \mathbb{S})$ by means of the following rules:

E1. There is a running thread i in $\mathbb{S}$ that performs an operation *op* and (l_i, op, l_i') is an edge of the CFG of thread i:

- If *op* is *not* a call to a primitive function, then let $\hat{op}$ be *op* and $\mathbb{S}' = \mathbb{S}$.
- If *op* is a call to a primitive function, then $(\mathbb{S}', \hat{op}) =$ SEXEC$(\mathbb{S}, op)$.

The successor node is $(\langle l_1, \varphi_1'\rangle, \ldots, \langle l_i', \varphi_i'\rangle, \ldots, \langle l_N, \varphi_N'\rangle, \varphi', \mathbb{S}')$, where

- $\varphi_i' = SP^{\pi^{l_i'}}_{\hat{op}}(\varphi_i \wedge \varphi)$,
- $\varphi_j' = SP^{\pi^{l_j}}_{\text{HAVOC}(\hat{op})}(\varphi_j \wedge \varphi)$ for $j \neq i$, and
- $\varphi' = SP^{\pi}_{\hat{op}}(\varphi)$.

The function HAVOC collects all global variables possibly updated by $\hat{op}$, and builds a new operation where these variables are assigned with fresh variables. The precisions π^l and π are associated with the location l of the corresponding CFG and the global region, respectively.

E2. There is *no* running thread in $\mathbb{S}$. For each $(S, P^{lf}) \in$ SCHED(η) and for each scheduler state $\mathbb{S}' \in S$, we create a successor node $(\langle l_1, \varphi_1'\rangle, \ldots, \langle l_N, \varphi_N'\rangle, \varphi', \mathbb{S}')$, where

- $\varphi_j' = SP^{\pi^{l_j}}_{\text{HAVOC}(P^{lf})}(\varphi_j \wedge \varphi)$, for $j = 1, \ldots, n$, and
- $\varphi' = SP^{\pi}_{P^{lf}}(\varphi)$.

such that the successor node becomes the root node of a new ART added to the ARF.

Note that the strongest post-condition with respect to P^{lf} can always be computed because P^{lf} is a loop-free program.

Similar to ESST, the construction of an ARF in S3ST starts with a single ART representing reachable states of the main thread. In the root node of that ART all regions are initialized with *True*, all thread locations are set to the entries of the corresponding threads, and the only running thread in the scheduler state is the main thread.

The ARF is expanded using the rules E1 and E2. An ARF is *complete* if it is closed under the expansion of those rules. An ARF is *safe* if it is complete and, for every node $(\langle l_1, \varphi_1\rangle, \ldots, \langle l_N, \varphi_N\rangle, \varphi, \mathbb{S})$ in the ARF such that $\varphi \wedge \bigwedge_{i=1,\ldots,n} \varphi_i$ is satisfiable, none of the locations $l_1, \ldots, l_N$ are error locations. If one of the locations $l_1, \ldots, l_N$ is an error location, we build a counter-example consisting of paths in the trees of the ARF and check if the counter-example is feasible. Unlike ESST, in the building of counter-example S3ST has to take into account the generated loop-free programs. If the counter-example is feasible, then we have found a real counter-example witnessing that the program is unsafe. Otherwise, we use it to discover predicates to refine the ARF. Coverage checks and refinements of S3ST are the same as those of ESST. We refer to [8] for further details. Note that, because the updates of the time-delay variables are represented by the on-the-fly generated programs that are analyzed symbolically, the existing refinement methods of ESST can discover predicates that speak about the relations between time-delay variables.

Predicate Filtration. One possible bottleneck in S3ST is there can be too many predicates about the relations between time-delay variables that have to be tracked during the ARF construction. The more predicates to track, the more expensive the computations of abstract strongest post-conditions. To alleviate this problem, we perform a predicate filtration that looks up the scheduler state to filter out predicates that contains "inactive" time-delay variables during the computations of abstract strongest post-conditions.

Let q be a predicate and $\mathbb{S}$ be a scheduler state. Denote by $fvar(q)$ the set of free variables occurring in q and by $\Theta(\mathbb{S})$ the set of time-delay variables such that, for each ϑ_e in $\Theta(\mathbb{S})$, we have $\mathbb{S}(st_e) = None$. Given an ARF node $\eta = (\langle l_1, \varphi_1\rangle, \ldots, \langle l_N, \varphi_N\rangle, \varphi, \mathbb{S})$ to be expanded with an operation $\hat{op}$ such that the successor scheduler state is $\mathbb{S}'$, then instead of computing the successor global region φ' as $SP^{\pi}_{\hat{op}}(\varphi)$, we compute φ' as $SP^{\pi'}_{\hat{op}}(\varphi)$, where $\pi' = \pi \setminus \{q \in \pi \mid fvar(q) \cap \Theta(\mathbb{S}') \neq \emptyset\}$. The successor thread regions can be computed similarly.

Partial-Order Reduction (POR). POR [10] alleviates the problem of exploring a large number of redundant thread interleavings by exploiting the commutativity of concurrent transitions that result in the same state when they are executed in different orders. The POR techniques developed for ESST in [6] are applicable to S3ST. In [6] we have the procedure PERSISTENT that implements the persistent-set technique. The procedure takes as inputs an ARF node η and a set S of scheduler states resulting from a scheduler run, and outputs a subset of S. For S3ST, we simply run PERSISTENT(η, S_i) for each S_i in $\{(S_1, P_1^{lf}), \ldots, (S_n, P_n^{lf})\} = $ SCHED(η).

We remark that, the S3ST approach is not a form of POR, particularly because TIMEDNOTIFICATION explores all possible combinations of event notifications. Indeed we can optimize TIMEDNOTIFICATION by techniques inspired by POR. Suppose that we can partition the set of threads in the system design such that in each partition the variables accessed by the threads and the events notified and waited by the threads are disjoint from those of other partitions. Such a partitioning is often possible on a system design that consists of components that do not interact with each other. Given partitions of threads, if a subset E' of E of events to be notified by TIMEDNOTIFICATION wake up threads in partitions different from those woken up by the events in $E \setminus E'$, and the notifications of events in E' can be delayed, then we do not explore the possibility to notify E' together with $E \setminus E'$, but only explore the case where the notification of E' is further delayed.

Correctness. Let S3ST$_{SC}$ be the specialization of S3ST to SystemC, as explained above. In what follows, we assume to work on a threaded program P (representing a SystemC design) with N threads $T_1, \ldots, T_N$. Following the programming framework in [8], a *configuration* γ of P is a tuple $\langle \gamma_{T_1}, \ldots, \gamma_{T_N}, gs, \mathbb{S}\rangle$ where (1) each $\gamma_{T_i} = (l_i, s_i)$ is a thread local configurations, where l_i is a program location and s_i is a mapping (or state) from $LVar_{T_i}$ to values, (2) gs is a mapping (or state) from $SVar$ to values, and (3) $\mathbb{S}$ is a scheduler state. Given a configuration γ and an expression e consisting of variables in $SVar$ and $LVar_{T_i}$ (for $i = 1, \ldots, N$), we denote by $\gamma(e)$ the value resulting from the evaluation of e over γ. The evaluation can be extended naturally to the case of multiple expressions as arguments. For a configuration γ with $\mathbb{S}$ as its scheduler state, we denote by $\gamma[\mathbb{S}'/\mathbb{S}]$ the configuration obtained from γ by replacing $\mathbb{S}$ with a scheduler state $\mathbb{S}'$. Given a state s, we denote by $Dom(s)$ the domain of s. For two states s_1, s_2 with disjoint domains, we denote by $s_1 \cup s_2$ the union s_1 and s_2 such that, for every $x \in Dom(s_1 \cup s_2)$, we have $(s_1 \cup s_2)(x) = s_1(x)$ if $x \in Dom(s_1)$, otherwise $(s_1 \cup s_2)(x) = s_2(x)$. Let φ be a formula over variables in the domain of a state s, we denote by $s \models \varphi$ for a state s satisfying φ.

Let $\eta = (\langle l_1, \varphi_1\rangle, \ldots, \langle l_N, \varphi_N\rangle, \varphi, \mathbb{S})$ be an ARF node. We denote by $eq(\mathbb{S})$ the conjunctions of equalities induced by the mappings in $\mathbb{S}$. We say that the configuration $\gamma = \langle (l'_1, s_1), \ldots, (l'_N, s_N), gs, \mathbb{S}'\rangle$ satisfies the node η, denoted by $\gamma \models \eta$, if for all $i = 1, \ldots, N$, we have $l_i = l'_i$, $s_i \cup gs \models \varphi_i$, $\bigcup_{i=1,\ldots,N} s_i \cup gs \models \varphi$, and $\bigcup_{i=1,\ldots,N} s_i \cup gs \cup \mathbb{S}' \models \varphi \wedge eq(\mathbb{S})$.

Following [8], the semantics of each n-ary primitive function f is defined by an $n+1$-ary function $\hat{f}$ that takes as input, in addition to the arguments of f, a scheduler state $\mathbb{S}$ and returns an updated scheduler state $\mathbb{S}'$. For the correctness of S3ST$_{SC}$, we assume that the primitive executor SEXEC implements correctly the definitions of primitive functions. Let η, η' be ARF nodes such that η' is obtained from η by applying rule E1, where the operation op is a call $f(\vec{e})$ to primitive function f with expressions $\vec{e}$ as the arguments. Then, for configurations γ, γ' such that $\gamma \models \eta$, $\mathbb{S}$ is the scheduler state of γ, and $\gamma' = \gamma[\hat{f}(\gamma(\vec{e}), \mathbb{S})/\mathbb{S}]$, we have $\gamma' \models \eta'$.

In what follows, we show that the scheduler SCHED of S3ST$_{SC}$ explores all possible combinations of event notifications. First, let η be an ARF node such that there is no running thread indicated by its scheduler state. Let $(S, P^{lf}) \in$ SCHED(η) and $\mathbb{S} \in S$, we denote by $\eta_{S,\mathbb{S}}$ the successor node obtained from S and $\mathbb{S}$ by rule E2. Second, the SystemC scheduler, as in [8], can be implemented by a function *Sched* that takes a scheduler state as an input and outputs a set of scheduler states.

Lemma 1: Let η be an ARF node such that there are no running threads in its scheduler state, and let γ be a configuration such that $\gamma \models \eta$ and $\mathbb{S}$ is γ's scheduler state. Let $\hat{S} = Sched(\gamma)$ be the set of scheduler states obtained by running the scheduler. Then, there are a pair $(S, P^{lf}) \in$ SCHED(η) and a one-to-one correspondence C between $\hat{S}$ and S such that, for every scheduler state $\mathbb{S}' \in \hat{S}$, we have $\gamma[\mathbb{S}'/\mathbb{S}] \models \eta_{S,C(\mathbb{S})}$.

Proof: (Sketch) The proof of this lemma relies on the fact that TIMEDNOTIFICATION of SCHED enumerates all possible combinations of event notifications yielded by configurations that satisfy the ARF node η. ■

Intuitively, the above lemma says that, for any configuration that satisfies the ARF node η, the successor configuration obtained by running the scheduler $Sched$ is in the set of configurations represented by the successor abstract state obtained by running S3ST$_{SC}$'s scheduler SCHED.

The following theorem states the correctness of S3ST$_{SC}$:

Theorem 1: Let P be a SystemC design with parametric event-notification time delays. For every terminating execution, S3ST$_{SC}(P)$ returns a safe ARF if and only if P is safe for all possible values of its parameters.

Proof: (Sketch) The proof can be derived from that of ESST in [8]. The correctness of S3ST$_{SC}$ relies on the abovementioned assumption about the primitive executor SEXEC and Lemma 1. In particular the computations of abstract strongest post-condition on the on-the-fly generated loop-free programs over-approximate the set of possible values for the time-delay variable ϑ_e of an event e when st_e is $Timed$. ■

V. Related Work

There has been a large amount of work on developing techniques for the verification of both sequential and concurrent (or multi-threaded) programs; see [11] and the related work section of [8] for recent surveys. Most of these techniques do not address timed systems, and assume to deal only with a simple non-deterministic scheduler.

Sequentialization. One popular approach to verifying multi-threaded programs is by means of sequentialization. In this approach the multi-threaded program is translated into a (non-deterministic) sequential program that is behaviorally equivalent, or equivalent up to some bounds (e.g., number of context switches), to the multi-threaded program. The resulting program is then analyzed by off-the-shelf techniques for sequential programs. Our previous work in [2] on sequentializing SystemC designs is already able to handle SystemC designs with parametric event-notification delays because the sequentialization captures the precise semantics of the SystemC scheduler. Indeed, the sequentialization approach can be used to verify general parametric SystemC designs. However, as demonstrated in that paper, the approach does not scale up to large designs.

The work in [12] is concerned with the verification of safety properties of periodic real-time systems with priority-sensitive scheduling. The verification is based on the translation of the system into a sequential program that over-approximates all executions of the system up to some time bound. The resulting sequential program is then verified using bounded model checking (BMC). Similar to our work, the work abstracts time via job-bounded abstraction. However, due to being over-approximations, the analysis of the sequential programs can result in false warnings.

Timed and Hybrid Systems. Other branch of work on the analysis of timed systems is in the context of timed and hybrid systems/automata [13], [14]. The analysis mostly abstracts away data variables, and particularly for timed automata, the analysis cannot handle non-deterministic inputs. Notable exceptions are the SMT-based verification of timed and hybrid automata in [15], [16]. The work in [15] reduces schedulability analysis of parametric timed automata to reachability of error location in the symbolic representation (SMT formulas) of the automata. The reachability analysis is done via BMC and is complete only for periodic systems. The analysis involves neither abstraction nor refinement processes. The work in [16] is concerned with the scenario verification of hybrid systems. Similar to [15], the hybrid systems are represented symbolically as SMT formulas and analyzed by means of BMC.

Path Exploration and Test Case Generation. Techniques that involve mixed symbolic and concrete executions have also been developed in the context of automated path exploration and test cases generations. Popular approaches have been implemented in DART (Directed Automated Random Testing) [17], EXE [18], SPF (Symbolic PathFinder) [19], [20], and S2E [21] DART performs bounded concrete executions on random inputs, while at the same time collects the path constraints of the executed paths. The constraints are then systematically negated to obtain new input values that will direct the next concrete executions to alternative paths. These steps are repeated until the coverage criteria is achieved. EXE and SPF essentially perform symbolic executions, but perform concrete executions to simplify the path constraints. S2E interleaves concrete and symbolic executions. On switching from concrete execution to symbolic one, S2E generalizes the concrete values to symbolic values, and run simultaneously concrete and symbolic executions. On switching in the reverse direction, S2E performs lazy concretization by on-demand instantiations of symbolic data.

VI. Experimental Evaluation

We have implemented the S3ST algorithm, and its specialization to SystemC, in the KRATOS software model checker [5].

Setup. We have carried out an experimental evaluation using new benchmarks and benchmarks derived from [6] and [7]. The derived benchmarks generalize the original ones by adding parameters that control the time delays of event notifications. The number of added parameters corresponds to the number of primitive function calls that concern event notifications (which is linear with the number of threads). For each benchmark x from [6] and [7], we call the derived benchmark p-x. The benchmarks that exhibit thread-scheduler interaction delays that may vary from cycle to cycle are marked with a $\star$.

We compared the S3ST algorithm with the sequentialization approach described in [2]. For the experiments with S3ST, we enabled partial-order reduction. For the sequentialization, we experimented with the lazy predicate abstraction of KRATOS and CPACHECKER SVN revision 6080 [22], the eager abstraction of SATABS-3.0 [23], and the BMC of CBMC-4.0 [24]. For CBMC, we set the number of loop unwinding to 3 and only considered the unsafe benchmarks.

We ran our experiments on an Intel Xeon 3GHz box with 4GB of RAM, and running Linux. We set the time limit to 1000s and the memory limit to 2GB.

Data to reproduce our experiments is available at http://es.fbk.eu/people/roveri/tests/fmcad2012.

Results. Table I shows the results of experiments. The column V shows the status of the benchmarks: S for safe and U for unsafe. For each tool we report the execution time in seconds. We use T.O for out of time, M.O for out of memory, U.R for returning unknown, E.R for having run time errors, and N.A for not available. For S3ST, we performed experiments with and without predicate filtration (resp. columns PF and No-PF).

In general, it is clear that S3ST outperforms the sequentialization techniques. For the sequentialization approach, a close inspection on KRATOS reveals that, even for the small `p-token-ring.2` benchmark, the analysis has to keep track of 45 predicates. For CBMC, the * mark on the results indicate that, due to insufficient loop unwindings, CBMC reports that the benchmarks are safe. Any attempt to increase the number of loop unwindings results in out of time. We also see that the impact of the predicate filtration is very significant for the scalability of S3ST. For example, for `p-token-ring.4` benchmark the predicate filtration, on average, can filter out 44% of predicates used in the abstraction computations. Finally, we notice that the $\star$ benchmarks, featuring cycle-varying parameters, are even harder for sequentialization.

The following table shows the behavior of S3ST when the number of parameters in the benchmarks is increased. We present the results for the `p-token-ring.10` and the `p-toy` benchmarks that have, respectively, 11 and 3 parameters. (Other benchmarks show a similar behavior.)

	p-token-ring.10									p-toy			
#Parameters	0	1	2	3	4	5	6	7	8	0	1	2	3
Run Time	1.6	23.6	27.6	33.4	42.2	61.8	289.5	448.2	743.0	2.5	4.7	99.7	99.8
# ARF Nodes	1378	2513	3422	4673	5941	9324	21110	24558	28018	673	787	4245	4233
#Preds	23	54	56	60	66	74	84	96	110	23	27	55	54

For the `p-token-ring.10` table, the column j shows the experiment on a benchmark obtained from `p-token-ring.10` by concretizing $11 - j$ parameters with some constants. Similarly for the `p-toy` table. The presence of parameters potentially increases the number of thread interleavings that S3ST has to explore, as shown by the number of visited ARF nodes. For the experiments reported in the `p-token-ring.10` table, the predicate filtration is effective in reducing the number of predicates that concern the relations of the time-delay variables: on average, 41.46% reduction. However, S3ST still has to keep track of the predicates that concern the relations between the constraints over the parameters themselves. The more parameters, the more predicates it has to track, as indicated in the row #Pred. Analyzing benchmarks containing both constant and parametric time delays can be as hard as analyzing those containing only parametric time delays. Recall that the scheduler uses the relations between the time-delay variables to determine the events to notify. Thus, even though the time delay of the notification of an event is a constant, S3ST may still have to keep track of predicates containing the time-delay variable associated with that event.

We have also investigated the possibility of analyzing with ESST the benchmarks obtained by grounding the time delay parameters with a number of (random) values. For

TABLE I
RESULTS OF EXPERIMENTAL EVALUATION (IN SEC).

		S3ST		Sequentialization			
Name	V	PF	No-PF	KRATOS	CPA	SATABS	CBMC
p-kundu-bug-1	U	**1.18**	1.19	23.18	U.R	375.04	5.26
p-kundu-bug-2	U	**0.87**	0.89	44.54	U.R	T.O	22.04
p-kundu	S	**54.62**	62.66	T.O	U.R	T.O	N.A
p-mem-slave-tlm.1	S	**10.07**	38.87	T.O	E.R	531.79	N.A
p-mem-slave-tlm.2	S	**54.16**	T.O	T.O	M.O	878.71	N.A
p-mem-slave-tlm.3	S	**185.95**	T.O	T.O	M.O	T.O	N.A
p-mem-slave-tlm.4	S	**517.00**	T.O	T.O	M.O	T.O	N.A
p-mem-slave-tlm.5	-	T.O	T.O	T.O	E.R	T.O	N.A
p-mem-slave-tlm-bug.1	U	**6.65**	25.18	T.O	M.O	T.O	*306.33
p-mem-slave-tlm-bug.2	U	**35.64**	882.55	T.O	M.O	T.O	*286.46
p-mem-slave-tlm-bug.3	U	**106.80**	T.O	T.O	M.O	T.O	*278.67
p-mem-slave-tlm-bug.4	U	**402.51**	T.O	T.O	M.O	T.O	*293.06
p-mem-slave-tlm-bug.5	U	**991.57**	T.O	T.O	M.O	T.O	*323.01
p-mem-slave-tlm-bug2.1	U	**4.23**	4.79	T.O	M.O	T.O	*295.68
p-mem-slave-tlm-bug2.2	U	**15.48**	17.58	T.O	M.O	T.O	*295.17
p-mem-slave-tlm-bug2.3	U	**43.17**	45.87	T.O	M.O	T.O	*283.85
p-mem-slave-tlm-bug2.4	U	**99.73**	104.42	T.O	M.O	T.O	*306.81
p-mem-slave-tlm-bug2.5	U	**236.81**	244.65	T.O	M.O	T.O	*336.84
p-pc-sfifo-1	S	**3.45**	4.29	T.O	U.R	197.29	N.A
p-pc-sfifo-2	S	**3.49**	4.00	239.01	U.R	193.05	N.A
p-token-ring.1	S	**0.56**	0.59	20.97	83.22	904.94	N.A
p-token-ring.2	S	**1.49**	2.09	T.O	M.O	T.O	N.A
p-token-ring.3	S	**3.49**	13.48	T.O	M.O	T.O	N.A
p-token-ring.4	S	**8.08**	430.58	T.O	M.O	T.O	N.A
p-token-ring.5	S	**14.73**	T.O	T.O	M.O	T.O	N.A
p-token-ring.6	S	**27.84**	T.O	T.O	M.O	T.O	N.A
p-token-ring.7	S	**70.53**	T.O	T.O	M.O	T.O	N.A
p-token-ring.8	S	**192.87**	T.O	T.O	E.R	T.O	N.A
p-token-ring.9	S	**789.14**	T.O	T.O	M.O	T.O	N.A
p-token-ring.10	-	T.O	T.O	T.O	M.O	T.O	N.A
p-token-ring-bug.1	U	**0.47**	0.49	14.73	20.59	485.92	6.47
p-token-ring-bug.2	U	**1.18**	1.29	T.O	E.R	773.84	16.62
p-token-ring-bug.3	U	**2.49**	5.09	T.O	M.O	T.O	*33.83
p-token-ring-bug.4	U	**5.68**	95.05	T.O	M.O	T.O	*91.53
p-token-ring-bug.5	U	**9.98**	T.O	T.O	E.R	T.O	*154.57
p-token-ring-bug.6	U	**17.76**	T.O	T.O	M.O	T.O	*250.71
p-token-ring-bug.7	U	**45.55**	T.O	T.O	M.O	T.O	*478.11
p-token-ring-bug.8	U	**106.39**	T.O	T.O	M.O	T.O	*752.50
p-token-ring-bug.9	U	**537.08**	T.O	T.O	M.O	T.O	T.O
p-token-ring-bug.10	U	T.O	T.O	T.O	M.O	T.O	T.O
p-token-ring-bug2.1	U	**0.48**	0.49	14.05	15.51	465.44	6.55
p-token-ring-bug2.2	U	**1.39**	1.59	T.O	M.O	740.50	20.55
p-token-ring-bug2.3	U	**3.56**	6.49	T.O	E.R	T.O	*46.87
p-token-ring-bug2.4	U	**8.76**	103.04	T.O	M.O	T.O	*81.79
p-token-ring-bug2.5	U	**24.66**	T.O	T.O	M.O	T.O	*165.14
p-token-ring-bug2.6	U	**47.55**	T.O	T.O	M.O	T.O	*310.16
p-token-ring-bug2.7	U	**100.49**	T.O	T.O	M.O	T.O	*499.32
p-token-ring-bug2.8	U	**372.45**	T.O	T.O	M.O	T.O	*748.97
p-token-ring-bug2.9	U	**925.20**	T.O	T.O	M.O	T.O	T.O
p-token-ring-bug2.10	-	T.O	T.O	T.O	M.O	T.O	T.O
p-toy-bug-1	U	**13.66**	18.39	T.O	M.O	T.O	*47.09
p-toy-bug-2	U	25.04	**24.88**	T.O	M.O	T.O	*52.23
p-toy	S	**99.90**	T.O	T.O	M.O	T.O	N.A
p-transmitter.1	U	**0.07**	0.09	8.20	7.73	470.19	2.84
p-transmitter.2	U	**0.29**	**0.29**	T.O	E.R	618.49	*8.96
p-transmitter.3	U	**0.49**	**0.49**	T.O	E.R	T.O	*21.49
p-transmitter.4	U	**0.89**	0.99	T.O	M.O	T.O	*43.43
p-transmitter.5	U	**1.59**	1.79	T.O	E.R	T.O	*101.71
p-transmitter.6	U	**2.49**	2.99	T.O	M.O	T.O	*180.11
p-transmitter.7	U	**3.97**	4.89	T.O	M.O	T.O	*299.19
p-transmitter.8	U	**5.89**	7.89	T.O	M.O	T.O	*503.97
p-transmitter.9	U	**8.57**	12.38	T.O	M.O	T.O	*815.18
p-transmitter.10	U	**11.66**	20.08	T.O	M.O	T.O	T.O
rod1-bug.c ⋆	U	**28.89**	34.16	T.O	M.O	T.O	*312.76
rod1.c ⋆	S	**285.18**	308.38	T.O	M.O	T.O	N.A
rod2.c ⋆	S	**76.84**	88.13	T.O	M.O	T.O	N.A
modtrans-bug.c	U	**64.77**	379.59	T.O	M.O	T.O	*643.61
modtrans-nudc-bug1.c ⋆	U	**238.11**	T.O	T.O	M.O	T.O	*583.50
modtrans-nudc-bug2.c ⋆	U	**232.91**	T.O	T.O	M.O	T.O	*569.42
modtrans-nudc-bug3.c ⋆	U	**131.25**	657.79	T.O	M.O	T.O	*584.66
modtrans-nudc-bug4.c ⋆	U	**111.53**	675.21	T.O	M.O	T.O	*604.64
modtrans-nudc1.c ⋆	S	**211.64**	T.O	T.O	M.O	T.O	N.A
modtrans-nudc2.c ⋆	U	**157.75**	649.20	T.O	M.O	T.O	*827.36
modtrans-rec1-bug1.c	U	**4.08**	4.89	T.O	M.O	T.O	*588.06
modtrans-rec1-bug2.c	U	**4.09**	4.90	T.O	M.O	T.O	*602.50
modtrans-rec1.c	-	T.O	T.O	T.O	M.O	T.O	N.A
modtrans-rec2-bug1.c	U	**6.27**	17.58	T.O	M.O	T.O	*585.99
modtrans-rec2-bug2.c	U	**4.49**	6.89	T.O	M.O	T.O	*607.05
modtrans-rec2.c	-	T.O	T.O	T.O	M.O	T.O	N.A
modtrans.c	-	T.O	T.O	T.O	M.O	T.O	N.A
modtrans2-nudc-bug1.c ⋆	U	**143.80**	T.O	T.O	M.O	T.O	*810.07
modtrans2-nudc-bug2.c ⋆	U	**173.04**	T.O	T.O	M.O	T.O	*567.25
modtrans2-nudc-bug3.c ⋆	U	**236.81**	T.O	T.O	M.O	T.O	*564.44
modtrans2-nudc-bug4.c ⋆	U	**237.52**	T.O	T.O	M.O	T.O	*561.79
modtrans2-nudc-bug5.c ⋆	U	**226.81**	T.O	T.O	M.O	T.O	*804.03
modtrans2-nudc1.c ⋆	U	**150.09**	720.50	T.O	M.O	T.O	*557.49
modtrans2-nudc2.c ⋆	U	**159.44**	705.55	T.O	M.O	T.O	*565.91
ss1.c ⋆	U	**4.59**	6.98	T.O	M.O	T.O	*235.45
ss2-bug.c ⋆	U	**1.19**	1.29	38.04	M.O	T.O	*7.12
train-hytech-bug1.c ⋆	U	**0.29**	**0.29**	T.O	M.O	883.89	472.02
train-hytech-bug2.c ⋆	U	**0.29**	0.30	T.O	M.O	995.21	451.60
train-hytech-bug3.c ⋆	U	**13.79**	15.88	T.O	M.O	784.59	492.15
train-hytech-bug4.c ⋆	U	**13.78**	15.68	T.O	M.O	784.22	454.18
train-hytech1.c ⋆	S	**34.46**	57.56	T.O	M.O	787.31	N.A
train-hytech2.c ⋆	S	**34.25**	52.34	T.O	M.O	789.87	N.A
train1-bug.c ⋆	U	**0.99**	1.09	T.O	M.O	T.O	*208.07
train1.c ⋆	S	**1.59**	**1.59**	T.O	M.O	T.O	N.A
train2-bug.c ⋆	U	**2.59**	2.78	T.O	M.O	T.O	*200.73
train2.c ⋆	S	**2.09**	2.19	T.O	M.O	T.O	N.A

TABLE II
RESULTS OF GROUNDING APPROACH VS. S3ST.

	Number of Ground Values (ESST)					S3ST
	2	3	4	5	6	
#Unsafe/Safe/T.O	1/9/0	3/7/0	4/6/0	4/4/2	2/0/8	-
Max. Unsafe Time	12.9	47.2	193.8	700.8	828.7	25.1
Avg. Unsafe Time	12.9	39.9	159.5	457.2	590.6	25.1
Max. Safe Time	12.1	51.8	167.5	388.8	-	-
Avg. Safe Time	5.7	32.4	115.8	305.2	-	-

p-toy-bug-2

example, given a primitive function call `wait_time(`t`)` for a non-constant time t and assume that t ranges over the set $\{v_0, \ldots, v_k\}$ of concrete values, we replace the call with the following code:

```
assume(t == v0 || ... || t == vk);
if (t == v0) wait_time(v0); ...;
if (t == vk) wait_time(vk);
```

This is clearly an under-approximation, that can only be used for bug finding. The results for `p-toy-bug-2` are reported on Table II. The column k reports the results of 10 experiments with k concrete values. The table also compares the grounding approach with S3ST (on the original parametric benchmark). The table shows that increasing the number of values may increase the chance to find violations; however, the performance of ESST degrades (and possibly times out), even when it does find the bug.

VII. CONCLUSION AND FUTURE WORK

We have presented a novel approach, called S3ST, to the verification of designs where the interactions between thread and scheduler are parametric. The key feature of the approach is the semi-symbolic analysis of the scheduler, that requires a careful control of the interactions between the concrete and symbolic data. The approach allows us to verify parametric designs that are out of reach for techniques based on sequentialization, and is also competitive for bug finding.

For future work, we want to improve further the scalability of ESST and S3ST by applying symmetry reduction, and to generalize the methods to the case of multi-threaded software that is parameterized on the number of threads.

REFERENCES

[1] D. Tabakov, G. Kamhi, M. Y. Vardi, and E. Singerman, "A Temporal Language for SystemC," in *FMCAD*, A. Cimatti and R. B. Jones, Eds. IEEE, 2008, pp. 1–9.

[2] A. Cimatti, A. Micheli, I. Narasamdya, and M. Roveri, "Verifying SystemC: A software model checking approach," in *FMCAD*, R. Bloem and N. Sharygina, Eds. IEEE, 2010, pp. 51–59.

[3] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction," in *POPL*. ACM, 2002, pp. 58–70.

[4] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*, ser. Frontiers in Art. Int. and Applications, A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds. IOS Press, 2009, vol. 185, pp. 825–885.

[5] A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, and M. Roveri, "Kratos - a software model checker for SystemC," in *CAV*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 310–316.

[6] A. Cimatti, I. Narasamdya, and M. Roveri, "Boosting Lazy Abstraction for SystemC with Partial Order Reduction," in *TACAS*, ser. LNCS, P. A. Abdulla and K. R. M. Leino, Eds., vol. 6605. Springer, 2011, pp. 341–356.

[7] D. Campana, A. Cimatti, I. Narasamdya, and M. Roveri, "An analytic evaluation of SystemC encodings in promela," in *SPIN*, ser. LNCS, A. Groce and M. Musuvathi, Eds., vol. 6823. Springer, 2011, pp. 90–107.

[8] A. Cimatti, I. Narasamdya, and M. Roveri, "Software model checking with explicit scheduler and symbolic threads," *Journal of Logical Methods in Computer Science*, vol. 8, no. (2:18), 2012, arXiv:1206.3182v2 [cs.LO], http://arxiv.org/abs/1206.3182.

[9] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, 2003.

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

[11] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1165–1178, 2008.

[12] S. Chaki, A. Gurfinkel, and O. Strichman, "Time-bounded analysis of real-time systems," in *FMCAD*, P. Bjesse and A. Slobodova, Eds. FMCAD Inc, 2011, pp. 72–80.

[13] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks, "Uppaal 4.0," in *QEST*. IEEE, 2006, pp. 125–126.

[14] R. Alur, "Formal verification of hybrid systems," in *EMSOFT*, S. Chakraborty, A. Jerraya, S. K. Baruah, and S. Fischmeister, Eds. ACM, 2011, pp. 273–278.

[15] A. Cimatti, L. Palopoli, and Y. Ramadian, "Symbolic computation of schedulability regions using parametric timed automata," in *IEEE Real-Time Systems Symposium*. IEEE Computer Society, 2008, pp. 80–89.

[16] A. Cimatti, S. Mover, and S. Tonetta, "Efficient scenario verification for hybrid automata," in *CAV*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 317–332.

[17] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *PLDI*, V. Sarkar and M. W. Hall, Eds. ACM, 2005, pp. 213–223.

[18] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "EXE: Automatically Generating Inputs of Death," *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, 2008.

[19] C. S. Pasareanu, N. Rungta, and W. Visser, "Symbolic execution with mixed concrete-symbolic solving," in *ISSTA*, M. B. Dwyer and F. Tip, Eds. ACM, 2011, pp. 34–44.

[20] C. S. Pasareanu and N. Rungta, "Symbolic pathfinder: symbolic execution of java bytecode," in *ASE*, C. Pecheur, J. Andrews, and E. D. Nitto, Eds. ACM, 2010, pp. 179–180.

[21] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: a platform for in-vivo multi-path analysis of software systems," in *ASPLOS*, R. Gupta and T. C. Mowry, Eds. ACM, 2011, pp. 265–278.

[22] D. Beyer and M. E. Keremoglu, "CPAchecker: A Tool for Configurable Software Verification," in *CAV*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 184–190.

[23] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav, "SATABS: SAT-Based Predicate Abstraction for ANSI-C," in *TACAS*, ser. LNCS, N. Halbwachs and L. D. Zuck, Eds., vol. 3440. Springer, 2005, pp. 570–574.

[24] E. M. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," in *TACAS*, ser. LNCS, K. Jensen and A. Podelski, Eds., vol. 2988. Springer, 2004, pp. 168–176.