An Analyzer for Extended Compositional Process Algebras

Yang Liu
School of Computing
National University of Singapore
liuyang@comp.nus.edu.sg

Jun Sun
School of Computing
National University of Singapore
sunj@comp.nus.edu.sg

Jin Song Dong
School of Computing
National University of Singapore
dongjs@comp.nus.edu.sg

ABSTRACT

System simulation and verification become more demanding as complexity grows. PAT is developed as an interactive system to support composing, simulating and reasoning of process algebra with various extensions like fairness events, global variables and parameterized processes. PAT provides user friendly interfaces to support system modeling and simulation. Furthermore, it embeds two complementing model checking techniques catering for different systems and properties, namely, an explicit on-the-fly model checker which is designed to verify event-based fairness constraints efficiently and a bounded model checker based on state-of-the-art SAT solvers. The model checkers are capable of proving reachability, deadlock-freeness and the full set of Linear Temporal Logic (LTL) properties. Compared to other model checkers, PAT has two key advantages. Firstly, it supports an intuitive annotation of fairness constraints so that it handles large number of fairness constraints efficiently. Secondly, the compositional encoding of system models as SAT problems allows us to handle compositional process algebra effectively. The experimental results show that PAT is capable of verifying systems with large number of states and outperforms the state-of-the-art model checkers in some cases.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specification; D.2.4 [**Software Engineering**]: Program Verification—*Model Checking*

General Terms

Verification

Keywords

Simulation, Model Checking, Fairness, SAT Solvers

1. OVERVIEW

Critical system requirements like safety, liveness and fairness play important roles in system/software specification, verification and development. It is desirable to have handy tools to simulate the system behavior and verify the critical properties, which becomes more demanding as the complexity grows. PAT was started with the investigation of system verification under fairness assumptions based on our previous work in [1]. The motivation is that fairness assumptions are often necessary in system verification practice in order to prove desirable system properties, whereas existing languages and tools have limited support for fairness modeling as well as verification. After that, we went further to explore state-of-the-art verification techniques for more effective system verification. We developed novel techniques and extended PAT with the capability of bounded model checking as well as powerful system simulation. Till now, PAT has been developed as an interactive system which supports system modeling, simulating and verification. Currently, it is publicly available at out web site [5].

PAT has been implemented in C# 2.0 for the benefits of Object-Oriented design and competitive performance. Primary experiments has shown promising results. The current PAT is able to handle systems with large number of states and outperforms the state-of-the-art model checkers (e.g. Spin [3] and FDR [4]) in some cases.

The system models (including both the specification and desired properties) are parsed separately into internal representations. Simulator can take in the internal specification processes for the purpose of simulation. The Büchi automata generated from the LTL properties can also be viewed graphically. The model checkers build the product of the specification and the Büchi automaton (which represents the negation of the property) and conclude true if the product is empty and report with a counterexample otherwise. PAT consists of three main components as follows.

An editor The specification editor provides a user friendly interface for users to input system models (i.e., the full CSP syntax with various extensions is supported) as well as desirable properties (i.e., the full temporal logic syntax is supported).

A simulator The simulator allows users to perform various simulation tasks on the input models: complete states generation based on the execution graph, automatically random simulation, user interactive simulation, trace replay and so on.

Model checkers The model checkers embedded in PAT are event-based on-the-fly model checker with the support of annotated fairness assumptions and the bounded model checker based on the state-of-the-arts SAT-Solvers. They are complementary to each other to cater for various models and properties.

2. MODELING AND SIMULATION

PAT embeds an editor which serves as a user friendly interface for system modeling. The editor accepts extended CSP specifications and LTL properties with the features, like complete text editing functions, syntax highlighting, multi-documents environment and multi-threading execution.

The input language of PAT is based on the classic Communicating Sequential Processes (CSP), for its rich set of process constructs and multi-threaded concurrency. We remark that the simulation and verification algorithms are not limited to specific languages. For better expressiveness, we extended CSP with following features.

Event-based Fairness We introduce the notion of event-based fairness, i.e., a fairness constraint is associated with individual events. In this way, a variety of fairness constraints can be naturally expressed in event-based system models.

Global Variables and Parameterized Processes The process definitions are extended to accept global variables and parameters. To make full use of this extension, we further introduce the process guard and assignment constructs.

Because we are dealing with an event-based formalism, we extend standard LTL with events so that properties concerning both states and events can be stated and verified. The complete grammar rules of extended CSP and LTL supported by PAT are available at our web site [5].

System Simulator takes in the specifications and allows users to perform various simulation tasks including complete states generation based on the execution graph, automatically random simulation with animation, Execution trace display and replay and image printing and exporting functions.

3. VERIFICATION

PAT embeds two complementing model checkers catering for different systems and properties. Instead of presenting the technical details of two model checking algorithms, we will use a running example to illustrate the intuition behind the event-based fairness as well as the algorithms. The following is the model of the classic dining philosophers problem [2],

$$\begin{aligned}
\mathit{Phil}_i &= \mathit{think.i} \rightarrow \mathit{get.i.}(i+1)\%N \rightarrow \mathit{get.i.i} \rightarrow \\
&\quad \mathit{eat.i} \rightarrow \mathit{put.i.}(i+1)\%N \rightarrow \mathit{put.i.i} \rightarrow \mathit{Phil}_i \\
\mathit{Fork}_i &= \mathit{get.}((i-1)\%N).i \rightarrow \mathit{put.}((i-1)\%N).i \\
&\quad \rightarrow \mathit{Fork}_i \;\Box\; \mathit{get.i.i} \rightarrow \mathit{put.i.i} \rightarrow \mathit{Fork}_i \\
\mathit{Phils}_N &= \Big\|_{i=0}^{N-1} (\mathit{Phil}_i \parallel \mathit{Fork}_i)
\end{aligned}$$

where N is the number of philosophers, $get.i.j$ ($put.i.j$) is the action of the i-th philosopher picking up (putting down) the j-th fork. The property to verify is $\bigwedge_{i=0}^{N-1} \Box\Diamond eat.i$ which informally means that no philosopher would starve.

3.1 On-the-fly Model Checking

The explicit on-the-fly model checker constructs the product of the system and the Büchi automaton generated from the negation of the property on-the-fly and concludes with a counterexample as soon as a fair loop is discovered. If the specification is not annotated with fairness constraints, a fair loop means a loop which contains at least one fair state of the Büchi automaton. Our model checker uses a DFS-search algorithm to find one fair loop efficiently, e.g., verifying Phils_N against the property would return the trace $\langle \mathit{think.i}, \mathit{get.i.}(i+1)\%N \rightarrow \mathit{get.i.i} \rightarrow \mathit{eat.i} \rightarrow \mathit{put.i.}(i+1)\%N \rightarrow \mathit{put.i.i}\rangle$, i.e., one of the philosophers keeps eating and thus the rest starve. If the system is fair, i.e., every philosopher gets a chance to grab a fork, then non-starvation is guaranteed.

We go beyond existing model checkers like Spin or FDR to verify systems annotated with event-based fairness. The key idea of event-based fairness is that fairness constraints (weak or strong) is associated with individual events by annotating the event using $wl(event)$ or $sl(event)$. It may be the case that only a subset of the events is associated with fairness constraints, e.g., for a vending machine, it is natural to require eventually an item is dispatched after some coins have been inserted whereas it is unnatural to require that always eventually some coins will be inserted. Event-based fairness is extremely flexible. The explicit model checking algorithm is extended to take the fairness constraints into account. For systems with fairness annotations, a fair loop is a loop which not only contains at least one fair state of the Büchi automaton, but also is fair regarding the event-based fairness. We skip the semantics of event-based fairness and the extended model checking algorithm for brevity. In summary, a variety of fairness constraints can be naturally embedded in the model itself and the algorithm handles event-based fairness efficiently.

3.2 Bounded Model Checking

Bounded model checking has been shown to be a complementary approach for system verification. The idea is to encode the verification problem as a SAT problem and then apply state-of-the-art SAT solvers to conclude whether the property is true or false. PAT embeds a bounded model checker to complement the explicit model checker, i.e., to produce the shortest counterexample if there is one or to produce a counterexample if the explicit model checking is simply infeasible (due to time or memory limit). The challenging aspect of applying bounded model checking to compositional process algebra is that the encoding must be compositional and must avoid the building of the explicit state graph (which is destined to be huge). Given a number of processes running in parallel (say $\|_{i=0}^{N} P_i$), the compositional encoding (that has been implemented in PAT) encodes each sub-component P_i first and then composes the encoding of P_i to build the encoding of the composition. For instance, $\mathit{Phil}_i \parallel \mathit{Fork}_i$ is encoded first and then the encoding is manipulated so that the encoding of Phils_N is exactly the conjunction of the encoding of $\mathit{Phil}_i \parallel \mathit{Fork}_i$. This way, we avoid exploring the state space of Phils_N. If P_i is also an indexed parallel composition of multiple sub-processes (which is not possible in Spin but perfectly reasonable in process algebras like CSP or CCS), the same procedure is repeated.

4. CONCLUSION

In summary, we have developed a self-contained tool PAT for system specification and verification based on an event-based compositional processes. Experiment results show that PAT does verification rather efficiently. The future plan of PAT includes process refinement/equivalence checking, optimization techniques like partial order reduction as well as model checker based on BDD.

5. REFERENCES

[1] J. S. Dong, P. Hao, J. Sun, and X. Zhang. A Reasoning Method for Timed CSP Based on Constraint Solving. In *Proc. of the 8th Inter. Conf. on Formal Engineering Methods (ICFEM 2006)*, pages 342–359. Springer, 2006.

[2] C. A. R. Hoare. *Communicating Sequential Processes*. Inte. Series in Computer Science. Prentice-Hall, 1985.

[3] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engeering*, 23(5):279–295, 1997.

[4] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[5] J. Sun, Y. Liu, and J. S. Dong. A Simulator and Model Checker for Extended CSP. http://www.comp.nus.edu.sg/~liuyang/pat/, 2008.