Theory-Aided Model Checking of Concurrent Transition Systems

Guy Katz
Weizmann Institute of Science
guy.katz@weizmann.ac.il

Clark Barrett
New York University
barrett@cs.nyu.edu

David Harel
Weizmann Institute of Science
david.harel@weizmann.ac.il

***Abstract*—We present a method for the automatic compositional verification of certain classes of concurrent programs. Our approach is based on the casting of the model checking problem into a theory of transition systems within CVC4, a DPLL(T) based SMT solver. Our transition system theory then cooperates with other theories supported by the solver (e.g., arithmetic, arrays), which can help accelerate the verification process. More specifically, our theory solver looks for known patterns within the input programs and uses them to generate lemmas in the languages of other theories. When applicable, these lemmas can often steer the search away from safe parts of the search space, reducing the number of states to be explored and expediting the model checking procedure. We demonstrate the potential of our technique on a number of broad classes of programs.**

I. Introduction

In concurrent programming, the size of the composite program is typically exponential in the number of its constituent threads. This phenomenon, an instance of the *state explosion* problem, is a major hindrance to the verification of concurrent software. In recent decades, a prominent approach to tackling this difficulty has been that of compositional verification [13]: properties of threads are derived/verified in isolation, and are used to deduce global system correctness, without exploring the entire composite state space. When applicable, compositional verification can often significantly outperform direct verification techniques.

A key challenge in compositional verification is how to *automatically* come up with "good" thread properties — those whose verification is considerably cheaper than the verification of the global property on the one hand, but which are sufficiently meaningful to imply the desired system properties on the other. Automatic property generation is essential in rendering a compositional verification scheme scalable [11].

Since the compositional verification of arbitrary programs is difficult (and often impossible [9]), one reasonable approach is to trade generality for effectiveness — i.e., to limit the scope of programs that a scheme handles, in exchange for better performance on programs that remain within that scope. Here, we adopt this approach and propose an automatic compositional verification scheme for certain kinds of concurrent software.

The paper has two main contributions. The first is the rigorous formalization and implementation of a solver for a *theory of transition systems* ($\mathcal{TS}$) within the context of CVC4 [1] — a lazy, DPLL(T) based SMT solver [28]. The $\mathcal{TS}$ solver takes as input formulas describing a program's concurrent threads (given as transition systems) and the assertion that a certain safety property is violated; and it answers UNSAT if the program is safe, or SAT if it is not. As a standalone module, the $\mathcal{TS}$ solver explores the space of reachable states in order to determine a system's safety — an exploration that is driven by the SMT solver's underlying SAT engine.

Several existing approaches utilize SMT solvers in model checking (e.g., Lazy Annotation [26] and PDR [8]), but typically the process is driven by a model checker that uses an SMT solver as a black-box tool. In our approach the roles are reversed, and the SMT engine, via the $\mathcal{TS}$ solver, can be regarded as invoking a model checker. This design allows other theories within CVC4 to be seamlessly used in analyzing the input program at hand, determining which parts of the state space should be explored and which may safely be ignored. These theories may then influence the search conducted by the $\mathcal{TS}$ solver by asserting lemmas to the underlying DPLL(T) core, sometimes pruning significant portions of the search space and greatly improving performance. We term this process *theory-aided model checking*: the $\mathcal{TS}$ solver explores the state space while also looking for opportunities in which other theories may aid and direct the search.

The second contribution of the paper is in the way other theories determine which parts of the state space may be ignored during model checking. We perform this by having the $\mathcal{TS}$ solver analyze the input threads and look for pre-supplied *patterns*: structural properties of the threads that may be expressed as assertions in the languages of other theories, such as arithmetic or arrays. It is through these assertions that other theories can "understand" the program and efficiently discover, e.g., that a certain branch of the search space cannot lead to a violation. A key fact here is that each thread/transition system is analyzed separately — and hence the compositionality of our approach: the analysis complexity is proportional to the size of the program and not to that of its state space. We thoroughly describe three of the currently implemented patterns.

While our proposed technique is compositional and completely automatic, it is useful only when the input programs match one of the pre-supplied patterns. This is in line with our approach of trading generality for effectiveness, and, as we demonstrate in later sections, our approach is capable of effectively handling broad classes of programs even with just a few stored patterns.

The type of software that we target here is a family of discrete event systems. In particular, we focus on a computational model that has three fundamental concurrency idioms — requesting events, waiting-for events and blocking events — which we term the $\mathcal{RWB}$ model. The $\mathcal{RWB}$ concurrency idioms are widespread and appear, sometimes

in related forms, in various formalisms such as *publish-subscribe* architectures [12], *supervisory control* [29] and *live sequence charts* (LSC) [10]. Together, these three idioms also form the *behavioral programming* (*BP*) model [20]. Thus, by focusing on the $\mathcal{RWB}$ model, we hope to make our technique applicable (with appropriate adjustments) to a variety of programming formalisms. Further, we believe that the technique can be extended to cater to additional concurrency idioms and models.

The rest of the paper is organized as follows. In Section II we recap the definitions of the DPLL(T) framework for SMT solvers and of the $\mathcal{RWB}$ model. Next, in Section III we introduce the theory of transition systems ($\mathcal{TS}$) and describe a theory solver aimed at model checking $\mathcal{RWB}$ programs. In Section IV we demonstrate how the $\mathcal{TS}$ solver can cooperate with other theory solvers in order to expedite model checking. Subsequently, we apply our technique to two broad classes of problems: *periodic problems* in Section V, and programs with shared arrays in Section VI. Experimental results appear in Section VII, and we conclude with a discussion and related work in Section VIII.

II. Definitions

The DPLL(T) Framework. DPLL(T) [28] is an extensible framework used by modern SMT solvers. It employs multiple specialized theory solvers that interact with a SAT solver. The SAT solver maintains an input formula F and a partial assignment M for F. Periodically, a theory solver is asked whether M is satisfiable in its theory; and, if it is not, the theory solver generates a conflict clause, the negation of an unsatisfiable subset of M, that is added to F. The theory solver may request case splitting by means of the splitting-on-demand paradigm [2], which allows the solver to add theory lemmas to F consisting of clauses possibly with literals not occurring in F.

The $\mathcal{RWB}$ Model. In this work we focus on the $\mathcal{RWB}$ model for concurrent discrete event systems. An $\mathcal{RWB}$ program is comprised of a set of events and set of threads that communicate via the requesting, waiting-for and blocking of events. More specifically, the threads repeatedly synchronize with each other at predetermined synchronization points, and at each such point they each declare events that they request and events that they block. Then, an event that is requested by at least one thread and not blocked is selected for triggering, and all the threads that requested or waited-for this event proceed with their execution. Whenever a new synchronization point is reached, the process is repeated.

The $\mathcal{RWB}$ model is not intended to be programmed in directly. Rather, it is used to describe the underlying transition systems of threads written in higher level languages, for the purpose of analysis and verification. Actual programming in $\mathcal{RWB}$ is performed, e.g., using the *behavioral programming* (*BP*) framework [20], which is implemented in various high level languages such as C++ or Java (see http://www.b-prog.org). Thus, while inter-thread communication in BP is performed solely through the $\mathcal{RWB}$ idioms, threads may internally use any construct provided by the underlying programming language (e.g., C++). Indeed, the tool and examples described in this paper were prepared using a C++ version of BP (termed *BPC* [16]), and CVC4. It should further be noted that the $\mathcal{RWB}$ definitions as given here entail global lockstep synchronization between components, which may cause unwanted overhead. There exist extensions to $\mathcal{RWB}$ that mitigate this difficulty without altering the model's semantics [14], and our technique is applicable to these extensions as well.

Formally, an $\mathcal{RWB}$-thread T over event set E is a tuple $T = \langle Q, \delta, q_0, R, B\rangle$, where Q is a set of states (one for each synchronization point), q_0 is the initial state, $R : Q \to 2^E$ and $B : Q \to 2^E$ map states to events requested and blocked at these states (respectively), and $\delta : Q \times E \to 2^Q$ is a transition function (the definition is adopted from [19]).

$\mathcal{RWB}$ programs are created by *composing* $\mathcal{RWB}$-threads. The parallel composition of threads $T^1 = \langle Q^1, \delta^1, q_0^1, R^1, B^1\rangle$ and $T^2 = \langle Q^2, \delta^2, q_0^2, R^2, B^2\rangle$, both over the same event set E, yields the $\mathcal{RWB}$-thread defined by $T^1 \parallel T^2 = \langle Q^1 \times Q^2, \delta, \langle q_0^1, q_0^2\rangle, R^1 \cup R^2, B^1 \cup B^2\rangle$, where $\langle \tilde{q}^1, \tilde{q}^2\rangle \in \delta(\langle q^1, q^2\rangle, e)$ iff $\tilde{q}^1 \in \delta^1(q^1, e)$ and $\tilde{q}^2 \in \delta^2(q^2, e)$. The union of the labeling functions is defined in the natural way, i.e. $e \in (R^1 \cup R^2)(\langle q^1, q^2\rangle)$ iff $e \in R^1(q^1) \cup R^2(q^2)$. An *$\mathcal{RWB}$ program* P comprised of $\mathcal{RWB}$-threads $T^1, T^2, \ldots, T^n$ is the composite thread $P = T^1 \parallel \ldots \parallel T^n$. Denoting $P = \langle Q, \delta, q_0, R, B\rangle$, an execution of P starts from q_0, and in each state q along the run an enabled event is chosen for triggering, if one exists (i.e., an event $e \in R(q) - B(q)$). Then, the execution moves to state $\tilde{q} \in \delta(q, e)$, and so on. An execution can either be infinite, or finite if it ends in a state with no successors (a *deadlock* state). An illustration of a simple $\mathcal{RWB}$ program appears in Fig. 1.

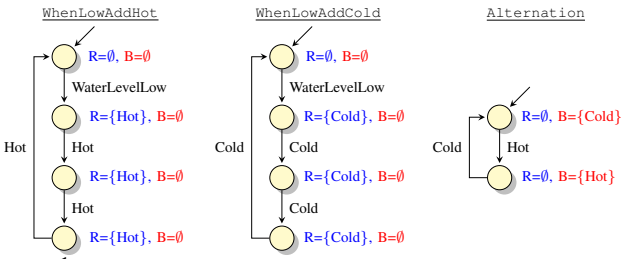

Figure 1: An $\mathcal{RWB}$ program for controlling the water level in a tank with hot and cold water sources. Each node corresponds to a synchronization point in a thread, labeled with its requested (R) and blocked (B) events. Waited-for events are not labeled, and are represented by transitions. If an event that a thread did not wait for is triggered, the thread does not change states. In the program depicted, the $\mathcal{RWB}$-thread `WhenLowAddHot` repeatedly waits for `WaterLevelLow` events (requested by a sensor thread, not shown) and requests three times the event `Hot`. `WhenLowAddCold` performs a similar action with the event `Cold`. In order to keep the water temperature stable, the `Alternation` thread enforces the interleaving of `Hot` and `Cold` events by using event blocking.

From a software-engineering perspective, the motivation for using the $\mathcal{RWB}$ idioms for inter-thread communication lies in the model's strict and simple synchronization mechanism. Studies show that this form of inter-thread interaction — i.e., through repeated synchronization and declaration of requested, waited-for and blocked events — facilitates incremental, non-intrusive development, and the resulting systems often have threads that are aligned with the specification [20].

Verifying $\mathcal{RWB}$ Programs. In [19], [22], the authors demonstrate how the transition systems underlying $\mathcal{RWB}$-

threads can be automatically extracted from high level code and then, using abstraction techniques, be symbolically traversed in order to verify safety properties. Safety properties are themselves expressed by *marker* $\mathcal{RWB}$-threads, marking that a violation has occurred with a special API call [19]. For simplicity, we assume that marker threads signal that a violation has occurred by blocking all events, causing a deadlock. Thus, safety checking is reduced to checking for deadlock freedom.

The manual compositional verification of $\mathcal{RWB}$ programs is discussed in [15]. There, it is shown how the simple $\mathcal{RWB}$ synchronization mechanism facilitates the generation of individual thread properties, which are then used for proving the system property at hand. The beneficial effect that simple concurrency idioms have on verification is also discussed in [18]. Indeed, the simplicity of the $\mathcal{RWB}$ idioms plays a key role in the pattern matching algorithm that we discuss later.

III. The Theory of Transition Systems

We now cast the model checking of $\mathcal{RWB}$ into a DPLL(T) setting, by defining a dedicated *theory of transition systems* ($\mathcal{TS}$). We assume familiarity with the definitions of many-sorted first order logic (see, e.g., [3]). The theory is parameterized by a set $\bar{Q} = \{Q_1, \ldots, Q_n\}$ of *state sorts* used to represent the state sets of the program's constituent threads. Let $\bar{Q}^+$ denote the composite state sorts obtained by taking the Cartesian product of one or more elements in $\bar{Q}$. Every element $Q \in \bar{Q}^+$ is a sort in $\mathcal{TS}$. Further, every such Q is also associated with a matching transition system sort, S_Q. Finally, $\mathcal{TS}$ has an event sort, E.

For every $Q \in \bar{Q}^+$ the signature includes: the predicate $I_Q : S_Q \times Q$, indicating initial states; the predicates $R_Q, B_Q : S_Q \times Q \times E$ to indicate whether an event is requested (R_Q) or blocked (B_Q) at a given state; and the predicate $Tr_Q : S_Q \times Q \times E \times Q$ to indicate the state transition rules.

In order to reason about composite transition systems, the signature includes the following functions and predicates. For every $Q^1, Q^2 \in \bar{Q}^+$ we have the transition system composition function $\|_{Q^1Q^2}$: $S_{Q^1} \times S_{Q^2} \rightarrow S_{Q^1 \times Q^2}$ (Recall that $(Q^1 \times Q^2)$ is itself a sort in $\bar{Q}^+$); and also the $pair_{Q^1Q^2} : Q^1 \times Q^2 \rightarrow (Q^1 \times Q^2)$ function for composing states, which, per the $\mathcal{TS}$ semantics, is a bijection. Later we often omit the Q subscripts when clear from the context.

For each $Q^1, Q^2 \in \bar{Q}^+$, $\mathcal{TS}$ has the following axioms which enforce the $\mathcal{RWB}$ composition rules. A composite state is initial iff its components are initial states:

$$\begin{aligned}&\forall s_1 : S_{Q^1}, s_2 : S_{Q^2}, s : S_{Q^1 \times Q^2}.\ s = s_1 \parallel s_2 \implies \\ &\quad \forall q : Q_1 \times Q_2.\ (I(s,q) \iff \\ &\quad \exists q_1 : Q_1, q_2 : Q_2.\ (I(s_1,q_1) \wedge I(s_2,q_2) \wedge q = pair(q_1,q_2))).\end{aligned}$$

Composite transitions are performed component-wise:

$$\begin{aligned}&\forall s_1 : S_{Q^1}, s_2 : S_{Q^2}, s : S_{Q^1 \times Q^2}.\ s = s_1 \parallel s_2 \implies \\ &\quad \forall q, q' : Q^1 \times Q^2, e : E.\ (Tr(s,q,e,q') \iff \\ &\quad \exists q_1, q_1' : Q^1, q_2, q_2' : Q^2.\ (q = pair(q_1,q_2) \wedge \\ &\qquad q' = pair(q_1', q_2') \wedge Tr(s_1,q_1,e,q_1') \wedge Tr(s_2,q_2,e,q_2'))).\end{aligned}$$

Requested and blocked events in a composite state are the union of those in the component states:

$$\begin{aligned}&\forall s_1 : S_{Q^1}, s_2 : S_{Q^2}, s : S_{Q^1 \times Q^2}.\ s = s_1 \parallel s_2 \implies \\ &\quad \forall q : Q^1 \times Q^2, e : E.(R(s,q,e) \iff \exists q_1 : Q^1, q_2 : Q^2. \\ &\quad q = pair(q_1,q_2) \wedge (R(s_1,q_1,e) \vee R(s_2,q_2,e))) \wedge \\ &\quad (B(s,q,e) \iff \exists q_1 : Q^1, q_2 : Q^2. \\ &\quad q = pair(q_1,q_2) \wedge (B(s_1,q_1,e) \vee B(s_2,q_2,e))).\end{aligned}$$

As previously discussed, by encoding safety properties as threads of the program to be checked, safety is reduced to deadlock freedom. For each $Q \in \bar{Q}^+$, the signature includes a $deadlock_Q : S_Q \times Q$ predicate, such that:

$$\begin{aligned}&\forall s : S_Q, q : Q.\ (deadlock(s,q) \iff \\ &\quad \neg\exists q' : Q, e : E.\ Tr(s,q,e,q') \wedge R(s,q,e) \wedge \neg B(s,q,e)),\end{aligned}$$

and the $safe\_state_Q : S_Q \times Q$ predicate, with:

$$\begin{aligned}&\forall s : S_Q, q : Q.\ \neg safe\_state(s,q) \implies deadlock(s,q) \vee \\ &\quad \exists q' : S_Q, e : E.\ (Tr(s,q,e,q') \wedge R(s,q,e) \wedge \\ &\qquad \neg B(s,q,e) \wedge \neg safe\_state(s,q')).\end{aligned}$$

$\neg safe\_state_Q(s,q)$ indicates that state q is unsafe, because it is (or can lead to) a deadlock state. Finally, for each $Q \in \bar{Q}^+$, $safe_Q : S_Q$ indicates that a transition system is safe:

$$\forall s : S_Q.\ \neg safe(s) \iff \exists q : Q.\ I(s,q) \wedge \neg safe\_state(s,q).$$

The Theory Solver. Inputs for the $\mathcal{TS}$ solver start with a *preamble* $\mathfrak{P}$ that contains assertions that describe the program's threads. Specifically, $\mathfrak{P}$ includes variables $s_1 \ldots, s_n$, each of sort S_Q for some basic state sort $Q \in \bar{Q}$; and for every s_i it includes assertions describing its initial states, its transitions and its requested and blocked events. After $\mathfrak{P}$, the solver expects an assertion Φ about the system's safety: $s = s_1 \parallel s_2 \parallel s_3 \parallel \ldots \parallel s_n \wedge \neg safe(s)$. The solver then returns SAT iff s is determined to be unsafe.

Fig. 2 shows derivation rules used to implement a simple explicit-state model checker.[1] Intuitively, $\mathcal{TS}$ traverses the state graph in a DFS-like manner, looking for bad states. The underlying SAT solver manages the splits by deciding which successor state to check at every point. The process ends when a deadlock state is found or when the state space has been exhausted and no derivation rules apply; an example appears in Fig. 3. As demonstrated in the next section, this implementation allows us to seamlessly leverage other theory solvers in curtailing the state space, which may reduce the overall runtime. Additional details and proofs of correctness and termination appear in Section A of the supplementary material [23].

IV. Automatic Analysis of Transition Systems

The calculus in Section III captures the basic proof strategy of our theory solver: a forward reachability search. We next enrich this basic strategy with additional derivation rules, aimed at narrowing down the state space that needs to be explored. The idea is to include within the $\mathcal{TS}$ solver a database of *structural patterns* that characterize common/useful threads and alongside each pattern also to keep lemmas that describe these threads' behavior in the language of some other theory

[1] While we do not assume the system is finite-state, we do assume that the initial states and the successors for each state are finite and decidable.

$$\textsc{Start}\ \frac{\Gamma[\neg safe(s)]}{\Gamma, \neg safe\_state(s, q_1) \quad \ldots \quad \Gamma, \neg safe\_state(s, q_n)}\ \text{IF}\ \begin{array}{c}\Gamma \vDash_{TS} q_1, \ldots, q_n \text{ ARE THE INITIAL STATES OF } s \\ \text{AND } \forall_{1 \le i \le n}.\ \neg safe\_state(s, q_i) \notin \Gamma\end{array}$$

$$\textsc{Decide}\ \frac{\Gamma[\neg safe\_state(s, q)]}{\Gamma, \neg safe\_state(s, q_1) \quad \ldots \quad \Gamma, \neg safe\_state(s, q_n)}\ \text{IF}\ \begin{array}{c}\Gamma \vDash_{TS} q_1, \ldots, q_n \text{ ARE THE SUCCESSORS OF } q\ (n \ge 1) \\ \text{AND } \neg deadlock(s, q) \notin \Gamma\end{array}$$

$$\textsc{Unsat}\ \frac{\Gamma[\neg safe\_state(s, q)]}{\bot}\ \text{IF}\ \forall q'.\ \neg safe\_state(s, q') \in \Gamma \implies \neg deadlock(s, q') \in \Gamma$$

$$\textsc{Deadlock-Lemma}: \mathfrak{P} \wedge \Phi \implies \neg deadlock(s, q)\ \text{IF}\ q \text{ HAS A SUCCESSOR IN } s$$

Figure 2: Γ represents an arbitrary set of assertions that the solver has gathered at a given state, and $\Gamma[\phi]$ indicates that ϕ appears in Γ. The *Start* rule starts the traversal of the graph: the solver initiates a forward reachability search for bad states by nondeterministically guessing an initial state that is unsafe. When a state with unvisited successors is asserted to be unsafe, the *Decide* rule is used to nondeterministically assert that one of its successors is unsafe. Splitting is handled through the splitting-on-demand feature of the DPLL(T) framework. The UNSAT rule closes branches that fail to reach a deadlock state. If all branches terminate with $\bot$, UNSAT is returned; otherwise, if a branch terminates with a state other than $\bot$ where no rule is applicable, we return SAT. The last rule, *Deadlock-Lemma*, is a lemma generation rule: the resulting lemma is theory-valid, i.e. does not depend on the context in which it was generated. These lemmas mark that a non-deadlock state has been visited, and that it does not need to be revisited in the future. As part of the proof strategy, the $\mathcal{TS}$ solver invokes the lemma generation rule for (s, q) immediately after the *Decide* rule is invoked for $\neg safe\_state(s, q)$, and only then (provided that q is not a deadlock state). This strategy, together with the side-conditions on the derivation rules, ensures that no state is visited more than once. See Section A of the supplementary material [23] for more details.

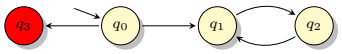

Figure 3: The depicted program has a reachable deadlock state, q_3. After reading the preamble, the solver uses the *Start* rule to assert $\neg safe\_state(s, q_0)$. Then, it invokes the *Decide* rule for state q_0, nondeterministically asserting $\neg safe\_state(s, q_1)$. This invocation of *Decide* is followed by the generation of the lemma $\neg deadlock(s, q_0)$. Next, *Decide* is invoked for state q_1, generating the assertion $\neg safe\_state(s, q_2)$ — followed by the lemma $\neg deadlock(s, q_1)$. *Decide* is then invoked for state q_2, generating $\neg safe\_state(s, q_1)$ and the lemma $\neg deadlock(s, q_2)$. At this point, the conditions for the UNSAT rule are met, and the solver closes this branch of the tree. The solver backtracks to the last nondeterministic split and generates the assertion $\neg safe\_state(s, q_3)$. State q_3 is deadlocked, and so the *Deadlock-Lemma* rule is not invoked. No additional derivation rules apply, and so the process terminates with a SAT result, indicating that the system is unsafe.

in CVC4. As the $\mathcal{TS}$ solver traverses the state space, it also repeatedly checks to see if any of the patterns apply to the threads at hand. When a match is found, the solver asserts the matching lemmas to the SMT framework. Sometimes, these lemmas may be contradictory to the assertion that the safety property is violated along the current search path, and another theory solver will raise a conflict: this will cause the $\mathcal{TS}$ solver to backtrack and check other areas of the state space.

We demonstrate the method on a simple example, adopted from [15]. Observe an $\mathcal{RWB}$ program over event set $E = \{0, 1\}$ that generates the event sequence $(0^5 \cdot (0+1))^\omega$. The program has three threads, depicted in Fig. 4. The safety property to be verified is that event 1 is never triggered (and so, the program is unsafe). Observe that direct model checking of this system requires visiting 6 composite states.

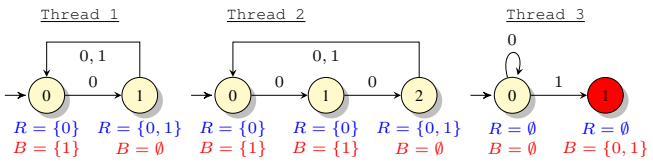

Figure 4: An $\mathcal{RWB}$ program adopted from [15]. Thread 1 counts the number of events triggered so far, modulo 2. Every second event it requests both events 0 and 1; otherwise, it requests 0 but blocks 1. Thread 2 does the same, but counts modulo 3. Thread 3 is a *bad marker*: it waits for a violation to occur, i.e. for 1 to be triggered, and then goes into a "bad" state that blocks all the events, forcing a deadlock. This $\mathcal{RWB}$ program can have 0 triggered at every index, and can have 1 triggered precisely every 6 events. Consequently, it is unsafe.

For this program as input, the $\mathcal{TS}$ solver performs the following automatic compositional proof. First, it compares the transition systems to its pattern bank, and recognizes that they match the *looped* thread mold — a thread whose state is determined uniquely by the step index in the run (assuming a violation has not occurred). This is a structural property of each thread, that is checked locally and in isolation from its siblings. After determining that all threads are looped, the solver finds all individual thread states in which 1 is not blocked. In our case, this is state 1 for thread 1, state 2 for thread 2, and state 0 for thread 3. Denoting composite states as triplets, this is state $\langle 1, 2, 0 \rangle$. Finally, the solver uses the gathered information to generate the following lemma in order to curtail the state space:

$$\begin{aligned}&\mathfrak{P} \wedge \Phi \implies ((\neg safe\_state(s, \langle 0, 0, 0 \rangle) \implies \\ &\neg safe\_state(s, \langle 1, 2, 0 \rangle)) \ \wedge \ \exists t : \mathbb{N}. \\ &(t \equiv 1 \pmod 2) \wedge (t \equiv 2 \pmod 3) \wedge (t \equiv 0 \pmod 1))).\end{aligned}$$

This lemma connects the safety of the initial state $\langle 0, 0, 0 \rangle$ with that of the only state in which 1 is not blocked, state $\langle 1, 2, 0 \rangle$ — provided that there exists an integer t for which $t \equiv 1 \pmod 2$, $t \equiv 2 \pmod 3$ and $t \equiv 0 \pmod 1$. Because, in looped threads, the step index determines the state, this last part captures the fact that state $\langle 1, 2, 0 \rangle$ is reachable.

Upon generation of this lemma, CVC4 asserts the lemma's arithmetical clauses to the arithmetic solver. If the latter determines that there is no solution for t, CVC4 answers UNSAT on the entire query. This signifies that the system is safe, which is indeed the case if state $\langle 1, 2, 0 \rangle$ cannot be reached. However, if the arithmetic solver manages to solve for t, as is the case here, the $\mathcal{TS}$ solver continues exploring the successors of state $\langle 1, 2, 0 \rangle$ and discovers that it has a bad successor. Then, SAT is returned for the query.

The key observation is that through the automatically generated lemma, the 4 intermediate states between state $\langle 0, 0, 0 \rangle$ and $\langle 1, 2, 0 \rangle$ did not need to be explored. Because the threads matched the looped pattern, CVC4 was able to deduce that these intermediate states would be safe iff state $\langle 1, 2, 0 \rangle$ was safe. Further, because the arithmetic solver can solve for t more quickly than the intermediate states can be traversed (especially when generalizing to $(0^n \cdot (0+1))^\omega$ for a large n), the solver's performance is improved.

Pattern Matching. The $\mathcal{TS}$ solver's pattern database consists of *pattern matchers*. A pattern matcher P is comprised of a family of *recognizer predicates* $\{R_n\}_{n \ge 1}$, where R_n is defined over n transition system variables $s_1, \ldots, s_n$, and

a lemma generating function f (described later). For input system $s = s_1 \parallel \ldots \parallel s_n$, we say that pattern P applies to s if $R_n(s_1, \ldots, s_n)$ evaluates to true. The R_n predicates can encode various facts about the transition systems: e.g., that threads always or never block certain events, that they have a certain state that must always be revisited, that certain events always send threads into a deadlock state, etc. For example, in the previously discussed looped pattern, R_n evaluates to true iff each of the threads' states has precisely one successor state.

In our proof-of-concept C++ implementation, recognizer predicates are coded as Boolean methods that take as input a list (of arbitrary length) of transition systems. Upon receiving a query, the $\mathcal{TS}$ solver passes the input program's threads to the recognizer predicates of each of the patterns, to determine which patterns apply in this case. Recognizer implementations may traverse the given transition systems, compute strongly connected components, etc. The only restriction, needed for the method to be efficient, is that recognizers do not compute the composite transition systems of the system; they are restricted to (polynomial) operations on the individual threads. Thus, the complexity of pattern matching is polynomial in the size of the individual threads — and because these threads are typically exponentially smaller than the composite program [17], we can quickly test multiple patterns.

The second component in a pattern matcher is the lemma generating function, f. When pattern P applies to an input program, its lemma generating function is invoked repeatedly during state space traversal, in order to allow P to generate lemmas that affect the search. Specifically, f is invoked whenever $\mathcal{TS}$ visits a new state q (i.e., after the *Decide* rule generates the assertion $\neg safe\_state(s, q)$), and returns a (possibly empty) list of lemmas concerning the safety of state q. The $\mathcal{TS}$ solver then asserts these lemmas to the underlying SAT engine, and other theories may use them in trimming the search space. In practice, the generated lemmas may depend on parameters extracted from the input threads by the pattern recognizers. For example, in the looped pattern, the size of the loop is extracted by the recognizer and is then used in generated lemmas.

Limitations. The above example demonstrates our method's potential advantages, but also raises a question regarding its generality: can the pattern database be sufficiently extensive, i.e. apply to a sufficient range of programs, so as to make our approach worthwhile? Indeed, if one needed to "teach" the solver new patterns for every new input program, the method would boil down to a manual compositional proof.

We believe that the answer to this question is affirmative: our findings show that even a small set of patterns included within the $\mathcal{TS}$ solver may already apply to broad classes of interesting programs. We demonstrate two such cases, *periodic programs* and *programs with arrays*, in Sections V and VI. Still, adding new patterns is not a trivial task, and so we store them in a central repository — amortizing the cost of adding additional patterns over future applications.

The $\mathcal{TS}$ Solver vs. Model Checking. In the simple example given above, our theory-aided approach could also be implemented by a more standard design: a model checker that issues queries to a black-box SMT solver. Our motivation for conducting model checking within the $\mathcal{TS}$ solver is in handling more elaborate examples, in which SMT theories partake in directing the state space traversal (see, e.g., Section V). While such cases can still be accommodated by a model checker that is "running the show" and an SMT solver that exposes proper callbacks, we feel that a DPLL(T)-based solution is cleaner, and also more extensible and robust. By encoding the state traversal engine as a few axioms and lemma generation rules, and by having the pattern matching mechanism likewise generate lemmas, the complexity of integrating and synchronizing the two is automatically and seamlessly handled by CVC4's DPLL(T) core — simplifying the implementation of the $\mathcal{TS}$ solver. Further, this enables the $\mathcal{TS}$ solver to be plugged into any other SMT solver that adheres to the DPLL(T) framework.

V. Verifying Periodic Programs

In this section, we discuss the theory-aided verification of *periodic programs* [25] — a class of single processor scheduling problems that have been widely studied over the last decades. A periodic program consists of a finite set of *tasks* $T_1, \ldots, T_n$, which are processes that repeatedly need to be scheduled for execution on a single processor. Each task T_i is characterized by its period time P_i and an execution time C_i (for simplicity, we ignore here other parameters such as relative deadlines and initial offsets). From task T_i's point of view, the execution of the program is divided into time cycles of length P_i each, and in each such cycle the task must be alloted C_i time slots on the processor. The least common multiple of the tasks' period times is called the program's *hyper-period*. Tasks may have *priorities*: a task with a higher priority will preempt another if both need to be scheduled at a specific point in time. A periodic program is said to be *schedulable* if there exists a task scheduling in which no deadlines are violated. See Section C of the supplementary material [23] for an example.

Here, we study the verification of safety properties in periodic programs: we assume that the input program is schedulable, and check whether it can violate a given property. This is typically done by transforming the periodic program into an equivalent sequential program and then verifying it using standard model checking [7]. Our approach is similar, but we seek to leverage the program's special structure in order to explore only a portion of its state space.

In $\mathcal{RWB}$, periodic programs may be programmed by expressing each task as a thread that requests an event whenever the task needs to be scheduled [15]. Priorities are expressed using blocking: a thread (task) may block events belonging to other threads with lesser priorities. Fig. 5 illustrates the structure of task threads and describes their pattern matcher.

Whenever all input threads are identified as *tasks*, the pattern recognizer reports that the program is periodic. This causes the pattern's lemma generation function to be repeatedly invoked during state space traversal, so that it may generate lemmas aimed at curtailing the search space. For this pur-

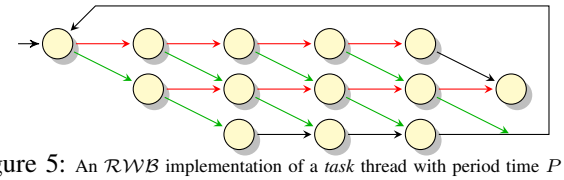

Figure 5: An $\mathcal{RWB}$ implementation of a *task* thread with period time $P = 5$ and execution time $C = 2$. The thread's underlying transition system can be regarded as a $(C+1) \times P$ matrix, where the columns represent the time passed since the beginning of the period and the rows represent the number of times the task has been scheduled so far. Green edges in the figure represent the task being successfully scheduled (i.e., its requested event was triggered) and red edges represent the task not being scheduled (an event requested by some other task was triggered). Thus, with every time unit the state moves to the right, and if the task was scheduled it also moves one row down. If the task's deadline is violated, it enters a deadlock state (the rightmost state in the figure). The task pattern matcher traverses the state graph of each input thread and checks whether it has these structural properties. If so, it also extracts the task's P and C parameters and its sequence of requested events (not illustrated). If blocking is used to prioritize tasks, the matcher also extracts the prioritization hierarchy.

pose, we extend the signature of $\mathcal{TS}$ to include a sort $\mathbb{Z}^+$ for non-negative integers, and the predicate $deadlock_Q : S_Q \times \mathbb{Z}^+$. Intuitively, $deadlock_Q(s,t)$ indicates that a deadlock state in s is reachable in t steps from an initial state. Further, we extend $\mathcal{TS}$ to support backward reachability analysis, in addition to the forward reachability analysis afforded by the *safe_state* predicate. To this end, we add the $reachable_Q : S_Q \times Q$ predicate, with the following semantics:

$$\forall s : S_Q, q : Q.reachable(s,q) \implies I(s,q) \vee \exists q' : S_Q, e : E.$$
$$(Tr(s,q',e,q) \wedge R(s,q',e) \wedge \neg B(s,q',e) \wedge reachable(s,q'))$$

Intuitively, a state is reachable if it is initial or has a reachable predecessor. For more details, see Section B of the supplementary material [23].

The lemmas generated by the pattern matcher assert that there must be a time t within the hyper-period in which a violation occurs. They also limit the possible values of t based on the information gathered about the individual tasks. Specifically, the pattern matcher generates the lemma:

$$\Psi \wedge \Phi \implies \exists t : \mathbb{Z}^+.\ deadlock(s,t) \wedge \Psi_t$$

where Ψ_t describes constraints on t that are deduced from the structure of the task threads. If the arithmetic solver finds a solution t_0 for Ψ_t it assigns it to t, and the $\mathcal{TS}$ solver then translates it, by analyzing the task threads' possible locations in time t, into candidate reachable bad states $q_1, \ldots, q_\ell$:

$$\Psi \ \wedge \ \Phi \wedge deadlock(s,t_0) \implies \vee_{i=1}^{\ell} reachable(s,q_i)$$

$\mathcal{TS}$ then performs backward reachability checks on candidates $q_1, \ldots, q_\ell$. If a path to an initial state is found, the system is unsafe and we are done. Otherwise, the contradiction forces the arithmetic solver to propose another solution $t = t_1$, which corresponds to additional candidate bad states. The process is repeated until the system is proven unsafe, or until all possible solutions are exhausted. Other bad states, which do not correspond to any of the proposed values of t, are guaranteed to be unreachable and are ignored.

In order to generate the constraints in Ψ_t, the pattern matcher identifies tasks *participating* in the violation: these are the threads whose requested events are part of a violating sequence. Then, it uses information about these threads, and about threads with higher priority, to put constraints on t.

We demonstrate this on a schedulable periodic program with 4 tasks: task T_1 with parameters $P_1 = 5, C_1 = 1$; T_2 with $P_2 = 6, C_2 = 1$; T_3 with $P_3 = 9, C_3 = 3$; and task T_4 with parameters $P_4 = 11, C_4 = 2$. Task 1 has the highest priority, task 2 has the second highest priority, and tasks 3 and 4 both share the lowest priority. The safety property in question is that it is impossible for task T_4 to be scheduled for three consecutive time slots. Here, direct model checking requires visiting 55000 states in the composite program.

By intersecting the violating event sequence with the events requested by each thread, the pattern matcher determines that T_4 is the only participating task. By the information extracted regarding task priorities, it deduces that tasks T_1 and T_2 supersede it. Then, it generates the Ψ_t constraint as follows. One conjunct in Ψ_t is $0 \leq t \leq 990$, as the hyper-period is $lcm(5,6,9,11) = 990$. Another conjunct is $((t \geq 3 \pmod 5) \wedge (t \geq 3 \pmod 6))$: if it did not hold, T_1 or T_2 would preempt T_4, preventing it from being scheduled 3 consecutive times. Yet another conjunct is $(t \leq 1 \pmod{11})$; it holds because in order for T_4 to be scheduled 3 consecutive times (with execution time $C_4 = 2$), a fresh period must start at time t or $t-1$. A few additional conjuncts are omitted. The complete lemma reduces the number of possible values for t from 990 to just 15, and the query as a whole entails exploring only 700 states out of 55000 reachable states in order to prove the system's safety.

VI. Verifying Programs With Shared Arrays

Next we demonstrate the theory-aided verification of programs with shared arrays — a widespread construct in concurrent programming. In the $\mathcal{RWB}$ model, a shared m-ary array with n cells may be implemented using n b-threads, each of size m. Each thread represents a single array cell and has a clique-like structure, where each state s_i is associated with a write event w_i and a read event r_i. Intuitively, each state s_i corresponds to a value v_i that is stored in the array cell. Whenever event w_i is triggered, the thread moves to state s_i; and whenever not in state s_i, the thread blocks r_i. Thus, other threads can request r_i in order to check if the thread is in state s_i (i.e., to check if the array cell has value v_i). See Fig. 6 for an illustration. Note that this implementation is only needed for shared arrays; internally, threads may use any construct available in the underlying programming language.

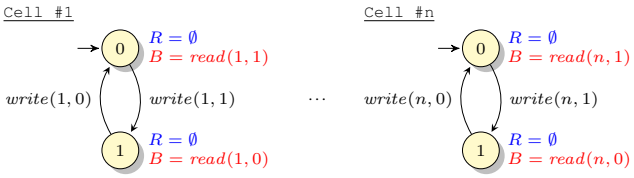

Figure 6: An $\mathcal{RWB}$ implementation of a binary array with n cells. Each cell is represented by a thread with two states, signifying the stored value in that cell, 0 or 1. Each thread/cell is associated with two `write` events, for 0 and 1; when they occur, the thread changes states to indicate the new stored value. Other threads in the program may read from a cell by requesting the two `read` events associated with it, one for 0 and one for 1; the read event that does not match the value in the cell will be blocked by the cell thread, and so only the "correct" read event may be triggered.

The $\mathcal{TS}$ solver has a pattern matcher that looks for threads that match this *array cell* pattern. If an array is found, the pattern matcher checks whether deadlocks are possible only in certain array configurations (e.g., when certain array cells

hold certain values; an example appears later in this section). If such constraints are found, it generates a lemma that conditions the system's unsafety on the array threads reaching an unsafe configuration.

We demonstrate with an example. Observe a program with a shared array of size n and an initial state q_0. The array pattern matcher creates an array expression arr_{q_0} whose value at each index i is set to some fresh constant c_i. This expression is used to represent the value of the array in various states of the program. The matcher also creates a *target* array, arr_{target}, and asserts constraints on arr_{target} signifying the state that the array has to be in for a violation to occur. Then, it generates the lemma $\mathfrak{P} \wedge \Phi \implies (arr_{q_0} = arr_{target})$.

The bulk of the work is then performed as $\mathcal{TS}$ traverses the state space. Whenever a new state q is visited, the pattern matcher analyzes the threads (each of them separately), looking for array entries that have become fixed. This can be determined, e.g., when additional write events to a cell are never requested or are always blocked. Suppose that it is discovered that the first cell's value has been fixed to e_0; then the lemma $\mathfrak{P} \wedge \Phi \wedge \neg safe\_state(s, q) \implies (c_0 = e_0)$ is generated. If this is consistent with the earlier assertion $arr_{q_0} = arr_{target}$, the solver continues traversing the successors of q; otherwise, the array theory solver will raise a conflict, resulting in q's successor states not being traversed.

A more detailed example and an evaluation of applying the shared array pattern to a web-server application appears in Section VII. An additional detailed example regarding the verification of an $\mathcal{RWB}$ application for playing Tic-Tac-Toe appears in Section D of the supplementary material [23].

VII. Experimental Results

We evaluated our proof-of-concept tool, implemented as an extension to CVC4, by comparing it to BPMC — a symbolic model checker specifically designed for $\mathcal{RWB}$ programs [19], [22] (the tool and experiments are available online [23]). Our tool uses a portfolio approach: if the input program does not match any of the known patterns, the tool simply invokes BPMC (or any other model checker, for that matter). The decision of whether or not to invoke BPMC is made within seconds, rendering the performance of both tools effectively the same in these cases. Hence, for the remainder of this section we focus on inputs in which a pattern did apply and theory-aided model checking was indeed attempted.

We first compared the tools using a benchmark suite of over 120 hand-crafted $\mathcal{RWB}$ programs — some periodic, and some containing shared arrays. The benchmarks' sizes ranged from a few hundred to over 10 million reachable states, and contained both SAT and UNSAT instances. The results are depicted and disucssed in Fig. 7.

Next, we set out to test our tool's applicability to a large, real-world system by using it to verify safety properties on a web-server (implementing TCP and HTTP stacks) written in BPC [16]. We were very curious to see whether our pattern recognition mechanism would pick up any matching threads.

As it turns out, the shared array pattern proved useful in verifying this application. Per the TCP protocol, the web-server only accepts TCP *push* segments on active connections. Slightly simplified, a connection to a client is active if the client sent a *syn* segment but not a *fin* segment. This functionality is implemented using blocking: for every connection, a dedicated thread, named *EnsureActiveConnection*, blocks *push* events while the connection is inactive. This blocking is removed when a *syn* segment is received, and is restored when a *fin* segment is received. Thus, the *EnsureActiveConnection* threads were picked up as shared array cells by our tool: they each had two states, labeled *active* and *inactive*, with respective read events *push* and *reject* and write events *syn* and *fin*. Interestingly, the programmers of the web-server did not seem to have had this design pattern in mind [16].

We tested 10 safety properties on the web-server (see Fig. 8). These properties included the proper rejection of messages on inactive connections, proper usage of allotted sequence numbers for outgoing segments, and the detection and blocking of unstable clients, who quickly and repeatedly opened and closed connections. The theory-aided approach did better on 7 of 10 instances (4 SATs and 3 UNSATs), demonstrating an average speedup of 16% over all instances. BPMC did better on 2 SAT and 1 UNSAT instances, where the property in question and the discovered patterns were disparate (e.g., properties involving proper usage of sequence numbers, that had nothing to do with the *EnsureActiveConnection* threads).

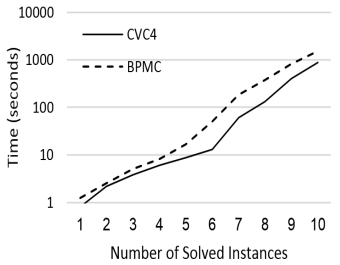

Figure 8: Experiments on the web-server.

These initial results are encouraging. We conclude that (i) the theory-aided approach is viable, in the sense that the stored patterns apply to real programs, sometimes significantly reducing verification times; and (ii) that performance may be further improved by enhancing the portfolio approach; i.e., if we were able to more accurately characterize cases in which, despite matching a stored pattern, a thread does not affect the property in question, we could delegate those cases to BPMC and achieve faster running times. This is left for future work.

VIII. Related Work and Discussion

In this work, we proposed a framework for the automated compositional verification of concurrent software. Our technique was based on casting the model checking problem into the DPLL(T) framework used by the CVC4 SMT solver, and then utilizing other theory solvers to prune the search space in order to improve performance. Other theories were able to affect the search through lemmas in their respective languages that were generated by matching the input program's threads to presupplied patterns.

SMT solving has been used for various verification-related tasks such as lemma dispatching [8], [26], reachability analysis [4] and model-checking concurrent programs [6], [27]. Our technique shares some of these aspects, but differs in that the state exploration is driven by an SMT solver and

		# Instances	Avg. # States Explored			Avg. Time (milliseconds)		
			CVC4	BPMC	Change	CVC4	BPMC	Change
Periodic Programs	SAT	11	9994	9236	+8%	18791	15894	+18%
	UNSAT	50	35299	184388	-80%	10247	15041	-31%
	UNSAT†	6	59816	8195666	N/A	170673	809946	N/A
	Timeout	2						
Shared Arrays	SAT	35	24416	293525	-91%	24882	168755	-85%
	UNSAT	15	121133	511292	-76%	124911	292779	-57%
	UNSAT†	6	267000	1989666	N/A	359324	1510028	N/A
Total		111	190842	998441	-80%	178831	492469	-63%

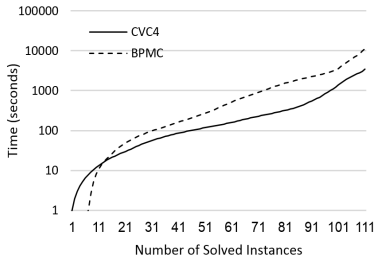

Figure 7: Experiments on a benchmark suite, conducted using an X230 Lenovo laptop with 16GB memory. The suite contained SAT and UNSAT instances of periodic $\mathcal{RWB}$ programs and programs with shared arrays. The table compares our tool (*CVC4* columns) to the *BPMC* tool, measuring the average number of explored states and average solving time for each category. The *Change* columns measure the effectiveness of CVC4 in comparison to BPMC. The UNSAT† row indicates UNSAT instances on which CVC4 answered correctly but on which BPMC ran out of space (but listing the number of states it was able to explore). The *Timeout* row indicates instances on which both tools ran out of space/time. We did not encounter examples on which BPMC returned and CVC4 did not. The table reveals that for SAT queries on periodic programs, BPMC was able to outperform CVC4. This is not surprising; indeed, the pattern for periodic programs is designed to quickly show that bad states are unreachable, which is not the case for SAT instances. In all other categories, i.e. UNSAT queries on periodic programs and both types of queries on programs with shared arrays, CVC4 typically outperformed BPMC. Instances where BPMC did better were either very small (the cost of thread analysis and pattern matching exceeded the cost of the actual model checking), or instances where the property in question had nothing to do with the recognized patterns, making it impossible for our tool to trim the search space. The UNSAT† instances had too many states for BPMC to cover, but with the theory-aided approach we were able to trim the search space down to a manageable size. Finally, the *Timeout* instances were too large to handle, even with theory-aided pruning. The *Total* row sums up the instances solved by both tools, demonstrating an encouraging average speedup of 63%; these 111 instances are also the ones described in the graph.

in that lemmas are derived using stored patterns. A related approach for circuit verification appears in [5], where the input is analyzed to find unreachable states in advance. Our framework follows a similar spirit, but extends the technique to concurrent software and utilizies a modern SMT solver.

In [30], the authors extend the Z3 solver with an automaton sort for symbolic automata over infinite alphabets. It would be interesting to combine this technique with ours, enabling it to reason about $\mathcal{RWB}$ programs with infinite event sets.

We evaluated our technique on two broad classes of $\mathcal{RWB}$ programs: periodic programs and programs with shared arrays. Specifically, we showed how the $\mathcal{TS}$ solver may leverage CVC4's arithmetic and array theory solvers in order to expedite the model checking process. Others have explored SMT-based techniques for similar models; e.g., the validation of guessed invariants in Lustre programs [21]. We consider this as encouragement that applying SMT-based techniques to synchronous, discrete event models may prove fruitful, and intend to extend our technique to Lustre as well.

We find our initial results encouraging, and plan to continue extending our pattern database. One direction that we are presently pursuing is the addition of a new pattern matcher that leverages CVC4's *string* theory solver [24], by translating constraints imposed by certain types of input threads into regular expressions. Indeed, a prototype implementation we have created shows interesting potential.

Acknowledgements. The research of Katz and Harel was supported by a grant from the Israel Science Foundation, and by the Philip M. Klutznick Fund for Research, the Joachimowicz Fund and the Benoziyo Fund for the Advancement of Science at the Weizmann Institute of Science.

REFERENCES

[1] C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. *CAV*, 2011.
[2] C. Barrett, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Splitting On Demand in SAT Modulo Theories. *LPAR*, 2006.
[3] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability Modulo Theories. *Handbook of Satisfiability*, 2009.
[4] J. Berdine, N. Bjørner, S. Ishtiaq, J. Kriener, and C. Wintersteiger. Resourceful Reachability as HORN-LA. *LPAR*, 2013.
[5] P. Bjesse and K. Claessen. SAT-Based Verification without State Space Traversal. *FMCAD*, 2000.
[6] R. Bryant, S. Lahiri, and S. Seshia. Modeling and Verifying Systems using a Logic of Counter Arithmetic with Lambda Expressions and Uninterpreted Functions. *CAV*, 2002.
[7] S. Chaki, A. Gurfinkel, S. Kong, and O. Strichman. Compositional Seqentialization of Periodic Programs. *VMCAI*, 2013.
[8] A. Cimatti and A. Griggio. Software Model Checking via IC3. *CAV*, 2012.
[9] J. Cobleigh, G. Avrunin, and L. Clarke. Breaking Up is Hard to do: an Investigation of Decomposition for Assume-Guarantee Reasoning. *ISSTA*, 2006.
[10] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *J. on Formal Methods in System Design*, 2001.
[11] M. Dwyer, J. Hatcliff, R. Robby, C. Pasareanu, and W. Visser. Formal Software Analysis Emerging Trends in Software Model Checking. *ICSE*, 2007.
[12] P. Eugster, P. Felber, R. Guerraoui, and A. Kermarrec. The Many Faces of Publish/Subscribe. *CSUR*, 2003.
[13] O. Grumberg and D. Long. Model Checking and Modular Verification. *TOPLAS*, 1994.
[14] D. Harel, A. Kantor, and G. Katz. Relaxing Synchronization Constraints in Behavioral Programs. *LPAR*, 2013.
[15] D. Harel, A. Kantor, G. Katz, A. Marron, L. Mizrahi, and G. Weiss. On Composing and Proving the Correctness of Reactive Behavior. *EMSOFT*, 2013.
[16] D. Harel and G. Katz. Scaling-Up Behavioral Programming: Steps from Basic Principles to Application Architectures. *AGERE*, 2014.
[17] D. Harel, G. Katz, R. Lampert, A. Marron, and G. Weiss. On the Succinctness of Idioms for Concurrent Programming. *CONCUR*, 2015.
[18] D. Harel, G. Katz, A. Marron, and G. Weiss. The Effect of Concurrent Programming Idioms on Verification. *MODELSWARD*, 2015.
[19] D. Harel, R. Lampert, A. Marron, and G. Weiss. Model-Checking Behavioral Programs. *EMSOFT*, 2011.
[20] D. Harel, A. Marron, and G. Weiss. Behavioral Programming. *C. ACM*, 2012.
[21] T. Kahsai, Y. Ge, and C. Tinelli. Instantiation-Based Invariant Discovery. *NFM*, 2011.
[22] G. Katz. On Module-Based Abstraction and Repair of Behavioral Programs. *LPAR*, 2013.
[23] G. Katz, C. Barrett, and D. Harel. Theory-Aided Model Checking of Concurrent Transition Systems: Supplementary Material. https://sites.google.com/site/guykatzhomepage/fmcad15.
[24] T. Liang, A. Reynolds, C. Tinelli, C. Barrett, and M. Deters. A DPLL(T) Theory Solver for a Theory of Strings and Regular Expressions. *CAV*, 2014.
[25] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, 1973.
[26] K. McMillan. Lazy Annotation Revisited. *CAV*, 2014.
[27] L. D. Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari. SAL 2. *CAV*, 2004.
[28] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T). *J. ACM*, 2006.
[29] P. Ramadge and W. Wonham. Supervisory Control of a Class of Discrete Event Processes. *SIAM J. on Control and Optimization*, 1987.
[30] M. Veanes and N. Bjørner. Symbolic Automata: The Toolkit. *TACAS*, 2012.