Testing Component-Based Real Time Systems

Rachid Bouaziz, Ismail Berrada

Testing Component-Based Real Time Systems*

Rachid Bouaziz [1], Ismaïl Berrada [2]

[1] University of Toulouse - CNRS - IRIT
31062 Toulouse Cedex - France
bouaziz@irit.fr
[2] L3I, Université de La Rochelle

iberrada@univ-lr.fr

Abstract

This paper focuses on studying efficient solutions for modeling and deriving compositional tests for component-based real-time systems. In this work, we propose a coherent framework that does not require the computation of the synchronous product (composition) of components, and therefore avoids a major bottleneck in this class of test. For this framework, we introduce an approach and associated algorithm. In our approach, the overall behavior of the system is obtained by restricting free runs of components to those involving interactions between them. This restriction is achieved through the use of a particular component called assembly controller. For the generation algorithm, compositional test cases are derived from the assembly controller model using symbolic analysis. This reduces the state space size (a practical size) and enables the generation of sequences which cover all critical interaction scenarios.

1 Introduction

Model-based testing is one of the most used formal techniques for the validation of software/hardware systems. It consists in applying a set of experiments (test cases), derived from the formal model of the specification, to a system with the intent of finding errors. Model-based testing offers the possibility of reducing the test efforts and enables systematic selection of test cases. However, modern dynamically evolving systems provide challenging problem solving activities for the testing community. In modeling activity, the problem to solve is how to adequately model the behavior of these complex systems in order to test them. Test generation and execution activity deals with the definition of efficient instantiation algorithms of test cases (based on the system model) and test cases execution against the system under test (SUT).

Real-time systems are computer systems in which the correctness of the systems depends not only on the logical correctness of the computation performed but also upon time factors. So, when testing such systems, the tester must pay attention to the correctness of outputs produced by the SUT as well as the correctness of the corresponding timing. Most of complex real-time systems are component-based systems. A component is a non-trivial, nearly independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. In other words, each component of a component-based system performs some defined tasks and interacts with other components in a specified way, to achieve a global function of the system. The main advantage of component-based real-time systems (CBRTS) is the possibility of reusing high-quality components provided by professional vendors.

Analysis and testing of components developed externally suffer from lack of information related to intra-component interactions. Often, the overall behavior of the composite system is given by the parallel composition of components (the synchronous product operation). As a result, internal communications are hidden to the tester. Besides, the state space size of the composite system is the product of the state space size of each component, and may grow exponentially with the number of interacting components. Consequently, deriving test directly from the composite system may lead to a large and intractable number of test cases.

1.1 Contributions of the paper

The main contribution of this paper is the introduction of a practical framework for testing CBRTS. Our framework is based on intra-component communication modeling, dead-

*This research has been supported by the european Marie Curie RTN TAROT project (MCRTN 505121).

lock detection, and compositional test cases generation using symbolic approaches.

- Intra-component communication modeling. In order to avoid the computation of the composite system, we clearly separate the description of individual behaviors of components from the way they interact. Interactions between system components are supervised by a particular component called *the assembly controller*. Our idea is inspired from supervisory control theory of discrete dynamical systems [15, 12, 3]. In fact, from the behavioral point of view, intra-component communications are considered as a restriction of overall behavior of the composite system. Such restriction is viewed as a controller that forces the system to operate within a desired region of the system's state-space. Assembly controller is also used to integrate the system components and to ensure the correctness of the composed system. Thus, as a first, contribution, we show how to construct an *optimal* and *non-blocking* assembly controller used to integrate components and to ensure the correctness of the composed system.

- Deadlock detection and composability checking. Composing a component-based application from components that are not specifically designed for the individual application poses a number of problems. Blocking (Deadlock) is one of the major problems in CBRTS. It corresponds to the situation when the system reaches a region in the state space where it cannot exit from. Deadlock can be due to synchronization conflicts and to temporal incoherence between shared events. Often, detecting deadlock requires exhaustive exploration of the system's state space. This is obviously inefficient while dealing with complex real-time systems. As a second contribution, we show that composability and deadlock problems can be solved by checking if there exists a non-blocking assembly controller who ensures a correct interaction between real time components.

- Compositional test cases generation. In cas there exists such a non-blocking assembly controller, then it will cover all (critical) interaction scenarios. Thus, as a third contribution, we show how efficient compositional test cases can be derived from the assembly controller using symbolic analysis. Compositional test cases are generated directly from the assembly controller model or indirectly by combining test cases derived from the *restricted behavior* of each individual supervised component.

Thus, the originality of our work is the proposition of a practical framework for compositional test case generation based on explicit modeling of intra-component interactions though the assembly control.

1.2 Organization of the paper

The remainder of the paper is organized as follows: Section 2 presents the model of timed input/output automata. Section 3 is related to assembly controller synthesis. Section 4 shows how compositional test cases are generated and selected automatically from a symbolic abstraction of TIOA. Finally, we conclude and draw some perspectives in section 5.

2 Timed Input/Output Automata

In this section, we present the model of Timed Input/Output Automata (TIOA) used to describe CBRTS. For reader not familiar with TIOA notations, a short description is given here, and more complete one can be found in [1].

Let $\mathbb{R}^+$ be a set of nonnegative real numbers, X be a finite set of nonnegative real-valued variables called *clocks*, and $G(X)$ be the set of all time guards over X defined by the following grammar $g := x \sim c \mid x - y \sim c \mid g \wedge g \mid$ *true*, where $(x, y) \in X^2$, $c \in \mathbb{R}^+$ and $\sim \in \{<, \leq, =, >, \geq\}$. Let Σ be a finite set of *actions*. A *timed sequence* $w = (a_1, d_1)(a_2, d_2)\ldots(a_n, d_n)$ is an element of $(\Sigma \times \mathbb{R}^+)^*$ such that $d_1 \leq d_2 \leq \cdots \leq d_n$. A *clock valuation* is a mapping $\nu : X \to \mathbb{R}^+$. Let $d \in \mathbb{R}^+$, $r \subseteq X$, and ν be a valuation. Then, $\nu + d$ and $\nu[r := 0]$ are defined respectively by :

- for all $x \in X$, $(\nu + d)(x) = \nu(x) + d$.
- for all $x \in X \setminus r$, $\nu[r := 0](x) = \nu(x)$, and for all $x \in r$, $\nu[r := 0](x) = 0$.

Definition 1 *Timed Input Output Automaton* (TIOA) is a tuple $A = (L, l_0, X, \Sigma, E, I)$, where

- L is a finite set of locations
- $l_0 \in L$ is the initial location
- X is a finite set of clocks
- $\Sigma = \Sigma^! \cup \Sigma^?$ is a finite set of input and output events.
- $I : L \to G(X)$ is a function that assigns an invariant to a location.
- $E \subseteq L \times G(X) \times \Sigma \times 2^X \times L$ is a set of edges. $t = (l, g, a, r, l') \in E$, written $l \xrightarrow{g,a,r} l'$, is a transition of source l and destination l', associated with occurrence of a, guarded by $g \in G(X)$. r is the set of clocks to be reset during the transition.

The semantics of a TIOA A is defined by associating a labeled transition system $S(A)$. A state of $S(A)$ is a couple (l, ν) such that l is a location and ν is a valuation over X. There are two types of transitions in $S(A)$:

- *Time transitions*: for a state (l, ν) and $d \in \mathbb{R}^+$, $(l, \nu) \xrightarrow{d} (l, \nu + d)$ if for all $0 \leq d' \leq d$, $\nu + d' \models I(l)$.
- *Discrete transitions*: for a state (l, ν) and a transition $l \xrightarrow{g,a,r} l'$, $(l, \nu) \xrightarrow{a} (l', \nu[r := 0])$ if $\nu \models g$ and $\nu[r := 0] \models I(l')$.

A TIOA A is said to be:

- input-complete, if it accepts any input action at any time.
- deterministic, if $S(A)$ is deterministic (semantics automaton)
- non-blocking, if in every state, an action transition (output or delay) will eventually become firable. Thus, for each state s of $S(A)$, there exists a state s' and an event $a \in \Sigma^! \cup \mathbb{R}^+$ such that $s \xrightarrow{a} s'$.

Finally, A *Path* P in A is a finite sequence of consecutive transitions $l_0 \xrightarrow{g_1,\alpha_1,r_1} l_1 \xrightarrow{g_2,\alpha_2,r_2} l_2 \ldots$. A *Run* of A over P is a sequence of the form $(l_0, \nu_0) \xrightarrow{(\alpha_1,d_1)} (l_1, \nu_1) \xrightarrow{(\alpha_2,d_2)} (l_2, \nu_2) \ldots$, where $d_i \in \mathbb{R}^+$, and ν_i is a clock valuation satisfying the following requirements:

- $\nu_0(x) = 0, \forall x \in X$,
- For all $i \in [1, n]$, $\nu_{i-1} + (d_i - d_{i-1}) \models g_i$, $\nu_i \models I(l_i)$ (with $d_0 = 0$)
- ν_i is equals to $(\nu_{i-1} + (d_i - d_{i-1}))[r_i := 0]$.

The timed sequence associated to this run is $\omega = (\alpha_1, t_1)(\alpha_2, t_2) \ldots (\alpha_n, t_n)$ where $t_1 \leq t_2 \leq \cdots \leq t_n$.

Example 1

Figure 2 illustrates an example of a CBRTS composed of two components C_1 and C_2. Events i_1, i_2, i_3 are synchronization events between C_1 and C_2. In this example, we can see some temporal incoherences between $C1$ and $C2$. In fact, in location l_0, if C_1 stays more than 1 time unit, then C_2 well be blocked (transition $m_0 \xrightarrow{z<1,!i_1,\{z,w\}} m_1$ in C_2). Moreover, assume that C_1 and C_2 are able to synchronize on i_1. Again, in location m_3, if C_2 stays more than 1 time unit, then C_1 well be blocked (transition $l_2 \xrightarrow{x\leq 1,?i_2} l_3$ in C_1). We can notice that if C_1 and C_2 synchronize with the respect to the controller behavior defined in Figure 2, no deadlock will occurs. The next section, we will show that there exists an assembly controller that ensures correct interaction between C_1 and C_2, and how it can be synthesized.

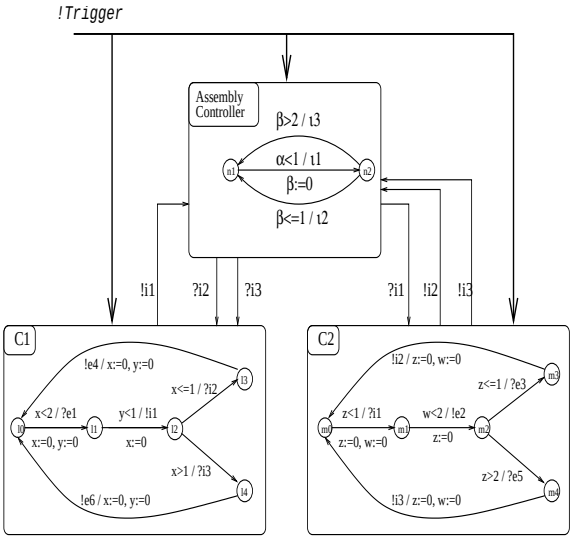

Figure 1. Example of CBRTS

3 Assembly controller synthesis

Even a simple timed automaton generates a labeled transition system with infinitely many reachable states. Thus, algorithmic verification and testing rely on the existence of exact finite abstractions. An efficient abstraction of the state-space for timed automata is based on the notion of zone [1]. A zone is the solution set of a clock constraint, that is the maximal set of clock assignments satisfying the constraint. zones are used to denote symbolic states. It is well-known that zones can be efficiently represented and stored in memory as DBMs (Difference Bound Matrices) [5]. DBMs offer the possibility of implementing operations over symbolic states in a simple and efficient way.

Testing CBRTS requires the exploration of the entire state space of the system. As the number of test cases generated may grow exponentially with the number of interacting components, we clearly separate the individual behavior of components from the way they interact (synchronizations). Thanks to the assembly controller, only relevant behaviors related to intra-component synchronizations will be tested. Thereafter, we give details for synthesizing an *optimal* and *non-blocking* assembly controller.

Definition 2

An assembly controller is a particular TIOA used to restrict the overall behavior of the composite system in order to ensure a correct interaction between components. It can:

- Authorizes or forbids the occurrence of some shared events according to the current state of the composite system.

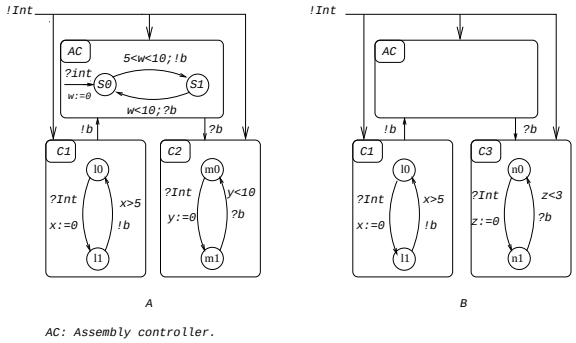

Figure 2. Checking for composability

- Forces components to follow desired paths. This can be achieved by restricting temporal constraints in which some shared events -output events- must be executed.

Example 2
Let us consider Figure 2. In this figure, C_1 can produce event b in interval $]5, \infty[$, and C_2 can consume the same event in interval $[0, 10[$. When C_1 sends b exactly at instant 10 after initialization, interaction between C_1 and C_2 will fail. Thus, assembly controller must restrict the behavior of C_1 by forcing b to be sent no after 10 time units. Now, if we consider C_3, we can notice that there is no controller which can ensure a correct interaction between C_1 and C_3, because the conjunction of temporal constraints related to the emission and the reception of b is empty. Next, we show how to check the composability of components and how to synthesize a non-blocking and optimal assembly controller.

Assembly controller synthesis

In our framework, the aim of an assembly controller synthesis is to limit intra-component behaviors to meet the global objective of the composite system. Restricting intra-component behaviors consists in modifying the time constraints associated with some shared events. The new temporal constraints force the composite system to follow some predefined paths in order to avoid blocking situations and synchronization conflicts.

Assembly controller is said to be optimal if the restriction applied to each component is the less constraining one that achieves correctness of the composite system. To guarantee the optimality, we compute new temporal constraints using symbolic analysis based on zones. The main lines of the algorithm for constructing an optimal and non blocking assembly controller are the following:

1. Identification of *potential* blocking states. This can be achieved by examining shared events. A potential blocking state can only be reached by executing an input/output shared event.

2. Computation of clock valuations from which the potential blocking state can be reached by performing a shared event e. This step can be done using backward reachability. Let $t = l_e \xrightarrow{g,e,r} l_p$ be a transition that leads to the potential blocking state l_p, and (l_p, Z_p) be a symbolic state associated to l_p (Z_p is a zone). Then, the predecessor of (l_p, Z_p) by t is the symbolic state (l_e, Z_e) such that :

$$(l_e, Z_p) = (l_e, (([r := 0]Z_b \cap g)\downarrow) \cap I(l_e))$$

with,

$$[r := 0]Z = \{\nu \,|\nu[r := 0] \models Z\} \qquad Z^{\downarrow} = \{\nu - d \,|\, \nu \models Z, d$$

   - $[r := 0]Z_b$ are clock valuations just before executing the shared event
   - $[r := 0]Z_b \cap g$ are clock valuations from which the shared event e can be performed.
   - $(([r := 0]Z_b \cap g)\downarrow) \cap I(l_e)$ are clock valuations reached by the passage of time in location l_e.

3. Computation of new temporal constraints that allow shared output events be sent while avoiding blocking situations. This can be achieved by analyzing clock valuations, computed in step 2, as following:

   Let $Z_{!e}$ (resp. $Z_{?e}$) be the clock valuations corresponding to the emission (resp. reception) of e. $Z^{new}_{!e}$ is the new temporal constraints of $!e$.

   - First, we compute $Z_1 = Z_{!e} \cap Z_{?e}$
   - If $Z_1 = \emptyset$ then $Z^{new}_{!e} = \emptyset$: Blocking detection. In this case, there is no assembly controller.
   - If $Z_1 \neq \emptyset$ then $Z^{new}_{!e} = Z_1$

4. Once the new temporal constraints are computed, the assembly controller arranges events according to the composite system architecture[2](interleaving architecture, hierarchical architecture, serial, ...). For example, in the interleaving architecture, events are performed interchangeably based on their time limits and their priorities.

Example 3 Let us consider the system of Figure 1. Recall that i_1, i_2 and i_3 are synchronization events (shared events). To ensure a correct interaction between components, the assembly controller given in Figure 2 restricts the behavior of C_1 and C_2. For that, it uses two clocks: α, controls the occurrence of the internal action i_1, and β, controls the occurrence of the internal actions i_2 and i_3. Now, to remove deadlock in location l_0 of C_2, action i_1 must be emitted within at most 1 unit of time. So, action $?e_1$ is only authorized in interval $[0, 1[$.

4 Generating Compositional test cases from the assembly controller

In the previous section, the overall behavior of the system is obtained by restricting free runs of components to those involving interactions between components. This restriction is achieved by assembly controller.Thus, compositional test cases are derived from the assembly controller model using symbolic analysis (zone-graph). This reduces the state space size (a practical size) and enables generation of sequences which cover all critical interaction scenarios.

4.1 Testing architecture

Figure 3 shows our compositional testing architecture where $C_1, C_2, \ldots C_n$ are real-time components and $T_1, T_2, \ldots, T_n$ are local testers. Each tester T_i controls and observes the local behavior of the component C_i. The assembly controller supervises interactions between components by authorizing, forbidding or forcing the execution of actions. It is the only component which has a global view of the whole system. The assembly tester checks the correctness of the assembly controller.

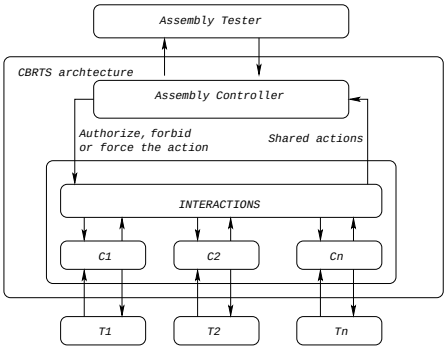

Figure 3. compositional testing architecture

In our architecture, components are assumed to be tested for conformance.

4.2 Deriving abstract tests from the assembly controller

Using symbolic analysis for generating tests is not new [4, 6, 10, 11, 13, 16]. However, using these approaches for CBRTS will leads to a huge number of test cases, and the non-observability of internal communications.

In our approach, we use an abstract model called the zone-graph to select abstract tests. Compositional tests are generated *directly* from the abstract model of the assembly controller or *indirectly* from the restricted behavior of each supervised component. Steps for generating abstract tests in the direct method can be summarized as follows:

1. Computation of assembly controller (according to section 3).
2. Computation of the zone-graph associated to the assembly controller. The zone-graph is constructed using *bounded-time reachability analysis* [18]. The computation of temporal successors is bounded by a delay $\Delta \in \mathbb{R}^+$. The resulting automaton is an *untimed graph* in which delay Δ is considered as an output action and each node is a symbolic state of the systems. Note that, extrapolation abstractions are used to ensure that the constructed graph remains finite.
3. Selection of abstract tests from the zone-graph according to a coverage criterion. For that, we adapt the untimed initial tour coverage tree of [8] for zone-graph. A path is an *initial tour* if it starts and ends in the initial location.An *Initial tour coverage tree* is a tree containing all minimal initial tours such that every edge is covered at least once and no tour is contained as a prefix or suffix of another tour. The initial tour coverage tree is constructed in two steps:
   - In step 1, from a given node, we compute all cycle free paths that lead to the initial node. The resulting tree is called *Homing Tree*
   - In step 2, we construct the initial tour coverage tree.

Compositional test can be also generated indirectly by combining test cases derived from the *restricted* behavior of each individual component. Test cases are combined according to the way the assembly controller arranges the shared events. Steps for generating abstract tests in this case can be summarized as follows:

1. Computation of the restricted behavior of each component (i.e. the new temporal constraints of shared events (see section 3))
2. Computation of the untimed graph for each (restricted) component.
3. Computation of the initial tour graph for each untimed graph.
4. Combining initial tours according to the following arrangement policies :
   - The order of events in each initial tour is preserved
   - Initial tours are synchronized only in Δ and in shared actions (This ensures that the time elapses at the same rate in all initial tours).

Figure 4 illustrates an example of combination of two initial tours T_1 and T_2 where (a) is a shared event. T_3 can not be combined with T_1 and T_2. In fact, the behavior described by T_3 is not allowed by the assembly controller and thus, the restricted untimed graph do not contain such behavior.

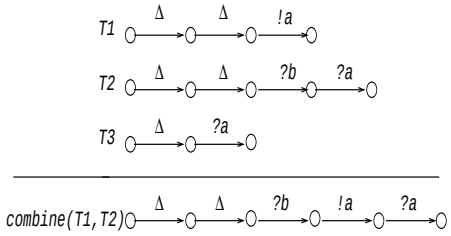

Figure 4. combining initial tours

4.3 Test cases concretization

4.3.1 Earliest and latest runs.

Initial tour tree constructed above defines a finite number of abstract tests. However, each abstract test contains a huge number of runs and thus of test cases. To reduce the number of test cases, while preserving a good coverage of symbolic states, we consider, for each abstract test, test cases derived from the *earliest* (fastest) and the *latest* runs respectively. The existence of the latest run assumes that time is bounded; this hypothesis is realistic since that testing is a finite experience. The general algorithm for computing the earliest run can be found in [14].

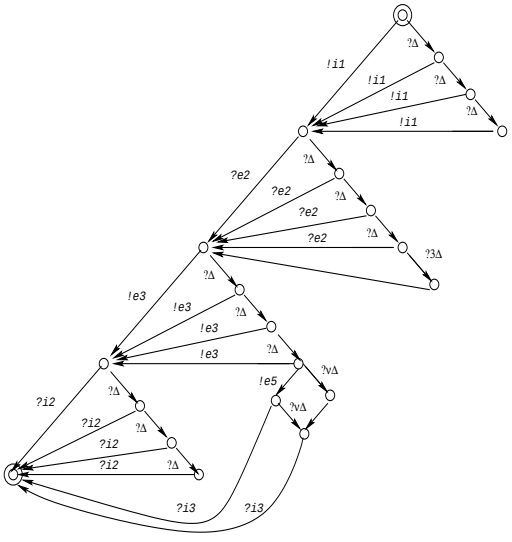

Figure 5. Untimed graph of C_2

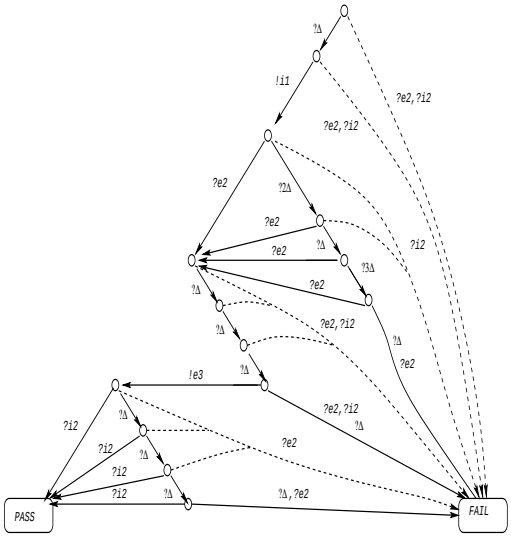

Figure 6. Test case generated for the untimed graph of C_2

4.3.2 Generating test cases

In our approach, a test case is dynamically generated, from an abstract test of the initial tour tree, to meet either the earliest or the latest run. The principal of construction is as follows: from a leave node l_f in the initial tour tree, we compute a path that reaches the root node. Then we extend this path into a tree such as for each node l in the path, and for each output action $\alpha \in \Sigma_! \cup \Delta$, we add output transitions. For example if there is an edge $l \xrightarrow{\alpha} l'$, where $\alpha \in \Sigma \cup \Delta$, then outgoing transitions $l \xrightarrow{\beta} l''$, $\beta \in \Sigma_! \cup \Delta$ must be added. The leaves of the tree are labeled PASS except an empty leave which is labeled FAIL. Figure 5 shows the untimed graph of component C_2, and Figure 6 shows an example of test case generated from the untimed graph of C_2.

5 Conclusion

In this paper, we proposed an approach to model and test component-based real-time systems. In order to avoid the construction of the whole system, we clearly separate the individual behavior of components from the way they interact. Interaction between components are *explicitly* modeled by the assembly controller.

In its first stage, our method check the composability of components. This can be achieved by synthesizing a non-blocking and optimal assembly controller. Then, compositional test cases are derived from the initial tours coverage tree of the assembly controller using a symbolic approach.

We are currently implementing a prototype tool that generate compositional tests with respect to the initial tour coverage criterion discussed in section 4. The prototype is written in C++ and contains two main modules : constraints solving module and graphs analysis module. We are also working on reducing the size of generated tests using partial order reduction techniques.
In this work, we have presented an approach that take into account the timing aspects while testing component-based systems. Other works propose methods that deal with the data aspects. A challenging problem is how to combine these two approaches to test component-based system with data and time.

References

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

[2] A. Basu, M. Bosga, J. Sifakis, *Modeling Heterogeneous Real-Time Components in BIP*. In 4 th IEEE International Conference on Software Engineering and Formal Methods (SEFM06), September 2006, pp 3-12.

[3] B. Brandin, W. M. Wonham *Supervisory Control of Timed Discrete Event Systems*. IEEE Trans. Automat. Control 39 (2), 1994, pp 329-341.

[4] R. Cardell-Oliver, *Conformance Tests for Real Time Systems with Timed Automata Specification*. Formal Aspect of Computing Journal, 350-371. 2000.

[5] David L. Dill. *Timing assumptions and verification of finite-state concurrent systems*. In Proceedings, Automatic Verification Methods for Finite State Systems, volume 407 of Lecture Notes in Computer Science, pages 197Ű212. Springer-Verlag, 1989.

[6] A. En-Nouaary, G. Liu,*Timed Test Cases Generation Based on MSC-2000 Test Purposes*, in Workshop on Integrated-reliability with Telecommunications and UML Languages (WITUL'04), France, November 2004.

[7] . H. Fauchal, A. Roullet, A. Tarhini *Robustness of composed timed systems.* In 31 annual conference on concurrent trends in theory and practice of informatics, LNCS, Springer 2005.

[8] R. Gotzhein, F. Khendek, *Compositional Testing of Communication Systems.* IFIP TestCom06, 227-244, 2006.

[9] R. D. Nicolas, M. Hennessy, *Testing equivalences for processes*. Theoretical computer science , 34: pp 83 - 133. 1984.

[10] M. Krichen, and S. Tripakis, *Black-Box Conformance Testing for Real-Time Systems*. In SPIN 2004. Spring-Verlag Heidelberg, 109-126.2004.

[11] K. Larsen, M. Mikucionis, and B. Nielsen, *On line Testing of Real-Time Systems*. Formal Approaches To Testing of Software, Link2, Austria. September 2004.

[12] O. Maler, A. Puneli, J. Sifakis, *On the synthesis of discrete controllers for timed systems.* Proc. STACS'95. LNCS 900, 1995, pp 229-242.

[13] B. Nielsen and A. Skou, *Automated Test Generation from Timed Automata*. In TACAS'01. LNCS 2031, Springer, 2001.

[14] P. Niebert, S. Tripakis and S. Yovine. Minimum-time reachability for timed automata. *In Mediterranean Conference on Control and Automation*, 2000.

[15] J. G. Ramadge, W. M. Wanham, *The Control of discrete event systems*. Proc. IEEE 77 (1), 1999, pp 81-97.

[16] J. Springintveld, F. Vaandrager and P. D'Argenio. *Testing Timed Automata*. Theorical Computer Science. 254, 2001.

[17] T. Higashino, A. Nakata, K. Taniguchi, and A. Cavalli, *Generating Test Cases for a Timed I/O Automaton Model.* IFIP (IWTCS'99) Budapest, 1999.

[18] S. Tripakis, *Fault diagnostic for timed automata*. In FTRTFT'02. volume 2469 of LNCS. Spring,2002.