Analyzing Hierarchical Complex Real-time Systems*

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

Jun Sun
Singapore University of Technology and Design
sunjun@sutd.edu.sg

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

ABSTRACT

Specification and verification of real-time systems are important research topics which have practical implications. In this work, we present a self-contained toolkit to analyze real-time systems, which supports system modeling, animated simulation and automatic verification (based on advanced model checking techniques like dynamic zone abstraction). In this tool, we adopt an event-based modeling language for describing real-time systems with hierarchical structure. Experiments show that our tool has compatible performance with the state-of-the-art verifiers, and complement them with additional capabilities like LTL model checking, refinement checking.

1. OVERVIEW AND SYSTEM DESIGN

Ensuring the correctness of life-critical applications is crucial and challenging. This is especially true when the correctness of such systems depends on quantitative timing. The state-of-the-art approach for specifying real-time systems is based on the notation Timed Automata (TA) [1]. TA often have a flat structure, e.g. a network of TA with no hierarchy, which makes the efficient model checking feasible. Nonetheless, designing and verifying compositional real-time systems is becoming an increasingly difficult task. High-level requirements for real-time systems are often stated in terms of *deadline*, *time out*, and *timed interrupt*. Unlike statecharts with clocks or timed process algebras, TA lack these compositional patterns. As a result, users often need to manually cast those terms into a set of clock variables with carefully calculated clock constraints. The process is tedious and error-prone.

To solve this problem, we proposed an alternative approach [6] for modeling and verifying hierarchical real-time systems. Based on process algebra, our modeling language introduces a rich set of concurrent operators and compositional timed behavioral patterns like $deadline$, $within$, $timed\ interrupt$, etc. Instead of explicitly manipulating clock variables (as in TA), the timed patterns are designed to build on implicit clocks. Further, we augment a system model with mutable variables and data structures (e.g., arrays, stacks or arbitrary data types), synchronous or asynchronous channels, etc. Our proposed language adopts a dense-time semantics, where the clock values are rational numbers. Hence, there may be infinitely many transitions between any two time points. To offer efficient verification support, a fully automated abstraction technique is developed to build an abstract finite state system from the (infinite) model. We show that the abstraction has finite state and is subject to model checking. Further, it weakly bi-simulates the concrete model and, therefore, we may perform sound and complete LTL model checking, refinement checking upon the abstraction.

Our engineering efforts realize the proposed techniques into a self-contained toolkit for analyzing real-time systems, which is built as Real-Time System (RTS) module in our home grown model checker PAT [5] (freely available at http://pat.comp.nus.edu.sg). Fig. 1 shows the architecture design of our toolkit with four components. The editor (see Fig. 4) is featured with powerful text editing, syntax highlighting and multi-documents environment. The parser compiles the system models and the properties into internal representation. Abstraction (see Section 3) is applied during the compilation of a model so that a finite state abstract model is yielded internally. The simulator (see Fig. 5) allows users to perform various simulation tasks on the models: complete states generation of execution graph, automatic simulation, user interactive simulation, trace replay and etc. The simulator is also used to visualize Büchi automata generated from the negation of LTL assertions. Most importantly, we implement several verifiers catering for checking deadlock-freeness, reachability, LTL properties with fairness assumptions, refinement checking and etc. To achieve good performance, advanced optimization techniques are implemented, e.g., partial order reduction, process counter abstraction, parallel model checking, etc. All the verification algorithms perform on-the-fly exploration of the state space. If any counterexample is identified during the exploration, then it can be animated in the simulator for the purpose of debugging.

2. REAL-TIME SYSTEM MODELING

In RTS module, a system model is composed of multiple elements, i.e. constants, global variables/channels, a set of timed process definitions, a set of assertions, etc. The process definitions identify the computational logic of a system. A timed process P (hereafter process) can be defined using a rich set of process constructs (see [6] for details). Furthermore, a number of timed process constructs can be used to capture common real-time system behavior patterns. For example, let d be an rational number. Process $Wait[d]$ idles for d time units. In process $P\ timeout[d]\ Q$, the first observable event of P shall occur before d time units elapse. Otherwise, Q takes over control after exactly d time units elapse. In real-time systems, requirements are often structured into phases,

*This research was partially supported by a grant "SRG ISTD 2010 001" from Singapore University of Technology and Design.

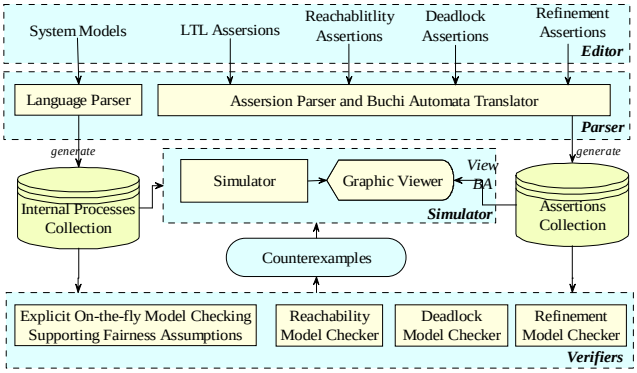

Figure 1: Architecture Design

which is hierarchical in nature (e.g., pacemaker). Our language is hierarchical and uses implicit clocks, hence the modeling process is much simpler without complicated clock calculations. The complete language syntax can be found in PAT's user manual. The formal operational semantics can be found in [6].

3. ABSTRACTION AND VERIFICATION

Model checking requires a finite state system model. Hence, we assume that all variables have finite domains and the process forbids unbounded non-tail recursion. However, the number of system states (and hence the transition system) is still infinite because of our dense-time semantics. We propose a zone abstraction [6] to build an abstract system. Different from zone abstraction applied to TA [3], we dynamically create/delete a set of clocks to precisely encode the timing requirements. A *zone* is the maximal set of clock valuations satisfying a set of primitive clock constraints. A primitive constraint on a clock is of the form $tm \sim d$ where tm is a timer, d is a constant and $\sim$ is $\geq$, $=$, or $\leq$. Because clocks are implicit, clock readings cannot be compared directly. In order to support efficient verification, we use difference bound matrices (DBM) [3] as an equivalent representation for the zone.

To perform verification on the original systems, we need to show the abstract transition system is equivalent to the original transition system. We show our zone abstraction is sound and complete with respect to the following three properties using a specialized bi-simulation relationship [6].

LTL Model Checking In this setting, the properties are linear temporal logic (LTL) formulae, constituted by propositions on global variables and events. Notice that no clocks are allowed in the property. In order to reflect model checking results on the abstract transition system to the original system with respect to LTL formulae, we show stutter equivalence between traces of the abstract system and the original system [6]. To verify the LTL formulae, we adopt the automata-based on-the-fly verification algorithm [5], i.e., by firstly translating a formula to a Büchi automaton and then check emptiness of the product of the system and the automaton.

Refinement Checking In this setting, we investigate an alternative verification schema. That is, to verify whether the system satisfies the property by showing a refinement relationship between the system and a model which models the property. In order to check refinement between two (timed) models, zone abstraction must be applied to both models. In [6], we prove that it is sound and complete to show stable failures refinement between the two abstraction transition systems in order to show failures refinement between the two corresponding original models. The refinement relationship is verified using an on-the-fly simulation checking approach.

Timed Refinement Checking We have looked at the refinement checking without timed transitions. Specification of practical systems, however, may be complicated. For instance, the following is requirement from the pacemaker specification [2]. "The first Pace-Now pacing pulse shall be issued within two cardiac cycles plus 500 ms from the time of the last user action required to activate the Pace-Now state". To support refinement checking with timing aspects, we introduce time stamps in the traces. We assume a global clock t_G which starts when the system starts. We extend the algorithm for un-timed refinement checking with the synchronization of time stamps of the specification and implementation [4].

4. EXPERIMENTS AND DISCUSSION

Table 1 shows the experiment results on a pacemaker system [2], Fischer's algorithm and a railway control system. Using refinement relationship, we can encode a variety of different properties, including mutual exclusion, bounded by-pass, etc. The experiment on Fischer's mutual exclusion algorithm against bounded by-pass shows that the timed refinement checking algorithm in PAT finds a counterexample quickly. It is time consuming if a system contains multiple concurrent processes and the property is true. PAT can handles 10^7 states within an hour which is comparable to model checkers like SPIN and UPPAAL. The data on UPPAAL or RED verifying the same models have similar results, hence omitted.

Our work is related to a number of automatic verification supports for TA, including UPPAAL, KRONOS, RED, Timed COSPAN and Rabbit. Different from the TA approach, our model checker is the first dedicated verification tool support for hierarchical real-time systems by adapting advanced verification techniques. In addition, PAT complements UPPAAL with the ability to check full LTL properties and refinement relationship. For timed refinement checking, there is no tool support to the best of our knowledge. The reason is that it has been proved that the refinement checking (or equivalently the language inclusion) problem in the setting of TA [1] is undecidable. Other negative results include that TA cannot be determinized. We show that timed refinement checking is decidable in our setting [4].

Starting from 2008, RTS Module in PAT has come to a stable stage with solid testing and 60 built-in examples. It has been applied to verify many real-time systems ranging from classical concurrent algorithms to real world problems. Many institutions uses PAT as a research or educational tool. Currently, there are 800 registered users from 168 organizations in 32 countries. Our future works include optimization techniques using fast DBM and state reduction techniques like symmetry reduction.

5. REFERENCES

[1] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[2] S. S. Barold, R. X. Stroopbandt, and A. F. Sinnaeve. *Cardiac Pacemakers Step by Step: an Illustrated Guide*. Blachwell Publishing, 2004.

[3] D. L. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In *Automatic Verification Methods for Finite State Systems*, pages 197–212, 1989.

[4] Y. Liu. *Model Checking Concurrent and Real-time Systems: the PAT Approach*. PhD thesis, National University of Singapore, 2009.

[5] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *CAV*, pages 702–708, 2009.

[6] J. Sun, Y. Liu, J. S. Dong, and H. H. Wang. Verifying Stateful Timed CSP using Implicit Clocks and Zone Abstraction. In *ICFEM 2009*, pages 581–600, Dec 2009.

APPENDIX

A. DEMO SCRIPT

We will demonstrate our toolkit step by step as follows. First, we will give an overview of this tool and its architecture design. Second, we will introduce the modeling languages for the compositional real-time systems. In the third and forth parts, we would like to present the zone abstraction and verification algorithms. Follow that, we will conduct the demonstrate to illustrate modeling languages and the functionalities (model composition, simulation and verification). Finally, we will discuss some experiment results and possible future works. The pacemaker example and Fischer's mutual exclusion algorithm will be used as running examples to illustrate the ideas. Furthermore, we will also demonstrate the pacemaker system for the timed refinement checking.

A.1 Overview of RTS module in PAT

RTS module in PAT is a self-contained toolkit to support composing, simulating and reasoning of compositional real-time systems. The architecture diagram presented in Fig. 1 will be presented first to explain the design and functionality.

A.2 Real-time System Modeling

The syntax and semantics of the proposed language will be introduced. Language constructs will be listed as follows.

$e \rightarrow P$	– event prefixing and P is a process
$P \mid Q$	– general choice
$P \parallel Q$	– parallel composition
$P;\ Q$	– sequential composition
$P \mathrel{\widehat{=}} Q$	– process definition
$Wait[d]$	– delay
$P\ timeout[d]\ Q$	– timeout
$P\ interrupt[d]\ Q$	– timed interrupt
$P\ waituntil[d]$	– wait until
$P\ deadline[d]$	– deadline
$P \mathrel{\widehat{=}} Q$	– process definition

To illustrate the syntax, we use the Fischer's algorithm (following) and a pacemaker system (Fig. 2) to demonstrate the hierarchical modeling support of the proposed language.

$$
\begin{aligned}
&\textbf{var } x && = -1;\\
&\textbf{var } ct && = 0;\\
&Proc(i) && = [x == -1]Active(i)\\
&Active(i) && = (update.i\{x = i\} \rightarrow Skip)\ \textbf{deadline}[\delta];\\
& && \quad \textbf{Wait}[\epsilon];\\
& && \quad \textbf{if } (x == i)\ \{\\
& && \qquad cs.i\{ct++\} \rightarrow\\
& && \qquad exit.i\{ct--;\ x = -1\} \rightarrow Proc(i)\\
& && \quad \}\ \textbf{else}\ \{Proc(i)\}\\
&Protocol && = Proc(0) \parallel Proc(1) \parallel \ldots \parallel Proc(N);
\end{aligned}
$$

A.3 Zone Abstraction

We will use example to illustrate the zone abstraction and abstract system. Assume a model $(\varnothing, \varnothing, P)$ with no variable and P is $(a \rightarrow Wait[5];\ b \rightarrow Stop)\ interrupt[3]\ c \rightarrow Stop$. The abstract transition system is shown in Figure 3, where transition label τ is skipped for simplicity. The construction and changes of the DBM of the process will be explained.

Figure 3: A simple example

A.4 Property Verification

We will explain the supported properties and verification algorithms developed using the Fischer's mutual exclusion algorithm and the pacemaker program.

- Deadlock and reachability checking for orchestration.

  $$\#assert\ Protocol\ deadlockfree;$$

- LTL properties.

  $$\#assert\ Protocol \models \Box(request \Rightarrow \Diamond accessCS);$$
  $$\#assert\ Protocol \models \Box(update \Rightarrow \Diamond cs);$$

- Refinement checking algorithm.

  $$\#assert\ Protocol\ refines\ Untimed\_Protocol;$$

- Timed Refinement checking algorithm.

  $$\#assert\ Sys\_AAT\ refines < T > \ Spec\_AAT;$$

A.5 Demonstration

A.5.1 Specification Editor

First we will show the specification editor (see Fig. 4) using the Fischer's mutual exclusion algorithm.

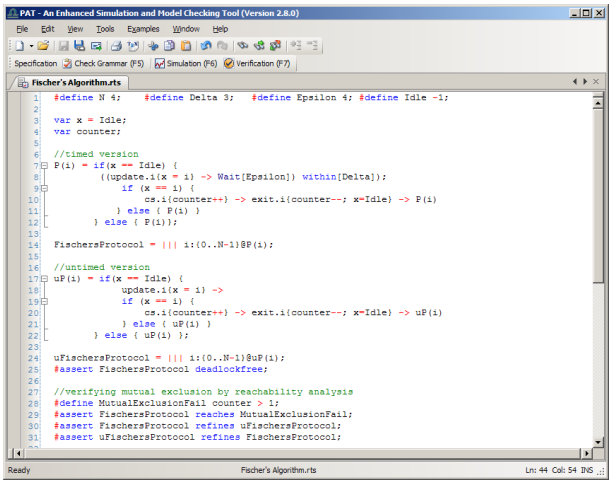

Figure 4: Main Window of PAT with Fischer's Algorithm

A.5.2 Simulator

We will illustrate the simulator (Fig. 5) using the previous loaded example. Firstly, we will show the complete states graph generated based on the execution. Secondly, we will play the animation of automatically random simulation. Thirdly, we will show the how the step-by-step user guided simulation is conducted. Finally, we will demonstrate the functions of execution trace display and replay.

$$
\begin{array}{ll}
\textbf{var}\ SA & = 0;\ //variable\ definition \\
AAT & = Heart\ ||\ Sensing\ ||\ Pacing(LRI); \\
Sensing & = \textbf{if}\ (SA = 1)\ \{\ \textbf{atomic}\{pulseA \rightarrow senseA \rightarrow Skip\};\ Sensing\ \}\ \textbf{else}\ \{\ pulseA \rightarrow Sensing\}; \\
Pacing(X) & = (\textbf{atomic}\{senseA \rightarrow paceA\{SA = 0\} \rightarrow Skip\}\ \textbf{timeout}[X]\ (paceA\{SA := 0\} \rightarrow Skip)\ \textbf{within}[0]); \\
 & \quad \textbf{Wait}[URI];\ (enableSA\{SA := 1\} \rightarrow Pacing(LRI - URI))\ \textbf{within}[0];
\end{array}
$$

Figure 2: A model of the AAT mode of a pacemaker

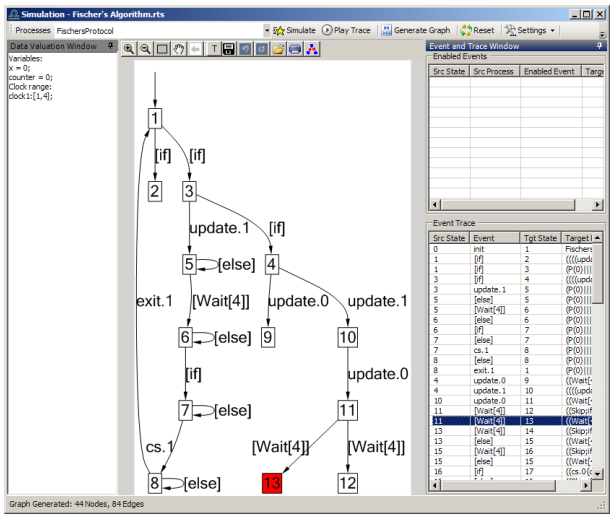

Figure 5: PAT Simulator User Interface

A.5.3 Verification

We will verify properties in the Fischer's mutual exclusion algorithm to illustrate the verification support (see Fig. 6).

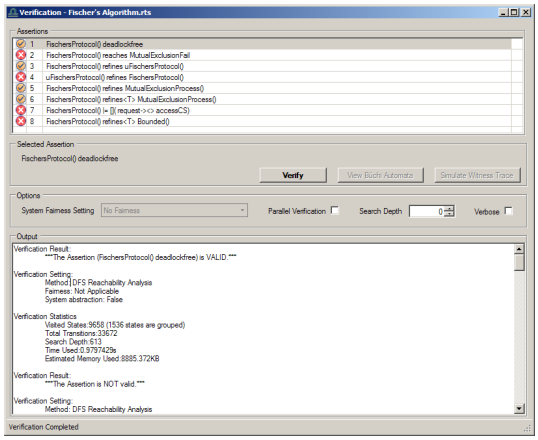

Figure 6: PAT Verification User Interface

A.6 Experiments

Table 1 shows the experiment results on pacemaker system [2], Fischer's algorithm and a railway control system. The data are obtained with Intel Core 2 Quad 9550 CPU at 2.83GHz and 2GB memory. The pacemaker, though complicated, contains little concurrency and hence is verified efficiently. Using refinement relationship, we can encode a variety of different properties, including mutual exclusion, bounded by-pass, etc. The experiment on Fischer's mutual exclusion algorithm against bounded by-pass shows that the timed refinement checking algorithm in PAT finds a counterexample quickly. It is time consuming if a system contains multiple concurrent processes and the property is true. Further, a simple experiment shows that the computational overhead of calculating clocks/DBMs is around one third of the overall time. PAT typically handles 10^7 states within an hour which is comparable to model checkers like SPIN and UPPAAL.

Model	Size	Property	States	PAT (s)
Pacemaker	-	deadlock-free	302442	92.1
Pacemaker	-	correctness	986342	122
Fischer	5	$P1$	26496	2.49
Fischer	6	$P1$	207856	27.7
Fischer	7	$P1$	1620194	303
Fischer	4	$P2$	5835	0.53
Fischer	5	$P2$	49907	5.83
Fischer	6	$P2$	384763	70.5
Fischer	4	mutual exclusion	9941	0.78
Fischer	5	mutual exclusion	141963	17.2
Fischer	6	mutual exclusion	2144610	401
Fischer	7	bounded bypass	9213	1.47
Fischer	9	bounded bypass	91665	21.1
Fischer	11	bounded bypass	693606	214
Fischer	4	$Prot\ refines\ uProt$	7741	5.22
Fischer	5	$Prot\ refines\ uProt$	72140	126.3
Fischer	6	$Prot\ refines\ uProt$	705171	3146
Railway Control	4	deadlock-free	853	0.11
Railway Control	6	deadlock-free	27787	3.07
Railway Control	8	deadlock-free	1563177	223.1
Railway Control	5	$\Box(appr \rightarrow \Diamond leave)$	8137	0.95
Railway Control	6	$\Box(appr \rightarrow \Diamond leave)$	50458	6.58
Railway Control	7	$\Box(appr \rightarrow \Diamond leave)$	359335	58.63
Railway Control	5	bounded waiting	4764	3.21
Railway Control	6	bounded waiting	28782	26.2
Railway Control	7	bounded waiting	201444	238

Table 1: Experiment results of RTS Module

A.7 Conclusion and Future Works

In the end, we would like to briefly talk about the related work and future research directions.