An Approach for Validation, Verification, and Model-based Testing of UML-based Real-time Systems

Mehdi Nobakht and Dragos Truscan
Department of Information Technologies, Åbo Akademi University, Turku, Finland
{mehdi.nobakht, dragos.truscan}@abo.fi

***Abstract*—UML is gaining popularity in designing real-time systems. However, UML tools often lack support for verification. This paper describes an approach and a tool in which UML models used for designing real-time systems are translated into UPPAAL timed automata in order to take advantage of validation and verification support in the UPPAAL tool. This allows one to increase the quality of the UML models by complementing static validation via OCL with behavioral validation and verification using the UPPAAL model-checker. Having an implementation of the system under consideration, the obtained UPPAAL timed automata serve as input of the UPPAAL-TRON tool to perform online model-based conformance testing. The proposed approach also generates a skeleton of the test adapter required to interface the testing tool and the implementation under test. The approach and the tool are exemplified with a telecommunication case study.**

***Keywords*—UML; UPAAL; model verification; model-based conformance testing; real-time systems.**

I. Introduction

Unified Modeling Language (UML) [1] is a standardized general-purpose modeling language originally designed for the object-oriented paradigm. UML has also been suggested for designing embedded and real-time systems. It has been gaining popularity and is familiar to most designers and developers in this class of systems [2]. A key advantage of UML is the hierarchical mechanism giving a high degree of modularity and encapsulation to the model. It is particularly useful for modeling the behavior of complex systems. Moreover, an increasing number of UML tools provide code generation facilities which has increased its popularity further.

Once a real-time system is designed using UML, there is a need to ensure that the model conforms to the system specification. Model validation and verification methods aim at finding possible discrepancies between a system model and the corresponding specification at an early design stage. The Object Constraint Language (OCL) [3] is a formal language to supplement UML for detecting both syntactic inconsistencies and, to a limited extent, semantic ones in the models. While UML is particularly promising in designing embedded and real-time systems, it lacks support for verification of the timing and schedule related properties.

Testing is the pivotal part of real-time systems development process, being used to ensure that a product meets its requirements. This way, it helps to increase the quality of the product. *Model-Based Testing* (MBT) [4] is a testing technique which automatically generates tests from the behavioral specifications of the System Under Test (SUT). Depending on how tests are generated and executed, there are two flavors of MBT; in *offline testing*, the test cases are generated before the execution step, whereas through *online testing* both steps are integrated [5].

The work presented in this paper proposes an approach for validation, verification, and online model-based conformance testing of real-time systems which are designed using UML. In our approach, in order to compensate for the lack of formal and executable semantics of UML, the UML models including class and state machine diagrams are translated into the UPPAAL timed automata and later on validated and verified using the UPPAAL model-checker tool [6]. The translation is automated by a tool which beside creating the UPPAAL specifications, it propagates requirement information from the UML models to the UPPAAL timed automata and generates deadlock free and reachability queries for verification purposes. In addition, it generates a tester adapter stub required to interface UPPAAL-TRON – an online model-based testing tool [7] – and the Implementation Under Test (IUT).

Overview. The information presented in this paper will appear in the following order: Section II contains the works related to verification methods for the UML-based designs of real-time systems. Section III provides a background to the theory of timed automata, the semantics of timed automata as used by the UPPAAL toolbox, and the underlying principles of TRON. Section IV initially provides a UML solution for designing real-time systems explaining UML notation of static structures and timed state machines. Then, it describes the principles of our approach for translation of a UML model into UPPAAL timed automata and the tool support to automate the translation process. Section V describes a real telecommunication case study to demonstrate applicability of our approach. In addition, it describes the TRON test setup to perform model-based conformance testing. Section VI concludes the paper, while discussing future work.

II. Related Work

In the context of the UML model validation, Richters and Gogolla [8] propose the USE animation-based tool for validation of UML models and OCL constraints. We propose using the UPPAAL tool which integrates validation and verification processes. Later on, the obtained UPPAAL timed automata can also be used as input to the UPPAAL-TRON testing tool for test generation.

Work on verification of the UML-based design of real-time systems has been published by several authors. Similar to our approach, many of these authors base their approaches on the UPPAAL model checker and on the translation of UML to the input language of the UPPAAL, but in general they use different elements of UML.

A translation of the UML timed sequence diagrams into UPPAAL timed automata has been presented by Firley et al. [9]. Sequence diagrams specify required sequence of message between objects, but they are too weak to specify stronger properties like state invariants. In contrast, our approach uses class and state machine diagrams which are richer in expressing the system properties.

Similar to our work, Ober et al. [10] use class diagrams and state machine diagrams to capture the structure and the behavior of the system respectively. They utilize the IF toolset [11] to analyze the model and propose a translation from UML 1.4 to input language of IF, though no implementation of IF seems to be available. David et al. [12] suggest the time extension of the state diagram by adding clocks, timed guards, and invariants. However, their approach mainly focuses on flattening the hierarchical timed automata. Moreover, the event communication between processes has to be coded by hand.

A prototype tool called Hugo/RT has been presented by Knapp et al. [13]. It uses UML collaboration (sequence diagram) with time constraints and a set of timed UML state machines as input for the tool. However, their approach has several limitations. Most prominently, the input/output events between IUT and its environment model cannot have parameters. Muniz et al. [14] discussed an approach for verification of real-time systems represented for the CORBA component model. In their approach, UPPAAL is deployed for verification purposes and their tool called TANGRAM takes UML component and state machine diagrams to generate the equivalent UPPAAL timed automata. They extended the component diagram with a stereotype to model *event* passing between components. This mechanism does not allow to have parameterized events like [13]. Compared to these approaches, we use the UML interface element to model parameterized event passing.

III. Background To Timed Automata and UPPAAL

A. Timed Automata

According to theory of timed automata [15], a *timed automaton* is a non-deterministic finite state machine accompanied with *clock* to express timing properties. Clocks can be set to zero and their value increases linearly with time. At any instant, the value of a clock is equal to the time elapsed since the last time it was reset. The state of a system of timed automata includes the control state, variables and the clocks. Execution of timed automata are infinite sequences of system states that fulfil the invariants which may be either the passing of time or running of transitions. A transition is enabled either separately or synchronized with another automaton. The transition is taken when the associated time constraint is satisfied and its guard expression evaluates to true in the system state.

B. UPPAAL

UPPAAL is a tool-suite for modeling and model checking of real-time systems. It uses an extended version of timed automata, called UPPAAL Timed Automata (UPTA), to specify a system as a network of timed automata consisting *locations* and *transitions*. The behavior of the system is expressed by transitions (called edges in UPPAAL) between these locations. UPPAAL enriches the notion of *timed automata* by allowing to declare bounded integer variables in a automaton locally either or globally. Structured data types, user defined functions, binary channel synchronization, and broadcast channels are other UPTA extensions to timed automata. Moreover, it defines *urgent* and committed locations. In *urgent* location time is not allowed to pass as long as the location is active. Additionally, leaving the *committed* location has precedence over other possible transitions.

The channel synchronization between processes is denoted with $a?$ for the sending process, and with $a!$ for the receiving process. This way, several transitions are enabled simultaneously, but the assignment(s) in the *sending* automaton (with $a!$ label) is executed before the *receiving* automaton (with $a?$ label). This enables communication in a network of concurrent automata with the help of *global variables*. Value assignment and clock resetting can be two possible actions when a transition is enabled. It has to be noted that a transition is not taken when the resulting system state would not satisfy the associated invariant with the target location. The next system state is achieved by updating the control states of the timed automata involved in the transition by performing its defined actions. Furthermore, UPPAAL uses the idea of invariant which is a progress condition imposed on the location, that is, the system is not allowed to stay in the location more than the value mentioned by the invariant.

C. UPPAAL-TRON

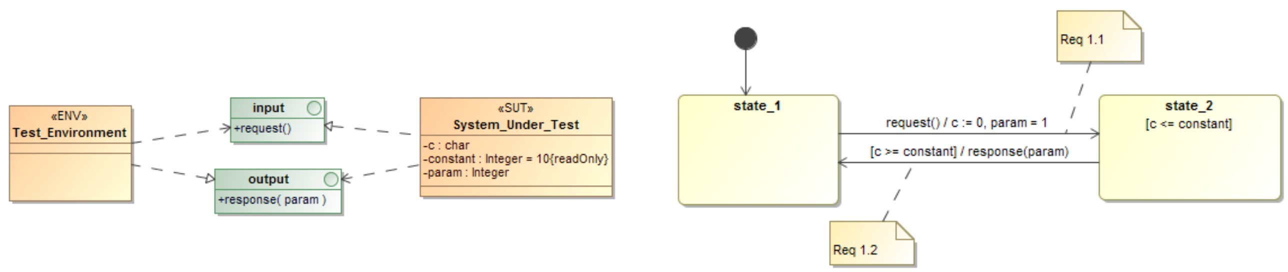
The UPPAAL-TRON tool – or simply TRON – is an extension to the UPPAAL tool for conformance testing of real-time systems, designed according to *relative timed input/output conformance relation (**rtioco**)* [5]. The test specification in TRON is partitioned into a model of the environment and a model of the SUT. These two models communicate using input/output channels. TRON attaches to the IUT via a *test adapter* which is a physical interface to enable communication between the testing tool and an implementation under test.

IV. Description of the Approach

Throughout this section, we describe the main features of our approach to translate a UML model of a system, including *class diagram* and *state machines* into UPPAAL timed automata. More practical details and concrete examples can be found in [16].

A. UML Modeling

A real-time system interacts with its environment via input/output actions (from SUT's perspective). This work utilizes two types of UML diagrams to represent a real-time system and its environment: 1) a *class diagram*, describing SUT and its test system environment, and 2) the corresponding



(a) Class diagram specifying SUT and its environment

(b) State machine diagram describing behavior of SUT

Figure 1. Abstract UML model of a system.

state machine diagrams specifying the behavior of each class element.

The class diagram describes the entities involved in a test process: SUT and its environment testing system which are communicating using dedicated protocols. SUT is a real-time system taking input form the environment via communication networks and producing output to it. In our approach, class elements are used to represent all entities in a system which typically consists of SUT and its environment. We also deployed two model elements using *stereotype* extensibility mechanism in UML to distinguish SUT and its environment testing system in the system architecture model rendered as «SUT» and «ENV» respectively. The defined stereotypes are derived from the *base element* in UML. In our approach, all classes have to be stereotyped before proceeding to the translation into UPTA. We also allow for several classes to be stereotyped as SUT or as environment. At the testing time, the partitioning will be used for identifying the test interface.

We specify communication between entities via *interface* elements containing a set of operations. The interface specifies the operations which a given class (*supplier*) can provide to other classes (*clients*). The class can have attributes of type `integer` or `char`. The latter are used to define clocks in our UML model. Figure 1 shows an illustrative example of a system model including a SUT and its environment. The architecture of the system is depicted by the class diagram in Figure 1a, showing two class elements named *Test_Environment* and *System_Under_Test*. The *Test_Environment* sends a *request* message to *System_Under_Test* via *input* interface, and receives a *response* message accordingly.

Each class has associated state machine describing the behavior of the class element in terms of *states* and *transitions*. Figure 1b describes the dynamic behavior of SUT showing an *initial state* and two *simple states state_1* and *state_2*. A state can have *time invariant* specified as Boolean expression, (e.g., the SUT state machine is not allowed to stay in *state_2* more than *constant* time units after entering the state).

Events are triggers of transitions between states and response actions become *effects* on the transitions. In real-time systems, an event can be either *call event* or *time event* to trigger a transition. A fully defined transition includes a *trigger*, a *guard*, and an *action*. UML uses the following syntax for transitions:

```
event trigger(parameters)[guard]/action(s)
```

The guard condition is a Boolean expression which has to be met in order to fire the transition. The actions are executed only if the transition is taken. Transitions without any explicit trigger are triggered by an implicit *completion event* which occurs when all activities of the source state have been finished. In fact, it is handled like a time event with duration of 0 time units.

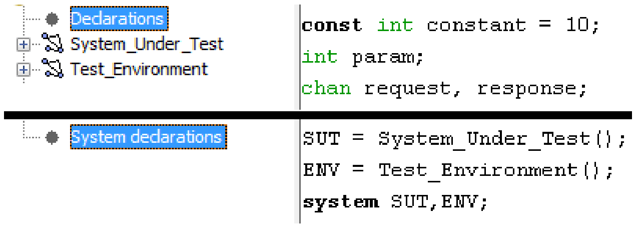
Specifying requirements. Requirements are modeled using SysML *requirements diagram* [17] and linked to different transitions in state machines, with the purpose of showing which requirements are fulfilled when a certain state is reached. An example of the approach can be found in [18]. For readability, in this paper, generic requirements are attached to transitions via a UML *comment* elements. For instance, in the state machine in Figure 1b, *Req 1.1* is achieved when the corresponding transition is taken and the state machine enters *state_2*.

B. *Translation from the UML model into UPTA*

A translation from UML models of real-time systems including class and state machine diagrams into UPTA consists of several steps as described below. Each step produces certain artifacts of UPTA.

1) Class element: class elements in class diagrams represent test entities in a test process. A class element in a UML class diagram whose behavior is defined by a state machine, is encoded by a timed automaton. Timed automata are represented by *templates* in UPPAAL. Templates are in turn instantiated to constitute the actual model.

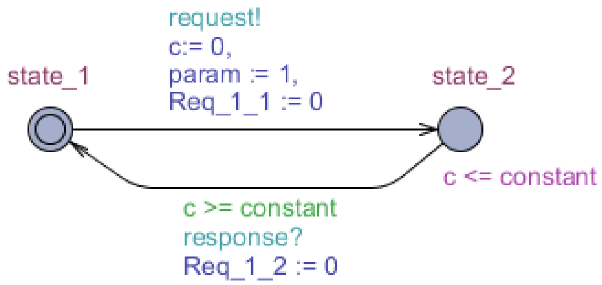
2) Interface and interface usage: A set of interface operations in the UML model is used as means of communication among test entities. The corresponding communication between templates in UPPAAL is represented by channel synchronizations. Each operation in an interface is translated into a binary synchronization channel in UPTA. The class element that realizes interfaces acts as the *receiving automaton*, whereas the class element that uses the interface acts as *sending automaton*. In addition, a list of interfaces in test adapter is created according to the interfaces between IUT and its environment. This list is used to generate Java source code including all input/output entries used by I/O handler as will be discussed later.



```
const int constant = 10;
int param;
chan request, response;
```

```
SUT = System_Under_Test();
ENV = Test_Environment();
system SUT,ENV;
```

(a) Declaration of SUT and its ENV



(b) SUT template

Figure 2. UPPAAL model of the example system.

3) Attributes: Class elements in UML class diagrams are inspected and for each attribute of integer type, a constant or an integer variable is declared in UPTA. For simplicity, all attributes with integer data type are declared globally in UPTA. UPPAAL only supports integer data types either as constants or variables. This approach takes advantage of this to represent `char` data type variables in class attributes as clocks. Consequently, these `char` variables are translated into the locally declared *clocks* of the corresponding timed automaton.

4) Superstates: In general, for each state in a UML state machine diagram, a single *location* is considered in a template. Initial and final pseudo-states in the UML state machine determine the initial and the final locations of the template, respectively.

5) Transitions: Each UML transition is represented by one or a sequence of *edges* in UPTA. Pertinent guards of a transition are copied appropriately to edge properties in UPTA. The trigger and effect actions of a transition are translated as receiving and sending binary synchronization channel respectively. In case a transition consists of a trigger and an effect action, it will be transformed by two *edges* and one *urgent location* in-between which the first edge is synchronized with the trigger and the second edge is synchronized with the effect action.

6) Requirements: Transitions in the UML model may have associated requirements. These requirements can be formulated as reachability properties and verified in UPPAAL. In addition, each requirement is translated into an auxiliary variable of type integer (initialized to 0) and attached on the corresponding edge in UPTA. These auxiliary variables are used during test generation for recognizing the coverage level or by formulating a property checking that an intended state can be reached or not.

7) Hierarchical state: UPPAAL does not support hierarchical locations. Thus, there is a need to flatten eventual hierarchical states in the UML state machines. This can be achieved by encoding hierarchical states as states of a flat timed automaton. Hierarchical states are replaced with several simple states so that the behavior of the system remains the same. Initial and final pseudo-states of sub-state machine are translated to *committed locations* in UPPAAL templates, and then, the transitions to and from sub-state machine are mapped to the corresponding committed locations.

C. Tool support

The transformation defined above is generic and can be used in conjunction with any UML based approach which follows the same modeling principles. To automate the transformation, a tool has been developed in Python as a MagicDraw [19] plug-in. As such, the transformation can be directly invoked from the GUI of MagicDraw and automatically produces equivalent UPTA and test adapter. An example of applying these transformations steps to the models in Figure 1 is shown in Figure 2.

V. The LTE Control-Plane Case Study

The applicability of our approach is demonstrated in a case study on the Long Term Evolution (LTE) [20] interface for cellular mobile telecommunication systems. The LTE network consists of the access network and the core network. Evolved Universal Terrestrial Radio Access Network (E-UTRAN) is the radio access network technology and Evolved Packet Core (EPC) is the core part. Together, they form the Evolved Packet System (EPS). The EPC consists of Packet Data Network Gateway (PDN-GW) router and Mobility Management Entity Serving Gateway (MME/S-GW) router. The latter is split into two parts: Mobility Management Entity (MME) – managing the control plane and tracking user equipment; and Serving Gateway (S-GW) – dealing with user plane IP packets. The E-UTRAN NodeB (eNodeB) network element is a central

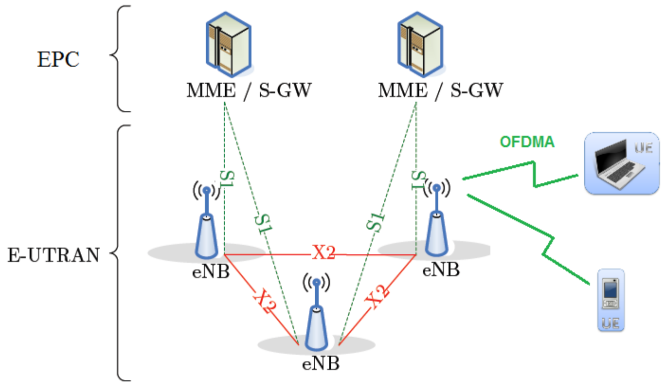
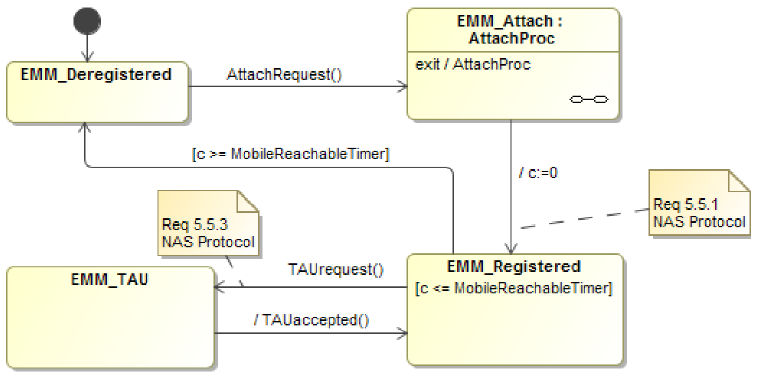
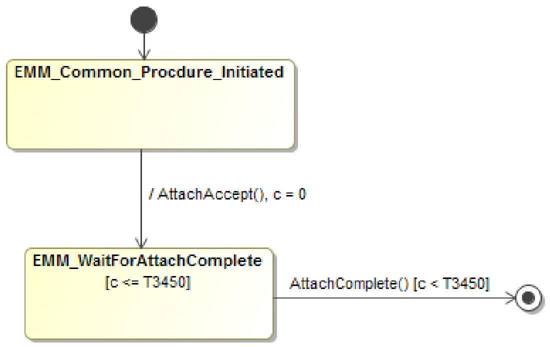


Figure 3. LTE Overall Architecture [20].



(a) The MME state machine


EMM_Common_Procdure_Initiated
/ AttachAccept(), c = 0
EMM_WaitForAttachComplete
[c <= T3450]
AttachComplete() [c < T3450]

(b) The MME attach sub-state machine

Figure 4. Dynamic Behavior of MME.

network element in the LTE infrastructure whose main functionality is to connect a User Equipment (UE) (e.g., a mobile phone) to the MME. The interface between the UE and the eNodeB is a radio interface, while the interface between eNodeB and MME, called S1AP, often is a fiber optic; refer to Figure 3.

Here, the main focus is on specific parts of the LTE control plane focusing on *Initial Attach* and *Tracking Area Updating* procedures from the EPS Mobility Management (EMM) layer from the None Access Stratum (NAS) protocols [21]. NAS is the highest stratum of the control-plane between the UE and the MME accounting for *mobility management* and *session management*. We designed a UML model to reflect the structure of UE and MME, and to express the behavior of aforementioned procedures, which are represented by class and state machine diagrams respectively. Implementations of the UE and of the MME were developed according to the UML model; however, only the MME will be used as SUT in this paper.

The main goals of the case study are: 1) to validate and verify the UML models of these two procedures regarding the NAS protocol requirements using the UPPAAL tool, and 2) to perform timed model-based conformance testing against the implementation of MME using TRON in order to determine whether the implementation conforms to the models.

A. *EMM specific procedures*

The *Initial Attach procedure* creates UE context when a UE is turned on and attaches to the network. According to Section 5.5.1 of the NAS Protocols, the UE sends a NAS *Attach Request* message to the MME via the eNodeB, starts timer T3410. The *Attach Request* reception in the MME is acknowledged with *Attach Accept* message and followed by starting timer T3450. Reception of the *Attach Accept* message by the UE causes to stop timer T3410. If timer T3410 expires prior to receiving an *Attach Accept*, the attach procedure is restarted. The MME also triggers the *update location* procedure, as well as the *route establishment* procedure. It communicates with Home Subscriber Server (HSS) and Home Location Register (HLR) in the update location procedure. S-GW is another entity that the MME communicates with it for route establishment procedure. After the bearers in the core network have been established, The MME tries to establish user-plane transport functions on interface between the UE and the eNodeB, as well as interface between the eNodeB and the MME. After establishment of user-plane, the UE sends *Attach Complete* message to the MME in order to confirm the assignment of user-plane tunnel. The MME supervises the reception of the *Attach Complete* by T3450 timer. However, in this case study, our main focus is on NAS protocols between MME and UE, making eNodeB, HSS, HLR, and S-GW irrelevant.

Based on Section 5.5.3 of the NAS Protocols, the UE must periodically perform tracking area updates procedure in order to update the registration of its actual tracking area in the network. This procedure is controlled in the UE by means of timer T3412. When timer T3412 expires, the tracking area update is started by sending *Tracking Area Update Request* to the MME. If this request has been accepted by the network, the MME shall send a *Tracking Area Update Accept* to the UE. The MME supervises the periodic tracking area updating procedure of the UE by mobile reachable timer which according to the protocol is 4 minutes greater than timer T3412. Upon expiry of the mobile reachable timer, the MME considers the UE to be inactive and performs *Detach procedure* to cancel the registration of this particular UE.

B. *UML models for SUT and the environment*

Here, we assume that the MME acts as SUT and the UE as its environment. However, having the implementation of both entities allows changing their role. Figure 4 displays the UML model for behavior of MME to support *initial attach* and *tracking area updating* procedures. The MME model is designed according to the procedures defined in the NAS protocols specification as explained earlier. The state machine of the MME in Figure 4a shows a hierarchical state named *EMM_Attach*. This gives modularity to the model and makes it easier to follow. The sub-state machine itself consists of one initial state, one final state, and two simple states, as presented in Figure 4b. The comment elements on the MME state machine named `Req 5.5.1` and `Req 5.5.3` express clearly the satisfying condition for the *initial attach* and *tracking area updating* procedures respectively.

C. Generating the UPPAAL model

Once the UML model of MME and UE includes all the necessary elements, it serves as input of the transformation tool to generate an equivalent UPTA using our tool. The resulting MME automaton in Figure 5 corresponds to the MME state machine in Figure 4 and exhibits the same behavior. It is worth mentioning that the hierarchical state machine in Figure 4 has been flattened automatically by the tool and included in the automaton of its parent.

1) Validation: The simulator tab of UPPAAL allows exploring the UPTA in a guided or random fashion without being exhaustive. When the simulation tab is selected, prior to the simulation phase, UPPAAL performs *syntax checking* which validates the UPTA with regard to *consistency*, *correctness*, and *completeness*. Once the syntax checking has succeeded, the UPPAAL simulator allows following the execution of the models visually, checking the instantaneous states and variables, and inspecting the communication trace between the UE and the MME parallel processes.

2) Verification: Different properties of the resulting model can be verified in UPPAAL. These properties are specified as queries written using a simplified version of Timed Computation Tree Logic (TCTL) [22]. The UPPAAL query language consists of the path and the state formulas. The path formulas quantify the paths or the traces of a UPTA with temporal logic, while state formulas describe individual states with regular logical operators.

As mentioned in the previous sections, our UML to UPTA translation automatically creates two types of queries. Firstly, we generate 'no deadlock' query to facilitate checking of this property in the system model.

```
A[] no deadlock
```

Secondly, we generate queries for checking the *reachability* property for the states whose incoming transition are tagged with the *comment element*. In our case study, the following query was produced by our tool, according to *Req 5.5.1* in Figure 4, and used by UPPAAL verifier to check whether the MME automaton eventually reaches the location *EMM_Registered*.

```
E<> MME.EMM_Registered
```

However, the *reachability* property does not guarantee the correctness of a system model, i.e., it just checks the basic behavior of the system model by performing such *sanity* checks. For instance, when the MME automaton enters the location *EMM_Deregistered* after the registration of a UE, the *mobile reachable* timer must have been expired. This requirement can be expressed with the following *safety* property:

```
A[] MME.EMM_Deregistered imply
                  MME.c >= MobileReachableTimer
```

D. TRON Test setup

The test setup for the MME entity of LTE includes TRON engine and its internal Socket Adapter, the TCP/IP Socket with input/output handler, and an implementation of MME as shown in Figure 6. The I/O Handler translates abstract inputs from TRON into concrete physical actions for the IUT. On the other hand, it recognizes physical output of the IUT and then encodes it into proper abstract message readable by TRON. The I/O Handler communicates with the TCP/IP Socket and the IUT via function calls. Communication between TRON built-in adapter and MME is done via TCP/IP.

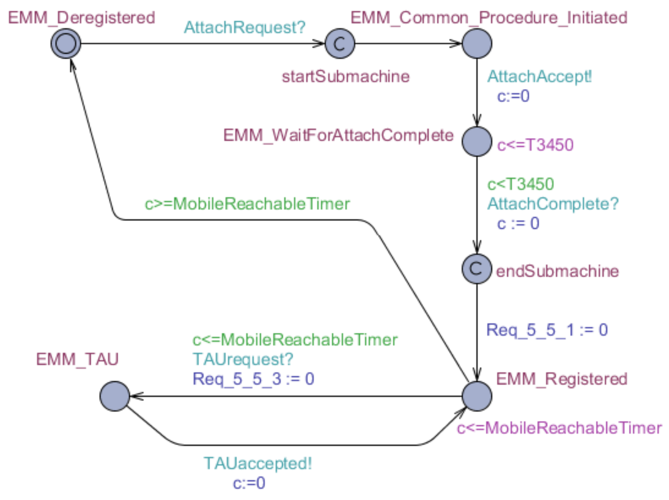


Figure 5. The MME UPPAAL Template.

Inputs in the implementation model are *AttachRequest*, *AttachComplete*, and *TAUrequest* and outputs correspond to *AttachAccept* and *TAUaccepted*, refer to Figure 5. TRON derives test cases directly from the environment model by choosing one of the possible inputs within allowed time delay at each state using the UPPAAL engine. It then executes them against an IUT and observes the output. Finally, it evaluates the correctness of a test experiment based on the model of IUT and determines the test verdict. Since TRON is an online testing tool, it keeps the connection to the IUT in real-time when performing all of the test procedure steps.

VI. Conclusions and Future Work

The proposed approach is aimed at increasing the quality of UML-based models of real-time systems via validation and verification using UPPAAL. For this purpose, we suggested an approach in which UML specifications are created and subsequently transformed into UPPAAL timed automata. Whenever a problem is discovered in the UPTA specifications, the UML model is updated and then re-transformed. Using this approach allows using UML and UPTA in a complementary fashion.

At UML level, our approach allows one to clearly identify the SUT and the test environment and to model their behavior and the communication interfaces. Via a set of mappings, we translate these models into UPTA. The translation also propagates requirement-related information which is then used to generate reachability properties.

The resulting UPTA specifications are also used for test generation using the TRON tool, which allows for generating and executing tests in timely fashion. One overhead of setting up the online MBT toolchain is the creation of the test adapter, which requires an initial investment followed by relatively

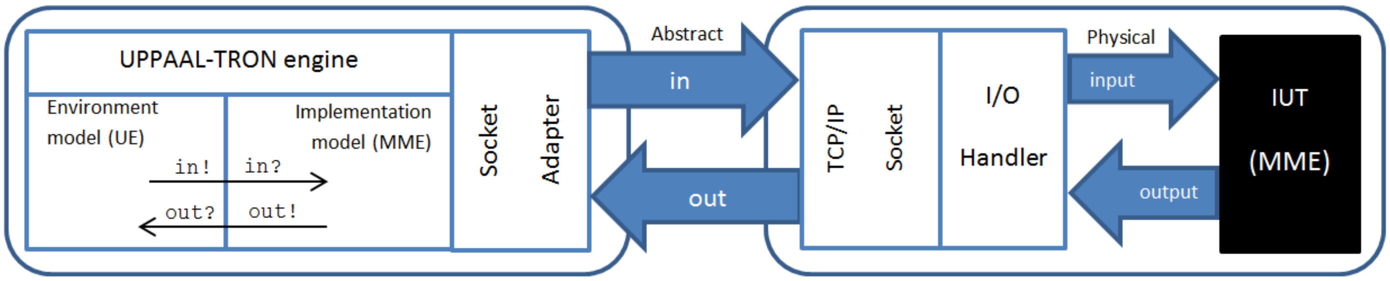


Figure 6. Specific TRON setup.

small updates each time the interfaces of the SUT is updated. In order to cut down on this initial investment, we generate a skeleton of the adapter during the transformation as described in [16]. Using TRON for model-based conformance testing, we managed to uncover a number of bugs in the implementation of MME which were addressed accordingly.

One current limitation of our approach is its scalability. Increasing the complexity of the specifications may result in a state space explosion in UPPAAL during verification and test generation. Although some ad-hoc optimizations can be considered to avoid this problem, we plan to search for a more systematic approach in future work.

In this study, we restricted ourselves to a limited set of UML model and extend this with real-time elements such as clock and state invariant. The clock expression in the UML state machine using the `char` data type is rather limited. Further research in this context will look into a more elaborated modeling of time and clock in UML. In addition, we will investigate how more UML diagram types can be included in our approach.

References

[1] (2013, August) Documents associated with unified modeling language (UML), version 2.4.1. [Online]. Available: http://www.omg.org/spec/UML/2.4.1/

[2] L. Lavagno, G. Martin, and B. V. Selic, *UML for Real: Design of Embedded Real-Time Systems*. Secaucus, NJ, USA: Springer, 2003.

[3] (2013, August) Documents associated with object constraint language (OCL), version 2.3.1. [Online]. Available: http://www.omg.org/spec/OCL/2.3.1/

[4] M. Utting, "The role of model-based testing," in *Verified Software: Theories, Tools, Experiments*, ser. LNCS. Springer, 2008, vol. 4171, pp. 510 – 517.

[5] A. Hessel, K. G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou, "Testing real-time systems using UPPAAL," in *Formal Methods and Testing*, ser. LNCS. Springer, 2008, vol. 4949, pp. 77–117.

[6] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on UPPAAL," in *Formal Methods for the Design of Real-Time Systems*, ser. LNCS. Springer, 2004, vol. 3185, pp. 200–236.

[7] K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou, "Testing real-time embedded software using UPPAAL-TRON: An industrial case study," in *Proc. 5th ACM international conference on Embedded software*, Jeresy, NJ, USA, September 2005, pp. 299–306.

[8] M. Richters and M. Gogolla, "Validating UML models and OCL constraints," in *«UML» 2000 - The Unified Modeling Language*, ser. LNCS. Springer, 2000, vol. 1939, pp. 265–277.

[9] T. Firley, M. Huhn, K. Diethers, T. Gehrke, and U. Goltz, "Timed sequence diagrams and tool-based analysis - a case study," in *«UML» '99 - The Unified Modeling Language*, ser. LNCS. Springer, 1999, vol. 1723, pp. 645–660.

[10] I. Ober, S. Graf, and I. Ober, "Validating timed UML models by simulation and verification," *International Journal on Software Tools Technology Transfer*, vol. 8, no. 2, pp. 128–145, 2006.

[11] M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.-P. Krimm, and L. Mounier, "IF: An intermediate representation and validation environment for timed asynchronous systems," in *FM '99 - Formal Methods*, ser. LNCS. Springer, 1999, vol. 1708, pp. 307–327.

[12] A. David, M. O. Möller, and W. Yi, "Formal verification of UML statecharts with real-time extensions," in *Fundamental Approaches to Software Engineering*, ser. LNCS. Springer, 2002, vol. 2306, pp. 218–232.

[13] A. Knapp, S. Merz, and R. Christopher, "Model checking - timed UML state machines and collaborations," in *Proc. 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Oldenburg, Germany, September 2002, pp. 395–416.

[14] A. L. N. Muniz, A. M. S. Andrade, and G. Lima, "Integrating UML and UPPAAL for designing, specifying and verifying component-based real-time systems," *Innovatioin in Systems and Software Engineering*, vol. 6, no. 1-2, pp. 29–37, 2010.

[15] R. Alur and L. D. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[16] M. Nobakht and D. Truscan, "Tool support for transforming UML-based specifications to UPPAAL timed automata," Turku Centre for Computer Science (TUCS), Tech. Rep. 1087, June 2013. [Online]. Available: http://tucs.fi/publications/view/?pub_id=tNoTr13a

[17] (2013, August) Documents associated with systems modeling language (SysML), version 1.3. [Online]. Available: http://www.omg.org/spec/SysML/1.3/

[18] F. Abbors, D. Truscan, and J. Lilius, "Tracing requirements in a model-based testing approach," in *Proc. First International Conference on Advances in System Testing and Validation Lifecycle*. Porto, Portugal: IEEE Computer Society, September 2009, pp. 123–128.

[19] (2013, August) MagicDraw webpage on NoMagic. [Online]. Available: http://www.nomagic.com/products/magicdraw/

[20] *ETSI TS 136 300 Evolved Universal Terrestrial Radio Access (E-UTRA) and Evolved Universal Terrestrial Radio Access (E-UTRAN); Overall description; Stage 2*, ETSI Std., Rev. V8.4.0, 04 2008.

[21] *ETSI TS 124 301 Universal Mobile Telecommunications System (UMTS); LTE; Non-Access-Stratum (NAS) protocol for Evolved Packet System (EPS); Stage 3*, ETSI Std., Rev. V8.10.0, 06 2011.

[22] R. Alur, C. Courcoubetis, and D. Dill, "Model-checking for real-time systems," in *Proc. Fifth Annual IEEE Symposium on Logic in Computer Science*, ser. LICS '90, 1990, pp. 414–425.