Validating time-constrained systems using UML Statecharts Patterns and Timed Automata Observers

Ahmed MEKKI¹, Mohamed GHAZEL¹, and Armand TOGUYENI²

¹ESTAS, INRETS, 20 rue Elisée Reclus BP 317, Villeneuve d'Ascq, 59666, France

²LAGIS, Ecole Centrale de Lille BP 48, Villeneuve d'Ascq, 59651, France

ahmed.mekki@inrets.fr, mohamed.ghazel@inrets.fr, armand.toguyeni@ec-lille.fr

Abstract

The work presented in this paper is part of a project that aims to develop of a new approach to time-constrained system verification based on UML. In fact, being relatively easy to learn and use, UML is very popular, unlike formal methods. Nevertheless, formal models provide developers with several advantages: they can be used for activities, such as properties verification, which are crucial for critical system development. Such activities are less effective when carried out on UML models. Also, because UML semantics is informal, the automatic verification of UML models is directly unfeasible. In this paper, we propose a new verification approach that takes advantage of UML Statecharts model to implement a new technique based on both Timed Automata Observers and model checking. The contributions of the paper are the definition of new time constraints (properties) taxonomy and the definition of a Statecharts patterns' base. These patterns are then translated to Timed Automata Observers. The obtained observers are synchronized with the system specification, thus reducing the verification task to a reachability analysis.

Keywords: UML Statecharts, Timed Automata, Patterns observers, model-checking, time-constrained system.

1. INTRODUCTION

Specification and verification of time-constrained systems are the most important research topics in critical systems engineering. They also have practical implications on safety and correctness of systems. The most popular approaches for specifying real-time systems are based on Timed Automata (TA). TA are well adapted for the modelling of real-time components with explicit clock variables. Time properties are captured by explicitly setting/resetting clock variables. A number of automatic verification tools for TA have been developed and have proven to be efficient (e.g., Uppaal [20] and Kronos [29]). Nevertheless, specifying and verifying time constraints is becoming a more and more difficult task due to the widespread applications and increasing complexity of checked systems. Despite, the different advantages proposed by TA, such as parallel composition, its users often need to manually express the time properties into a set of clock variables with complex calculated clock constraints. The process is tedious and error-prone. Another issue in this field is the abundance of formal methods and supporting tools: although they have been proven useful to support the development of real-time systems, they are not much used in industry. This is due to the variety of formal methods and to the difficulty in using the tools, which often require sophisticated logical or mathematical skills.

Currently UML is increasingly employed in real-time system verification. UML was first used as a standard for documenting system specification and design. The increasing reputation of this notation has allowed system developers to use UML to model application domains that were originally out of the language scope. UML is now used in application systems that call for a better formalization, such as real-time, safety-critical and embedded systems. In such systems, ambiguities and inconsistencies could have catastrophic consequences. That is why it is highly recommended that the modeling notation used should be formal.

The aim of this work is to introduce a new temporal requirements' verification method. In the framework of this approach we first define a Patterns' Basis for expressing time constraints. Indeed, based on a new time constraints' classification, we developed a set of time observation patterns expressed in UML Statecharts: this is expected to be a relatively inexpensive activity

since procedure is made once for all. UML has been chosen since it is relatively intuitive, offers a graphical description, and is well supported by tools. This patterns' set facilitates high-level system design. The developed patterns cover a large class of common time constraints. This basis is implemented once for all. And, it guarantees the reusability of our approach by providing a generic patterns' basis.

Once our observation patterns' basis is implemented, we introduce a verification process based on this basis. The observations patterns are instantiated to express time-constraints that can be extracted from the system requirements. This instantiation step generates a set of Statecharts Observers. The Statecharts observers were translated into more formal notation, the TA, which provides support for the properties' verification. This translation is made according to some transformation rules we developed. In this way, developers exploit the benefits of formal notations without being obliged to go through the complex and expensive formal modeling phase. This transformation generates a set of TA Observers. The obtained TA observers are synchronized with the system's model built on the same formalism. Hence, the verification task is performed with a reachability analysis (Reachability analysis is defined as follows: for verification of a property using reachability analysis, we compute at first the set of reachable configurations. Secondly, we intersect the reachable set with the set of invalid configurations. If the intersection is empty, then the property is satisfied).

This work identifies the strengths and weaknesses of TA and UML Statecharts and develops a new technique for temporal properties' verification. Comparing it to existing verification approaches, our method offers the following advantages:

- More than one property can be checked with the same resulting model,
- No temporal logic knowledge is needed to express requirements,
- Timed patterns facilitate high-level specification and promote reusability and knowledge capitalization,
- The verification task is reduced to a reachability analysis (this allows to overcome some limitations met with some existing tools such as Uppaal [20]).

The paper is organized as follows. Section 2 briefly introduces TA and UML Statecharts. Section 3 presents the main techniques used for time constraints' verification. Section 4 describes our contribution by presenting the new approach with the introduction of a set of timed observation patterns with formal semantics. We also show how these patterns can be instantiated to express time constraints. We show, also, how TA users may benefit from our patterns, by defining a formal translation from UML Statecharts to TA. Section 5 illustrates our approach using a level crossing use case, while section 6 draws some conclusions and future work.

2. BACKGROUND

2.1. Timed Automata

A timed automaton [2] is a graph containing a finite set of nodes (or locations) and a finite set of labeled edges extended with clocks. In this model, time information is represented by a finite set of clocks, which can be reset by state transitions. All the clocks are incremented synchronously when time elapses. The values of these clocks are then used in the transition guards. A clock guard is a boolean constraint on the values of the clocks. A state transition can only occur if the guard of the transition is satisfied.

Formal definition (Timed Automaton) [26]. A Timed Automaton A is a 7-tuple $(S; \text{init}; F; \Sigma; C; \text{Inv}; T)$, where

- S is a finite set of states,
- $\text{Init} \in S$ is the initial state,
- $F \subset S$ is a set of final states,
- Σ is a set of actions/events,
- C is a finite set of clocks,
- $\text{Inv}(s)$ is a mapping from a state s into $\Phi(C)$ such that clock valuations in s must satisfy $\text{Inv}(s)$,
- $T: S \times \Sigma \times 2^C \times \Phi(C) \times S$ is the transition relation. A transition $(s; a; \lambda; \delta; s')$ represents a transition from state s to state s' on event a . The set λ gives the clocks to be reset with this transition and δ is a clock constraint over C that specifies when the transition is enabled. The set of clock constraints $\Phi(C)$ is defined by the following grammar: $\varphi ::= \text{true} \mid x \leq c \mid c \leq x \mid x < c \mid c < x \mid \neg \varphi \mid \varphi_1 \wedge \varphi_2$ where x is a clock and c is a real number.

The semantics of a Timed Automaton is defined as a transition system. A run of the timed automaton starts with the initial state. The control may remain at a state s as long as $\text{Inv}(s)$ is not violated. A transition $(s; a; \lambda; \delta; s')$ is enabled if, and only if, the control is at state s , guard

condition δ is satisfied (by the valuation of the clocks), and event “a” is enabled. The transition is unguarded if δ is true. After firing the transition, the control moves to state s' and the clocks in λ reset.

Example. Figure 1 shows a sample Timed Automaton specifying a train behavior on a railway.

FIGURE 1: Train Timed Automaton

The timed automaton of figure 1 models the position of a train relatively to the level crossing. The train alternates between locations Far, Near, Inside. When transition labeled by ‘approach’ is fired, clock x is reset to 0. When clock x becomes greater than 300, transition “Near to Inside” can be taken. The train must leave locations Near, Inside before clock x becomes greater than 500.

The major limitation of this formalism is the state-space explosion. To cope with this problem, techniques for reducing the state-space and partial order techniques have been developed [4, 5, 24].

2.2. UML Statecharts

Statecharts are employed to express, graphically, finite state machines. A state or a location or a node is a situation where the object satisfies some condition, can perform some activity, or waits for some event. The object’s reactions include changing states, executing internal actions and sending events to other objects.

The notation and semantics of UML Statecharts [27] were adapted from Harel’s Statecharts diagrams [14]. UML Statecharts are a highly expressive visual language that is used to model the UML objects’ behavior. A Statecharts diagram specifies the states that an object can exist in and describes the result of the object’s reactions to events from the environment.

Example. Figure 2 shows the statecharts model of the train presented in Figure 1.

FIGURE 2: Train Statecharts model

According to [14], a Statecharts diagram consists of nodes, pseudostates and transitions between them. Nodes can be simple, composite (OR node and AND node), or final. Simple nodes and final nodes are the basic constructs, with final nodes indicating the completion of their containing nodes. Composite nodes are constructs containing a group of nodes. A node may have entry and exit actions, and internal activity. An entry action (resp. exit action) is performed when a node is entered (resp. exited). Internal activity begins after the entry actions are performed.

UML Statecharts allow a transition to return to a previous node within a composite node, without requiring that one specifies which node the system was in previously. This is represented by a transition arrow that ends in an encircled ‘H’ for ‘history’^(H). There are two kinds of history

returns in UML statecharts: shallow and deep. Shallow transitions (indicated by $\textcircled{H}$) return to the most recently exited node at the level where the 'H' is shown. If that does not lead to a simple node, then it is an error. Deep history (indicated by $\textcircled{H^*}$) means that the system will return to the most recent simple node within the compound node that it most recently exited.

3. TIME CONSTRAINTS VERIFICATION TECHNIQUES AND EXISTING APPROACHES

3.1. Model-checking technique

Model-checking is an automatic technique for verifying finite-state systems. In time-constraint systems, temporal requirements are generally expressed in a temporal logic, and the system is modeled as a state-transition graph (TA, Petri Net...). An efficient search procedure is used to determine automatically if the requirements are satisfied by the specification. This technique has several important advantages over theorem proving for verification methods. The most important is that the procedure is highly automatic. Typically, the user provides a high-level representation of the system and the requirements to be checked. The model checker will either terminate with answer true, indicating that the requirement is satisfied, or give a counter-example execution that shows how the formula is violated. The most used model-checkers for real-time systems are Uppaal [20] and Kronos [29]. Both tools use TA for modeling the system and TCTL logic for the expression of properties. Uppaal is very good at modeling by providing a simulator and a well-developed GUI. However, unfortunately, the tool provides only a binary synchronization between processes and can only verify the reachability properties. In the other hand, Kronos allows the verification of a large class of properties (reachability, safety and liveness), and offers multiple analysis techniques (forward analysis, backward analysis and on-the-fly). The tool also provides an algorithm of symbolic model-checking, which partially solves the combinatory explosion problem. Nevertheless, Kronos does not provide GUI nor simulation module.

3.2. Observer technique

Another technique is related to the so-called "observer" approach [13, 28]. Basically, observers are modules synchronized with the system specification. The observers check whether some particular state characterizing the violation of the property is reachable; in such a case, the product system reaches the error state (state that is defined in the observer). Proving that the property observed by the observer is valid consists in showing that the error state is not reachable. The observer technique presents two main advantages. The first is that it can easily be implemented, since it results in reachability analysis. The second advantage is that, unlike model-checking that allows checking of one property on each execution, the observer technique allows checking of more than one property simultaneously. However, the main limitation is that this technique is less powerful than model checking (some properties cannot be expressed using observers). Finally, one may point out that the property is expressed with the same formalism as the one used for specifying the system; this may be seen as an advantage (only one formalism needs to be known), but also as drawback the formalism used for the system specification may be incongruous for requirements expressing.

3.3. Related Work

The idea of applying formal analysis techniques on UML statecharts, such as model checking, is not new and has been a very active field of research in the last decade. This is witnessed by the numerous works on this subject [21, 19, 12, 21, 22, 25, 10, 23].

The most part of these works are based on an existing model checker (SPIN [15] is used in [21, 19, 12, 22, 25], and Uppaal [20] in [10]) and on the translation of UML specifications into the input language of the used model checker.

Most of the works done on UML models verification has been developed for the model checker SPIN[15]. Three of these most important works will be presented below.

Latella and al [21] presented a mapping from UML Statecharts to PROMELA (PROcess MEta LAnguage) which is the input language of SPIN, however Latella and al [21] only allow a model to contain a single state machine;

The Hugo project [19] uses SPIN as a back-end tool to verify UML models. The initial PROMELA translation was only feasible for very small models [16]. The mapping step from UML to SPIN is carried out automatically;

In [12], SPIN is used to generate test cases from abstract state machines.

Some other works done on UML models verification has been developed for the model checker Uppaal. A translation of UML collaborations with time-constraints into UPPAAL timed automata has been presented by Firley et al. [10].

Another work called OMEGA project [23] has been conducted in order to verify the UML models. OMEGA has created a tool set focusing on real-time properties. Its approach is based on translating UML to the IF intermediate language that has several model checking back ends.

Keeping in mind that the most important step on the UML models verification is the properties specification phase, some authors go for the property language of the model checker itself (e.g. [21, 22]). Others use UML collaboration diagrams (e.g. [19, 25]) which are too weak to express all the relevant properties. We propose to use the UML statecharts to express properties in terms of observers.

If a comparison is performed between the work presented in this paper and the above-mentioned work, we could conclude that the main advantages of our method are:

- The UML statecharts model is used for the properties specification; it is more expressive than the UML sequence diagrams or UML collaboration diagram used in other works;

- No temporal logic is needed for specifying system's temporal requirements, since these requirements are introduced as model that will be synchronized to the global model;

- Pattern basis that will be defined below simplifies the temporal requirement specification task;

- Overcome the limitation of existing model checker tools by reducing the verification to a reachability analysis;

- Well adapted model checker tools (Uppaal[20] and Kronos[29]) are using the TA formalism, the target formalism of our transformation.

4. DEVELOPED APPROACH

4.1. Global Idea

TA have proven to be useful for time-constrained system specification and verification. In spite of the number of automated analyzers developed for TA, these tools suffer from two main limitations; the first is that users must be familiar with its formal notation. The second is the lack of patterns for high-level system design. On the other hand, specification semi-formal languages, such as UML Statecharts, are well suitable for expressing systems' requirements. However, the automatic verification of these models is unfeasible directly. In order to take advantages of the analysis facilities offered by TA formalism and the expression flexibility of Statecharts, we introduced a temporal requirements verification method. In our method, first a Statecharts observation pattern is developed. Second, the verification process is performed based on the observation patterns basis.

4.1.1. Observation Patterns Basis

A pattern is a commonly reusable model in software systems that guarantees a set of functionalities. The identification of a pattern is based on the context in which it is used. The goal behind developing patterns is to offer a support for system design and development. Based on patterns, there is less focus on the technology. Software patterns first became popular with the object-oriented Systems' Design [11]. Using patterns helps in keeping design standardized and useful, and minimizes the reinventing in the system design, since patterns facilitate knowledge capitalization and reusability [7].

In this work, we define a set of UML Statecharts patterns for presenting temporal requirements of systems. The set of patterns is implemented once for all. The role of each UML Statecharts pattern is to express a given temporal requirement. The patterns' basis is introduced undependably of systems' specification and is used to model all the temporal requirement types that we can find. This pattern basis guarantees the reusability and the genericity of the mechanisms developed within our approach. Figure 3 describes graphically the patterns' general process.

Figure 3: Observation Patterns Basis implementation process

4.1.2. Verification Process

Once the observation patterns' basis is introduced, we discuss, next, how this basis is used in the verification process. Based on a given system's specification, the temporal requirements that the system should satisfy are identified and extracted. For each extracted requirement, the adequate pattern is instantiated from the patterns' basis. This instantiation generates a set of Statecharts observers. Each Statecharts observer represents an extracted requirement. The Statecharts observers are then translated into TA observers. The translation from the UML Statecharts to TA is performed using Model Transformation techniques (section 4.5). The generated TA are then synchronized with the system's specification model to generate a global model. Thus, the verification task is reduced to a simple error-state reachability analysis on the obtained global model. Figure 4 depicts graphically our approach.

FIGURE 4: The new approach architecture

4.2. Main Time-Constraints' Classification

Having described our motivations, we discuss now our approach. We start by describing a new basic classification for temporal requirements (Figure 5) to be used afterwards in our approach. Indeed, in spite of the large amount of work that has tried to propose time constraints classification [1, 3, 8, 18], we do not find a complete and detailed example. We present briefly a new classification. This new classification is based on events. An event represents a specific change which potentially occurs inside the system.

FIGURE 5: Time Constraints Structure

The timing requirements typology is split into two major classes, namely:

Requirements where time is expressed in a qualitative way. This class of requirements exclusively considers order between events.

Requirements where time is expressed in a quantitative way. This class of requirements considers the offered time for events. This kind of requirement must have a deadline after/before which a property is not satisfied. This implies that the system must be temporally bounded.

The main identified requirement (Table 1) types are deterministic delay, bounded delay (latency), maximumDelay, MinimumDelay, simultaneity, and sequence. (The relation which denotes that a system S satisfies a requirement R is written $S \models R$.)

MinimumDelay	R ensures that an event must occur after a minimum time Tmin. $S \models R$ is true if this event occurs after Tmin.
MaximumDelay	R ensures that an event must occur before a deadline Tmax. $S \models R$ is true if this event occurs before Tmax.
Latency	R ensures that an event must occur between a temporal interval]Tmin; Tmax[. $S \models R$ is true if this event occurs between temporal interval]Tmin; Tmax[.
Delay	R ensures that an event must occur at one punctual date T. $S \models R$ is true if this event occurs at the T date.
Simultaneity	R ensures that two events occur at the same date. $S \models R$ is true if $Event1.occurrence = Event2.occurrence$.
Sequence	R ensures that two events occur at two different date. $S \models R$ is true if $Event1.occurrence \neq Event2.occurrence$.

Tableau 1 : Temporal Requirement's Taxonomy Descriptions

4.3. Observation Patterns

Patterns can be seen as artifacts that describe subsets of raw data by isolating only the interesting properties. Thus, we may informally say that a pattern is compact and rich in data representation. A pattern is interesting to the end-users because it describes a recurrent behavior.

In order to ensure that the requirement verification process does not influence the system specification, observers are defined as external blocks that will be composed with the system specification model. The observers are instantiated from patterns (in this work, our patterns are specified in UML Statecharts models). The goal of the verification task is to check the requirements' violation. Using the new approach, this task will be completed through the instantiation of the Statecharts observers to represent the identified requirement. Then, each Statecharts observer will be translated into TA observer. The TA observer will be synchronized with the system model (the system model is expressed in TA formalism). The TA observer has a set of so-called error nodes (KO). If one of these nodes is reachable in the composite model, then the corresponding requirement is violated. For example, to check a system Σ with an observer O , the process is as follows: the observer O with Σ is composed using a synchronous composition ($\Sigma \parallel O$). An error node reachability analysis is performed on the product system ($\Sigma \parallel O \rightarrow \text{error}$). If any of these nodes (KO) is reachable, then the property is not satisfied. Otherwise, if none of these error nodes is reachable, the property is satisfied.

In the following, two timed patterns are introduced. First, the textual definition is presented, followed by a Statecharts graphic representation.

Deterministic Delay Pattern [Figure 6]. The delay pattern is used to check that a given event execution is delayed by exactly d time units relatively to a given reference.

FIGURE 6: Delay Pattern

The example process of Deterministic Delay Pattern (Figure 6) is provided by a simultaneously passage from both node A and node B. Node A is reached after exactly d time units after a reference node "state 0", and node B is reached after the event "b" detection. The error node "KO" is reached if we are not in node A and node B at the same time, i.e the event "b" does not occur exactly d units time after the event "c".

Latency Pattern [Figure 7]. The latency pattern is used to check that a given event execution is delayed by non-deterministic delay ℓ time units. This ℓ should be in the latency interval $]Tmin, Tmax[$.

FIGURE 7: Latency Pattern

The normal process of this example (Figure 7) is ensured by a transition from a reference node "state 0" to the node "OK". This transition is guaranteed if the event "a" detection is obtained within a temporal interval $]Tmin; Tmax[$. The error node "KO" is reached if the "a" is detected before $Tmin$ or if "a" does not occur until $Tmax$.

4.4. Translating Statecharts to Timed Automata

UML Statecharts diagrams are a well-known state-based formalism with high expressive power. One of the key parts of our approach is the transformation of UML Statecharts to TA. The transformation is inspired from [6, 9].

The UML Statecharts are transformed into TA in two steps: First, each Statecharts is flattened, meaning the hierarchical structure is removed because TA used in the existent approaches lack hierarchy concepts (Existing approaches and techniques deal only with sequential automata). Next, the family of flat Statecharts is transformed to a family of TA. In the following, the transformation is described in detail.

Each simple Statecharts state is converted into a location of the target automaton;

Final states are replaced by normal states;

Composite Nodes:

Sequential (OR) node: Sequential node “s” is translated into a single TA. Each node or sub-node in “s” is translated into a location in the TA. All the locations in the generated TA should have an outgoing transition that represents the exit event of the node “s”. Figure 8 shows an example of OR node (1) and the generated TA (2). All the locations of the generated TA contain an outgoing transition “Out”.

Figure 8: OR node

AND node: the translation requires a TA for each orthogonal region in the source node “s”. All the obtained TA must be entered at the same time. This is guaranteed by synchronization via an entering event. This entering event is the entering event to the node “s”. Also, the TA should be synchronized by an exit event. The exit event is useful to place all the TA in a “wait” location if the node “s” is abandoned. This exit event is the outgoing event from the node “s”. The translation of each orthogonal region is carried out according to the other rules with two exceptions; first an initial “wait” location should be added for each generated TA. “Wait” is left when the entering event is detected. Second for each location in the generated TA, an outgoing transition should be added. These transitions are fired on the exit detection. The target location of these transitions is the “wait” location. Figure 9 show an example of translation AND node. Part (1) represents the source model which is in this case an AND node. The (2) represent the generated automata. Unlike the transformation presented in [6], where they use two additional events to synchronize the generated automata, our rule does not add additional events to the model.

Figure 9: AND node

The state “s” invariants are determined based on outgoing transition conditions: invariant must be less than, or equal to, the maximum of clock of all the outgoing transitions from “s”. States without timed outgoing transitions should have invariance condition equal to TRUE;

The non-timed transition guards must be TRUE;

Timed transition guard values in the interval $[min, max]$ must have the form: $clock \geq min$ and $clock \leq max$ (or $clock = max$ if $max = min$);

Events generated by a transition and sent to other objects should be declared as synchronization events;

Entry (resp. exit) actions which may be associated with some states, are simply assigned to all entering (resp. outgoing) transition of the equivalent TA location.

An automatic translation is then performed using translators whose algorithms carry out the rules of translation. The rules of translation are defined according to the translation principles defined above. The different rules are expressed based on the MDE approach [Figure 10];

Source and target models are expressed conforming to their meta-model,

Rules are introduced at meta-model level,

Transformation should implement the defined rules. It takes the Statecharts model as the source model and generates a TA model.

FIGURE 10: Statecharts to Timed Automata Architecture

Once the TA are generated, they will be synchronized with the system specification. Error states reachability analysis is then implemented to check the properties violation.

The transformation presented above suffers from some lacks and several issues still remain to be handled.

The rules presented above describe a translation schema from UML Statecharts to TA only at the syntactical level. The correctness, i.e. whether relevant system properties can be preserved by the translation, of translation will be showed in future work by studying the translation at the semantical level.

So far, guards and actions expressions used have a simple arithmetic form. In Statecharts guards are more flexible;

History connectors and do activities are omitted from the transformation;

Then, guards of transition issued from the same source node cannot be satisfied simultaneously. This is not true in general.

5. CASE STUDY: LEVEL CROSSING SYSTEM

The railroad crossing system has been used to illustrate our approach. The specification is based on [17]. Figure 11 shows an automatic controller that opens and closes a gate at a railroad crossing. The system is composed of three components, Train, Gate, and Controller, which execute in parallel and synchronize through the events: Approach, Exit, GoDown, and GoUp. When a train approaches the crossing section, Train sends an *Approach* signal to Controller. The train enters the crossing section at least 300 seconds later. When a train leaves the crossing section, Train sends an *Exit* signal to Controller in order to synchronize with it. The *exit* signal is sent within 500 seconds after the *Approach* signal. Controller sends a signal *GoDown* to Gate exactly 100 seconds after the *Approach* signal and sends a *GoUp* signal within 100 seconds after *Exit*. Gate responds to *GoDown* by moving down within 100 seconds and responds to *GoUp* by moving up between 100 and 200 seconds.

FIGURE 11: Railway System

In this example, we identify the presence of 4 latency requirements and 1 Delay requirement. In the following, two requirement examples are introduced: “latency” and “delay”. First, a textual description of the requirement is presented, followed by an intuitive graphic representation. To express our requirement, we instantiated the equivalent Observation Patterns defined in the Observation Patterns Basis. This instantiation generates Statecharts observers. The statecharts observers are then translated into AT observers. The translation is performed by a model transformation process. The rules used in this process were defined in section 4. From the description we pick out a delay of 100 seconds between the “Approach” detection and “GoDown” transmission. Figure 12 shows the Statecharts observer (1) of the identified delay. The (2) is the TA observer generated from the (1).

FIGURE 12: Delay Observers

Also a MaximumDelay of 100 seconds between the “Exit” detection and “GoUp” transmission can be picked from the railroad crossing system. Figure 13 shows (1) the Statecharts observer of the identified MaximumDelay. (2) is the TA observer generated from the (1).

FIGURE 13: MaximumDelay Observers

Even if in our example, Statecharts observers and associated TA observers can seem very similar; this is not the case in general. Indeed, in Statecharts transitions' guards are much more permissive than TA transitions' guards. Moreover, TA do not offer hierarchy description and then do not allow to translate directly composite statecharts nodes.

6. CONCLUSION

In this paper we have presented a Statecharts-based method applicable to formal specification and validation of time-constrained systems. This approach is based on two main processes; first process defines an observation patterns basis. The second is a verification process based on the defined observation patterns basis.

The basis is used to specify all temporal requirements that we can meet in a systems' specifications. Using patterns allows genericity and reusability.

The verification process defined in this new method consists of four steps. First, temporal requirements are extracted and identified from the system's specification. These requirements are, then, expressed by instantiating the adequate patterns from the observation patterns basis. The instantiation process generates a set of Statecharts observers. Second, the Statecharts observers are translated into TA observers. The transformation is performed by using the transformation rules that we have defined. Third, the generated TA observers are synchronized with the system's specification TA model. Finally, the verification task is performed on the product model resulting from the synchronization between TA observers and TA specification model. The verification task is, thus, reduced to a simple error-state reachability analysis.

The main contributions of our work are: first, a new timed properties' classification is introduced. Second, a set of observation patterns which covers all the timed properties identified in the classification, is implemented. Our patterns allow developers to express system's time properties in a precise graphical manner. Therefore, this greatly simplifies the temporal requirements specification task. Third, a method for translating Statecharts models into TA is introduced. Finally, the verification process proposed used is based on the model-checking technique and the observer's technique. The verification is reduced to a reachability analysis enabling thus, to overcome some limitations met with some existing tools such as Uppaal [20].

In future, we aim to make transparent the verification formalism as much as possible for the user. Therefore, the proposed transformation algorithm will be tested to generate automatically the TA observers. We will, also, try to generate a graph from the error traces generated from model-checking tools. Then, we aim to make the approach application automatic and integrate it with some existing verification tools.

REFERENCES

- [1] Allen, J.F., (1984) Towards a general theory of action and time. *Artificial Intelligence*, 1984(23): pp.123-154.
- [2] Alur, R. and Dill, D.L. (1994) A Theory of Timed Automata. *Theoretical Computer Science*, 126, 183-235.
- [3] Alur, R., & Henzinger, T., (1993b) Real-time logics, Complexity and expressiveness. *Information and Computation*, (104) : pp.35-77.
- [4] Artho, C., Leungwattanakit, W., Hagiya, M., and Tanabe, Y. (2008) Architecture-aware Partial Order Reduction to Accelerate Model Checking of Networked Programs. *Proceedings of the 2008 Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, 807-813.
- [5] Bonakdarpour, B., and Kulkarni, S.S. (2007) Exploiting Symbolic Techniques in Automated Synthesis of Distributed Programs with Large State Space. *27th International Conference on Distributed Computing Systems*. 25-27 June 2007, 3 – 3.
- [6] Bouamari, A., and Mostefai, M., (2007) Vérification formelle des modèles UML temps-réel. *4th International Conference on computer Integrated Manufacturing CIP'2007*.
- [7] Coplien, J.O. *Software Patterns*, <http://www.hillside.net/patterns/definition.html>
- [8] Dasarathy, B., (1985) Timing constraints of real_time systems: construct for expressing them, methods of validating them. *IEEE transaction software engineering*, vol. 11, pp.80-86.
- [9] Diethers, K., Goltz, U., and Huhn, M., (2002) Model checking UML statecharts with time. *UML 2002: Workshop on Critical Systems Development with UML*.

- [10] Firley, T., Huhn, M., Diethers, K., Gehrke, T., and Goltz, U.(1999) Timed Sequence Diagrams and Tool-Based Analysis — A Case Study. In Proc. Second Int'l Conf. UML '99, 645-660.
- [11] Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995).- Design Patterns : Elements of Reusable Object-Oriented Software. Reading Mass, Addison Wesley.
- [12] Gargantini, A., Riccobene, E., Rinzivillo, S. (2003) Using SPIN to generate tests from ASM specifications. In: Abstract State Machines. Number 2589 in LNCS, Springer-Verlag.
- [13] Ghazel, M., Toguyéni, A., and Yim, P., (2009) State Observer for DES Under Partial Observation with Time Petri Nets, Journal of Discrete Event Dynamic Systems: Volume 19, Issue2 (2009), 137-165.
- [14] Harel, D., Pnueli, A., Schmidt, J. P., and Sherman, R., (1987) On the formal semantics of Statecharts. In Proceedings of the 2nd IEEE Symposium on Logic in Computer Science, pages 54–64. IEEE Computer Society Press.
- [15] Holzmann, G. J.(1997) The model-checker SPIN. IEEE Trans. on Software Engineering, Vol. 23, No. 5,279-295.
- [16] Jussila, T., Dubrovin, J., Porres, I., Latvala, T., and Junttila, T.(2006) Model Checking Dynamic and Hierarchical UML State Machines. Proc. 3rd International Workshop on Model Development, Validation and Verification, Seite(n) 94-110.
- [17] Kang, I., Lee, I., and Kim, Y.S, (2000) An efficient state space generation for analysis of real-time systems. Transactions of Software Engineering, 26, 453-477.
- [18] Konrad, S., & Cheng, B.H.C., (2005) Real-time Specification Patterns. In Proceedings of the 27th International Conference on Software Engineering (ICSE05), St Louis, MO, USA.
- [19] Knapp, A., Merz, S., and Rauh, C.(2002) Model checking timed UML state machines and collaborations. In W. Damm and E.-R. Olderog, editors, 7th Intl. Symp. Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT 2002), volume 2469 of Lecture Notes in Computer Science, pages 395–414, Oldenburg, Germany, September 2002. Springer-Verlag.
- [20] Larsen, K., Pettersson, P., and Yi W, (1997) UPPAAL in a nutshell. International Journal of Software Tools for Technology Transfer, 1(1-2), 134-152.
- [21] Latella, D., Majzik, I., and Massink, M. (1999) Automatic verification of a behavioral subset of UML statechart diagrams using the SPiN model-checker. Formal Aspects of Computing, (11), 637–664.
- [22] Lilius, J., and Paltor, I.P.(1999) Formalizing UML state machines for model checking. In Rumpe France, editor, Proceedings of UML'1999, volume 1723 of Lecture Notes in Computer Science. Springer-Verlag.
- [23] Ober, I., Graf, S., Ober, I. (2004) Validation of UML models via a mapping to communicating extended timed automata. In: 11th International SPIN Workshop on Model Checking of Software, 2004. Volume LNCS 2989.
- [24] Ribert, P., Vernadat, F., and Berthomieu, B. (2002) On Combining Persistent Sets Method with the Covering Steps Graph Method. 22th IFIP WG6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE' 2002), 344-359.
- [25] Schafer, Knapp, A., and Merz, S. (2001) Model checking UML state machines and collaborations. Electronic Notes in Theoretical Computer Science, 55(3).
- [26] Song Dong, J., Hao, P., Qin, S., Sun, J., and Yi, W. (2008) Timed Automata Patterns. IEEE Transactions on Software Engineering, vol. 34, no. 6, 844-859.
- [27] The Object Management Group. OMG Unified Modeling Language Specification, February 2009. Version 2.2, <http://www.omg.org>.
- [28] Yi Huang Luo, S., Svinin, Z.W., Odashima, M., and Hosoe, T. (2002) Extended state observer based technique for control of robot systems. Proceedings of the 4th World Congress on Intelligent Control and Automation. 4, 2807- 2811.
- [29] Yovine, S. (1997) KRONOS: a verification tool for real-time systems. International Journal of Software Tools for Technology Transfer, 1(1/2): 123-133.