Verifying Synchronous Reactive Systems using Lazy Abstraction

Kumar Madhukar*†, Mandayam Srivas†, Björn Wachter‡, Daniel Kroening‡ and Ravindra Metta*

*Tata Research Development and Design Center, Pune, Maharashtra, India
Email: {kumar.madhukar, r.metta}@tcs.com
†Chennai Mathematical Institute, Chennai, Tamil Nadu, India
Email: mksrivas@hotmail.com
‡Department of Computer Science, University of Oxford, United Kingdom
Email: {bjoern.wachter, kroening}@cs.ox.ac.uk

***Abstract*—Embedded software systems are frequently modeled as a set of synchronous reactive processes. The transitions performed by the processes are given as sequential, atomic code blocks. Most existing verifiers flatten such programs into a global transition system, to be able to apply off-the-shelf verification methods. However, this monolithic approach fails to exploit the lock-step execution of the processes, severely limiting scalability.**

We present a novel formal verification technique that analyses synchronous concurrency explicitly rather than encoding it. We present a variant of Lazy Abstraction with Interpolants (LAWI), a technique successfully used in software verification, and tailor it to synchronous reactive concurrency. We exploit the synchronous communication structure by fixing an execution schedule, circumventing the exponential blow-up of state space caused by simulating synchronous behaviour by means of interleavings. The technique is implemented in SYMPARA, a verification tool for synchronous reactive systems. To evaluate the effectiveness of our technique, we compare SYMPARA with Bounded Model Checking and k-induction, and a LAWI-based verifier for multi-threaded (asynchronous) software. On several realistic examples SYMPARA outperforms the other tools by an order of magnitude.

I. INTRODUCTION

Synchronous reactive processes are widely used for modeling and model-based design of embedded software systems. Processes of this kind synchronize at designated points in their control flow. Harel's *Statecharts* [1] is a popular formalism for specifying such processes. Statecharts extend conventional state-transition diagrams with the notion of hierarchy, concurrency and communication. Existing approaches to formal verification of statecharts (or, in fact, most other formalisms specifying synchronous reactive systems) predominantly rely on building a global transition relation for the system. This permits the application of a wide range of standard methods for state space exploration such as BDD-based Model Checking, Bounded Model Checking, k-induction or interpolation. The key disadvantage of the approach is that it requires that a scheduler is added to the model to account for all possible process interleavings, which increases the complexity of the model substantially. Furthermore, the approach fails to exploit the structure inherent in the control-flow graph of the processes.

An alternative is to use a verifier for asychronously composed threads and instrument the model to enforce synchronization of the processes at their synchronization points. This not only adds further states per thread (owing to the synchronization), but also leads to exploration of many irrelevant states, as interleavings that happen between the synchronization points have no effect.

In this paper, we explore a third, novel way of verifying synchronous reactive systems. Our approach uses a model checking algorithm based on Lazy Abstraction with Interpolants (LAWI) [2], also known as the IMPACT algorithm. It unwinds the control-flow graph of the program into an *abstract reachability tree* (ART). Whenever the exploration arrives at an error state, the nodes on the error path are annotated with invariants that prove infeasibility of the error path. The crux of the algorithm is a *covering* check that allows it to soundly stop the unwinding and terminate with a correctness proof of the program. This combination of low-cost program unwindings combined with path-based refinement and covering checks gives rise to an efficient software model checking algorithm.

Recently, the IMPACT algorithm was extended to support asynchronous concurrent processes using an interleaved semantics and implemented in a tool called IMPARA [3]. IMPARA, which analyses concurrent C programs with POSIX or Win32 threads, combines partial-order reduction with the IMPACT algorithm. We have tailored and optimized IMPARA to implement our new technique, which we call SYMPARA. The essence of our algorithm is a novel *concurrent static-single assignment (SSA)* form which enables us to use a fixed schedule for process execution. While the motivation for this work has been to formally verify STATEMATE [4] statechart specifications, there are several other formalisms (Esterel, Lustre, SIGNAL, etc.) implementing similar semantics for which SYMPARA would work effectively too.

We discuss the related work in the remainder of this section before illustrating the idea of concurrent SSAs in Section II. We briefly introduce IMPARA in Section III. Section IV presents the SYMPARA algorithm and our major optimizations. We discuss our experimental results in Section V.

Related Work

Most existing efforts in verifying synchronous reactive systems [5]–[7] extract the global transition relation imple-

978-3-9815370-4-8/DATE15/©2015 EDAA

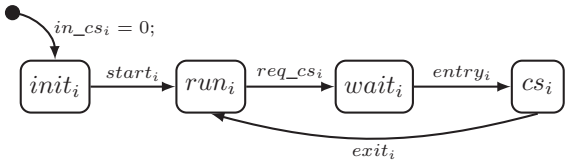

Fig. 1. State-transition diagram for i^{th} process

mented by the underlying system and encode it in the input language of a popular model checker. There have also been attempts [8] at preserving the underlying structure of these processes. SYMPARA, unlike these, (a) exploits the semantics of the underlying formalism by fixing a schedule, (b) employs a verification technique based on LAWI, and (c) supports input in ANSI-C, offering applicability to formalisms that permit code generation.

The most closely related work is by Cimatti et al. [9], who have used a similar software model checking approach based on lazy abstraction to verify SystemC models. They also avoid adding a scheduler, however, their analysis explores all possible interleavings. The distinguishing feature of SYMPARA is its ability to work with a *fixed schedule* of process execution, accounting for synchronous races with an efficient encoding.

II. CONCURRENT SSA

Consider two processes p_i and p_j, executing concurrently and synchronously, performing the transitions given in Fig. 1. The variables $start_i$, req_cs_i and rel_cs_i are inputs supplied by the environment. The value of in_cs_i indicates whether p_i is in state cs_i or not. It is set (reset) by all transitions entering (exiting) cs_i. The symbols $entry_i$ and $exit_i$ are place holders for the labels "$(lock = 0) \wedge (\oplus_i\ in(wait_i))$ / $lock = 1; in\_cs_i = 1;$" and "rel_cs_i / $lock = 0; in\_cs_i = 0;$", respectively. The variable $lock$ is shared and used by the processes to mark their entry to cs. The unary predicate $in(s)$ is `true` if the corresponding process is in state s. We are interested in verifying the correctness of this protocol, i.e., $in(cs_i)$ and $in(cs_j)$ must never evaluate to 1 simultaneously, as per the STATEMATE semantics.

Following STATEMATE conventions, a transition labeled $e[c]/a$ is said to be *enabled* if the *event* e (treated as a predicate for the purpose of this discussion) and *condition* c evaluate to `true`. Here, a denotes the *action* that is performed when the transition occurs. STATEMATE requires execution to progress in *steps*. In each *step*, a transition, if one is enabled, is executed in each process. It is this property of synchronous systems that we exploit in order to fix a schedule. The actions execute atomically and take effect only at the end of that step. Note that this gives rise to *write-write* races. The system keeps evolving till it reaches a state where none of the transitions are enabled (called a *stable* state). At this stage, inputs from the environment are read and the system evolves further. Evidently, a sound verification technique for synchronous reactive systems must reason over all schedules, which requires exponential effort in the worst case.

In Fig. 1, if the processes are at states cs_i and cs_j (say), and if rel_cs_i and rel_cs_j evaluate to `true`, then p_i and p_j modify the variable $lock$ in the same step. While both the processes here set $lock$ to 0, the values may differ in general. Instead of considering both orders of execution (p_i followed by p_j and vice-versa), based on a non-deterministically chosen boolean b, SYMPARA introduces the assignment "$lock = b?v : v'$" where v and v' are the values assigned to $lock$ by different threads. We term the SSA generated from such a conditional expression as *concurrent SSA*. It is concurrent SSAs which allow fixing an execution order. Note that concurrent SSAs are needed only when v is not equal to v'. During formal analysis, SYMPARA does syntactic checks and constant propagation before invoking a decision procedure to resolve this (in)equality.

III. IMPARA

IMPARA extends the original IMPACT algorithm to concurrent programs. The algorithm returns either a safety invariant for a given program, finds a counterexample or diverges (the verification problem is undecidable). To this end, the algorithm constructs an abstraction of the program execution in the form of an ART, which corresponds to an unwinding of the program's control-flow graph, annotating nodes with invariants. To put abstract reachability trees to work for proving program correctness for unbounded executions, a criterion is needed to prune the tree without missing any error paths. This role is assumed by a *covering* relation between the nodes.

Intuitively, the purpose of node labels (denoted by ϕ) is to represent inductive invariants, i.e., over-approximations of sets of states, and the covering relation (denoted by $\triangleright$) is the equivalent of a subset relation between nodes. Consider two nodes, v and w, which share the same control location (a vector of individual thread locations, which we denote by $\mathfrak{l}$), and $\phi(v)$ implies $\phi(w)$. If we establish that the superset node w cannot be on an error path, we do not need to search for an error path from subset node v. Therefore, if we can find a safety invariant for w, we do not need to explore successors of v. In this case, we say that the node w *covers* the node v.

We omit the details of the IMPARA algorithm for lack of space. Although our description of SYMPARA is complete in itself and does not presume knowledge of IMPARA, interested readers may refer to [3] for a thorough discussion of IMPARA.

IV. SYMPARA

We now present an extension of the IMPARA algorithm for multi-threaded software, to synchronously composed concurrent programs. Our algorithm differs from that of IMPARA in exactly two aspects.

- SYMPARA iterates over the set of processes and computes successors in a *fixed* order, tackling races using concurrent SSAs. This fixed order of process execution is encoded in the ART by augmenting the control information at each node with the process scheduled next on that node.
- SYMPARA adds an additional clause in the covering criterion, that the covering node and the covered node must agree on the process scheduled next.

Algorithm 1 SYMPARA algorithm for verifying synchronous reactive processes

```
1:  procedure MAIN()
2:      Q := {ϵ}, ▷ := ∅
3:      while Q ≠ ∅ do
4:          v := dequeue(Q); CLOSE(v)
5:          if v not covered then
6:              if error(v) then
7:                  REFINE(v)
8:              EXPAND(v)
9:      return P is safe
10:
11: procedure CLOSE(v)
12:     for w ∈ V^{≺v} : w uncovered do
13:         if ι(v) = ι(w) ∧ φ(v) ⇒ φ(w) then
14:             if sp(v) = sp(w) then
15:                 ▷ := ▷ ∪ {(v, w)}
16:                 ▷ := ▷ \ {(x, y) ∈ ▷ | v ⇝ y}
17: procedure EXPAND(v)
18:     (l, φ) := v
19:     T := sp(v)
20:     if T = size(T) then
21:         st(v) = 0
22:         T = 0
23:     for (l, (R, l')) ∈ A(T) with ι_T = l do
24:         // A(T) := actions of T
25:         // R := transition constraint (l → l')
26:         w := fresh node
27:         sp(w) := T + 1
28:         ι(w) := ι[T ↦ l']
29:         φ(w) := true
30:         Q := Q ∪ {w}, V := V ∪ {w}
31:         →:=→ ∪{(v, T, R, w)}
32: procedure REFINE(v)
33:     if φ(v) ≡ False then
34:         return
35:     π := v_0, ... v_N path from ϵ to v
36:     if F(π) has interpolant A_0 ... A_N then
37:         for i = 0 ... N do
38:             φ := A_i
39:             if φ(v_i) ⊭ φ then
40:                 Q := Q ∪ {w | w ▷ v_i}
41:                 ▷ := ▷ \ {(w, v_i) | w ▷ v_i}
42:                 φ(v_i) := φ(v_i) ∧ φ
43:         for w ∈ V s.t. w ⇝ v do
44:             CLOSE(w)
45:     else
46:         abort (program unsafe)
```

In the absense of the second criteria, above, if a process p becomes idle (does not change the global system state), the node obtained after executing p could incorrectly be covered by the one before executing p. The following definition formalizes *cover* for SYMPARA.

Definition IV.1. *A node v is said to be covered by another node w (denoted as $w \triangleright v$) if v and w have the same control location, $\phi(v) \rightarrow \phi(w)$ and the next scheduled process on both the nodes are the same i.e., $sp(v) = sp(w)$.*

The algorithm (Algorithm 1) constructs an ART by alternating three operation on nodes: EXPAND, REFINE, and CLOSE.

EXPAND takes an uncovered leaf node and computes its successors along the next scheduled process. The procedure maintains a set of active processes and schedules them in a round-robin fashion to generate new successor nodes. The resulting node is sensitive to the schedule order *only* when there is a race. We use concurrent SSAs to model these races, as described in Section II. For every enabled transition, EXPAND creates a fresh tree node w, schedules the next process on w, updates its location to the target location of the transition and initializes $\phi(w)$ to `true`. The node w is then enqueued to a work list Q and a tree edge is added which records the step from v to w, with the transition constraint R. If w happens to be an error location, the operation REFINE is invoked.

REFINE takes an error node v, detects if the error path is feasible and, if not, updates the node labels in order to eliminate the path. It determines if the unique path π from the initial node to v is feasible by checking satisfiability of the transition constraints, $\mathcal{F}(\pi)$, along π. If $\mathcal{F}(\pi)$ is satisfiable, the solution gives a counterexample in the form of a concrete error trace, proving the program unsafe. Otherwise, an interpolant is obtained, which is used to refine the labels and update $\triangleright$.

CLOSE takes a node v and checks if v can be added to $\triangleright$. As potential candidates for pairs $w \triangleright v$, it only considers nodes created before v, denoted by the set $V^{\prec v} \subsetneq V$. This ensures a stable behavior, as covering a node may uncover other nodes. To ensure the soundness of $\triangleright$, all pairs (x, y) where y is a descendant of v, denoted by $v \rightsquigarrow y$, are removed from $\triangleright$ at this point, as v and all its descendants are covered.

MAIN first initializes the queue with the initial node ϵ, and the relation $\triangleright$ with the empty set. It then runs the main loop of the algorithm until Q is empty, i.e., until the ART is complete, unless an error is found which exits the loop. In the main loop, a node is selected from Q. First, CLOSE is called to try and cover it. If the node is not covered and it is an error node, REFINE is called. Finally, the node is expanded, unless it was covered, and evicted from Q. We have added several optimizations in SYMPARA to improve its efficiency. In particular, SYMPARA

- handles synchrony in concurrent processes by scheduling every process in each step. This leads to addition of intermediate states (when only a subset of threads have executed) in the ART. SYMPARA extends $\triangleright$ to include these intermediate states as well.
- simplifies the task of SMT solver in two ways: (a) it uses a light-weight decision procedure for cover checks, which decides logical implication in a conservative way, and (b), it does syntactic simplification to reduce the complexity of verification conditions that are passed to solver.
- eliminates infeasible paths early during the path exploration, i.e., the path conditions are decided as they arise, while traditional LAWI lazily decides path-feasibility only when a program assertion is reached.
- eagerly resolves races to reduce case-splits.

V. EXPERIMENTS

We experimentally compare SYMPARA with IMPARA and CBMC. Since each of these tools is meant or optimized for a different class of programs, to keep the comparison fair, we have encoded each example used in our experiments in the form that is suitable for the tool it is given to. To obtain a correctness proof in the presence of unbounded control loops, we have used k-induction for our experiments with CBMC.

The benchmarks used in our experiments, listed in Table I, include STATEMATE, Simulink and Lustre programs. The first example, *mutex*, is a simple three-process model for implementing a mutual exclusion protocol. The *vw_alarm* example is a hierarchical eight-process statechart subsystem extracted from a real model of an alarm system. We have

TABLE I
EXPERIMENTAL RESULTS

No.	Tools → Example	Property	SYMPARA SAT #calls	SYMPARA SAT Time	SYMPARA Time	IMPARA SAT #calls	IMPARA SAT Time	IMPARA Time	CBMC + k-Induction Unwind	CBMC + k-Induction SAT	CBMC + k-Induction Time
1.	*mutex*	*correctness*	632	3.53	5.77	–	–	timeout	69	–	timeout
2.	*mutex*	*stability*	767	5.26	8.02	–	–	timeout	62	–	timeout
3.	*vw_alarm*	*non-determinism*	138	0.64	1.80	–	–	timeout	18	–	timeout
4.	*vw_alarm*	*stability*	1897	87.29	214.82	–	–	timeout	11	–	timeout
5.	*seq_car_alarm*	*sensitive*	202	1.13	1.35	3238	7.23	8.65	2	0.61	2.66
6.	*seq_car_alarm*	*independent*	193	1.12	1.36	4117	7.96	9.96	2	0.15	0.46
7.	*dragon*	*correctness*	115	0.51	0.64	–	–	timeout	65	–	timeout
8.	*switch*	*correctness*	20	0.00	0.02	930	0.26	0.39	3	0.00	0.12
9.	*prod_cons*	*correctness*	30	0.02	0.03	–	–	timeout	1800	–	timeout

also taken an almost sequential model, *seq_car_alarm*, reverse-engineered from a Simulink stateflow model. We use this example to quantify SYMPARA's competitiveness on sequential code. The final three examples, named *dragon*, *switch* and *prod_cons*, have been taken from a set of benchmarks for Lustre (available at https://bitbucket.org/lememta/lustrebenchmarks).

Table I lists the properties checked for these examples. The *correctness* property is with respect to a given specification (for example, at most one process in the critical section, in case of *mutex*). The property of a system to arrive in a stable state within a finite number of steps has been defined as *stability*. The check for *non-determinism* verifies that the system never reaches a state where multiple outgoing transitions are enabled. The properties *independent* and *sensitive* are assertions on program variables, the former independent of the loop in the program, and the latter sensitive to it. All the properties hold for the respective examples, except for *switch*, which is faulty.

Table I also summarizes our results. Apart from the total run-time (in seconds), we also give the time spent by each tool during SAT solving, and the number of SAT calls for SYMPARA and IMPARA. Our experiments were run on a dual-core machine at 2.73 GHz with 2 GB RAM, using a timeout of $900\,s$. The executables and examples are available at http://www.cmi.ac.in/~madhukar/sympara/experiments/. On examples that are either sequential or have a bug (rows 5, 6 and 8), there is little difference between SYMPARA and CBMC (the latter performing better in some cases). The benefits are far more significant in all other cases, where IMPARA and CBMC fail to generate a proof. The race freedom of Lustre benchmarks also accounts for SYMPARA's quick convergence in case of *dragon* and *prod_cons* (rows 7 and 9). The unwind column lists the smallest unwinding for which the tool either timed out, generated a proof or produced a counterexample. The reasons for this performance improvement are: (1) CBMC pessimistically considers all possible schedules, while SYMPARA works with a fixed one. Even if the system is race-free in all its reachable states, k-induction cannot exploit it as the step-case searches from an arbitrary state. (2) The diameter including the violating states is larger than the initialized diameter, which is not a problem for SYMPARA, as it uses forward search. SYMPARA fares better than IMPARA owing to its fixing of an interleaving. Another reason for this efficiency is the result of aggressive eager simplifications added in SYMPARA.

VI. CONCLUSION

In this paper we have proposed a technique tailored for verifying synchronous reactive systems, as an extension of the LAWI algorithm implemented in IMPARA. We also described an implementation of our technique into a tool called SYMPARA.

A unique feature of SYMPARA is that it exploits key features of synchronous parallelism by means of concurrent SSAs, enabling on-the-fly race analysis to prune the search space during symbolic analysis. Our experiments indicate that SYMPARA provides significant performance advantage over CBMC, which generates a monolithic constraint corresponding to the global transition relation. SYMPARA also has the advantage that it is complete, and that it can potentially discover inductive invariants faster than k-induction. When compared to IMPARA, our experiments suggest that restricting synchrony upfront is far more effective in pruning search space than encoding the same via locks, even with powerful partial-order reduction techniques.

REFERENCES

[1] D. Harel, "Statecharts: A visual formalism for complex systems," *Sci. Comput. Program.*, vol. 8, no. 3, pp. 231–274, Jun. 1987.

[2] K. L. McMillan, "Lazy abstraction with interpolants," in *Computer Aided Verification (CAV)*. Springer, 2006, pp. 123–136. [Online]. Available: http://dx.doi.org/10.1007/11817963_14

[3] B. Wachter, D. Kroening, and J. Ouaknine, "Verifying multi-threaded software with Impact," in *Formal Methods in Computer-Aided Design (FMCAD)*, 2013, pp. 210–217.

[4] D. Harel and A. Naamad, "The Statemate semantics of statecharts," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 4, pp. 293–333, Oct. 1996.

[5] Q. Zhao and B. Krogh, "Formal verification of statecharts using finite-state model checkers," *Control Systems Technology, IEEE Transactions on*, vol. 14, no. 5, pp. 943–950, Sept 2006.

[6] T. Bienmüller, W. Damm, and H. Wittke, "The Statemate verification environment – making it real," in *Computer Aided Verification (CAV)*. Springer, 2000, pp. 561–567.

[7] D. Latella, I. Majzik, and M. Massink, "Automatic verification of a behavioural subset of UML statechart diagrams using the Spin model-checker," *Formal Aspects of Computing*, vol. 11, no. 6, pp. 637–664, 1999.

[8] A. Roscoe and Z. Wu, "Verifying Statemate statecharts using CSP and FDR," in *Formal Methods and Software Engineering*, ser. LNCS. Springer, 2006, vol. 4260, pp. 324–341.

[9] A. Cimatti, A. Micheli, I. Narasamdya, and M. Roveri, "Verifying SystemC: A software model checking approach," in *Formal Methods in Computer-Aided Design (FMCAD)*, 2010, pp. 51–60.