Model Transformation-Based Verification of ArchMDE Software Architecture By Using UPPAAL
From TURTLE Profile to Blackboard Style

Nourchene Elleuch
MOSIC Research Team
University Of Sciences
Tunis, Tunisia
nourchene.elleuch@gnet.tn

Adel Khalfallah
MOSIC Research Team
University Of Sciences
Tunis, Tunisia
adel.khalfallah@fst.rnu.tn

Samir Ben Ahmed
MOSIC Research Team
University Of Sciences
Tunis, Tunisia
samir.benahmed@fst.rnu.tn

Abstract—ArchMDE is a formal software architecture-based approach for the development and the verification of software architecture of real time systems. It provides a way to obtain the structure and the behavior of software architecture with hard real-time constraints and in accordance with blackboard style. This paper presents an overview of this approach and focuses on the behavior transformation rules. The timed automaton in UPPAAL is chosen as the semantic unit that allows analysis and verification of both functional and timing requirements in early phases of the development lifecycle.

Keywords- MDE; Meta-model; Model Transformation, Architectural Style; Real Time System; UPPAAL; Timed Automata

I. INTRODUCTION

The development of Real Time Systems (RTS) requires high costs and long delays due to their inherent complexity. New development methods and tools that address the temporal and concurrent aspects of the RTS behavior and structure should be elaborated in order to improve RTS developers’ productivity. Software Architecture-centric Development (SAD) [14] and Model Driven Engineering (MDE) [7] seem to be good candidates to deal with the following resulting challenges: a) MDE involves the systematic use of models, meta-models and models transformations as essential artifacts to make the development process cost effective. For that, different viewpoints are defined to design models: Computational Independent Model (CIM), Platform Independent Models (PIM) and Platform Specific Models (PSM) and a set of transformations may then be performed to make the system more platform-specific; and b) SAD is centered on the idea of reducing complexity through abstraction and separation of concerns. The software architecture is considered as the backbone for any successful software-intensive system and it is recognized as a first-class element when modeling a system. As a result, we notice the emergence of architectural styles, such as blackboard and filter architectures that can be used to guide the modeling and the evolution of software architecture in order to solve the recurrent architectural issue. In addition, the use of architectural style continues to be a vehicle for the improvement of software quality.

RTS engineering is a domain in which MDE can be helpful, particularly in addressing the problems of platforms and environments evolution as well as the problems of systems properties validation. For instance, authors [4, 13] describe how MDE can be used for the code generation of RTS and the author [8] describes the possibility of RTS validation by using test models. However, RTS model driven development is at its infancy and many challenges are still have to be overcome. The expression of architectural design decisions is one of the challenges [12]. In fact, the definition of architectural styles and the way to enforce their use and validation are still unresolved problems. This paper describes our approach “Architecture-centric Model Driven Engineering (ArchMDE)” [5, 6] that we propose as a potential solution in response to these issues. In fact, ArchMDE is a component-based software approach that adopts MDE to define blackboard software architecture with hard real-time constraints. It assists the systematic construction of the software architecture based on the use of meta-modeling and model transformations processes. The blackboard architectural style is treated as a meta-model and the appliance of architecture design decisions is set in terms of model transformations. However, a meta-modeling process defines only the syntax and the static semantic of blackboard component model.

The specification and verification of the correctness of composed component behavior is still a difficult issue to resolve. We choose to endow ArchMDE by a formal foundation to verify the correctness of composed component behavior. Thus, the elements of functional
architecture defined by TURTLE profile [2] (class diagram for structure point view and activity diagram for behavior point view) are being equivalently transformed into their counterparts (blackboard component diagram for structure point view and timed automata in UPPAAL [3] for behavior point view). The resulting model can be verified and validated using UPPAAL model checker.

This paper is structured as follows. We present in section 2 a brief overview of ArchMDE approach. Since ArchMDE is based on meta-modeling and model transformations, we describe in section 3 the majority of ArchMDE meta-model. In section 4, we describe the way how the semantic differences, between activity diagram and timed automaton, are bridged in order to define a set of transformation rules between TURTLE and UPPAAL. We explain, in section 5, the transformation rules applied to generate a behavior architecture-specific model. We enumerate, in section 6, various properties that must be verified in the resulting model. We discuss related work in section 7 and we conclude by providing perspectives and areas for future improvements.

II. ARCHMDE OVERVIEW

As software systems become more complex, the overall system structure and particularly software architecture, becomes a central design problem. Hence, we share the view of the author [12], who highlights that the architecture should be considered as “a first class modeling citizen” and specified as platform-independent implementation. Therefore, architectural requirements must be dealt at the PIM level which is itself decomposed into two models: an Architectural-Style Independent Model (AIM) and an Architecture Specific Model (ASM).

AIM and ASM are system models that don’t have any information about technology-specific implementation. AIM exhibits a specified degree of architecture independence in order to be suitable with a number of different architectural styles. It also describes the functional architecture of RTS. However, ASM is a refinement of AIM in which technology-independent architectural considerations are introduced. The main purpose of the ASM is to make architectural styles explicit in the model. The distinction between the two models increases the reusability of the AIM and allows a more control over possible types of implementations.

The ArchMDE engineering process is based on a top-down approach that supports the analysis, the design of functional, software and hardware architecture, the implementation and the verification phases through six steps. The starting point of ArchMDE is the requirements model (CIM) from which the AIM is derived. The AIM defines the structure and the behavior of the RTS. With respect to architectural style, the AIM can be transformed into the Software Architectural-Style Specific Model (SASM) that takes into account both the functional, non-functional and architectural characteristics. Adding Hardware Architecture Specific Model (HASM) features will give Operational Architecture Specific Model (OASM). Finally, the Platform Specific Model (PSM) can be derived from SASM and/or HASM from which source code can be generated (Fig. 1).

Figure 1. ArchMDE process and artifacts.

We use semi-formal language as the foundation of ArchMDE’s development process. But, it is mandatory to guarantee that the first specification of system is valid. Since formal methods allow this, we have endowed ArchMDE approach by verification process. This process is based on transforming any semi formal specification into a more formal one for the sake of validation and verification. On the one hand, the possibility of transforming an AIM into formal specification is ensured by the choice of TURTLE profile as a modeling language of AIM. TURTLE has a formal foundation that is given by the use of timed automaton in UPPAAL. On the other hand, the transformation of AIM into FASM is feasible when the definition of timed automata network meta-model and a set of transformation rules that allow the definition of architectural-style behavior is possible. We focus in the next section on models at level above architecture, particularly the software architecture and the formal-architecture levels.

III. SYSTEM MODELING AND META-MODELING IN ARCHMDE

ArchMDE takes into account both the structural and the behavioral aspects of architectural style. For that, we have compared, in [6], two languages Unified Modeling Language (UML) and Architecture Description Language (ADL) that are used to express the main concepts and connectivity rules of architectural styles. Instead of being faced to choose between ADL and UML languages, we wonder whether their merge can be considered to benefit...
from their advantages and avoid their drawbacks. This is what our ArchMDE’s software architecture meta-models have considered (Fig 2).

![Diagram of ArchMDE packages.](image)

Figure2. ArchMDE packages.

On the one hand, we define a new meta-model that considers the syntax and the static semantic of architecture elements to design the software architecture structure. The blackboard architectural style is defined by meta-model, which enumerates the essential elements that can be used to create software architecture according to blackboard. ArchMDE generates a network of communicating components that interact with each other by using shared data objects, posted on blackboard component. “One writer and several possible readers” is the synchronization policy of communication model. The user component reads object values emitted during previous period of other user component. A detailed description of ArchMDE software architecture structure is available in [6].

On the other hand, the software architecture behavior modeling is also based on meta-modeling techniques and particularly on model transformation process. The TURTLE activity diagram defined in AIM layer is transformed into timed automata network that specifies the system behavior according to blackboard style. The correctness of software architecture behavior is ensured by using UPPAAL model checking. But, the most challenge of the transformation process is: how to bridge the semantic gaps between the two languages? For that, we provide the similarities and differences between TURTLE activity diagrams and UPPAAL timed automaton (Table 1) based on three behavioral characteristics: concurrency, time and reactive behavior. Concurrency raises several challenges like scheduling, synchronization, and communication of tasks. In TURTLE, the synchronization is based on rendez-vous protocol with the possibility to data exchanged via gates and local declared variables (synchronization with or without communication). Scheduling protocol respects the composition operators defined into tclasses diagram. As an example, preemption operator is more prior than synchronous operator. As for UPPAAL, the synchronization between two (or several) processes (automata) is ensured by defining a global shared binary (or broadcast) channel. Over this channel, the data can be exchanged via globally declared shared variables (synchronization with or without communication). However, the scheduling policy must be designed by another automaton to define the priorities of processes and to make sure that time critical processes with hard deadlines are not delayed. The time model in UPPAAL is a continuous time. Technically, it is implemented as regions and the states are thus symbolic, which means that at a state we do not have any concrete value for the time, but rather differences [1]. This is the same thing in TURTLE profile.

TABLE I. SIMILARITIES AND DIFFERENCES BETWEEN TURTLE ACTIVITY DIAGRAMS AND UPPAAL

<table>
<thead>
<tr>
<th>Behavioral Characteristics</th>
<th>TURTLE</th>
<th>UPPAAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Concurrency</td>
<td>binary synchronisation by Rendez-vous with/without blocking read-write semantics</td>
<td>binary and/or broadcast synchronisation with/without values passing</td>
</tr>
<tr>
<td>Time</td>
<td>Continuous time</td>
<td>Continuous time</td>
</tr>
<tr>
<td>Reactive Behavior</td>
<td>TURTLE Activity diagram with synchronization and temporal operators</td>
<td>Timed automata network with channel, global and local data/clock</td>
</tr>
</tbody>
</table>

IV. TIMED AUTOMATON IN UPPAAL

A. Syntax and Semantic

The theory of timed automata is widely used to specify and verify the real-time systems. In this section, we briefly describe the syntax and the semantic of those automata used in the UPPAAL tool. An automaton is composed of a set of locations and transitions between these locations. A location can be tagged as urgent (time, but rather differences [1]. This is the same thing In TURTLE profile.

![Timed Automaton Metamodel.](image)

The meta-model of timed automaton in UPPAAL, as depicted in Figure 3, doesn’t describe the complete semantics of timed automaton. The main aim is to provide minimum concepts of timed automaton syntax, which can be used to describe a blackboard architecture behavior’s specification.

In UPPAAL, a system behavior is described by a set of processes that are considered as an instance of templates. The later is an extended timed automaton with parameters that can be of any type. A process can synchronize with others processes. This synchronization is often steered by defining binary, broadcast or urgent channel. The use of global variables allows the communication between processes. Moreover, each process can have its own local variables. The assignments to local or global variables can be attached to transition as so-called updates.
Each activity diagram defined in AIM layer is transformed into template in ASM layer. This transformation takes in account: the mapping between activity diagram elements and timed automaton and the characteristic of blackboard style that implies the introduction of blackboard and controller templates. We describe in the following sections the most important transformation rules.

A. Rule 1: ActivityDiagram2Template

Each activity diagram is transformed into template with the same name. Furthermore, we have defined, in the declarative part of template, a constant number that is used as identification.

B. Rule 2: Initial&FinalNode2Location

The initial and final nodes are transformed into two locations whose names are respectively id_s and id_f.

C. Rule 3: ActionNode2(Sub)Templates

In TURTLE activity diagram, the action node is used to design either internal or external synchronization on gate, variable assignment or method invocation. So, the transformation rules depend on the instructions for using the action node.

1) VariableAssignment2Edge&Location: An outgoing edge and a new location are created. The edge is labeled with the assignment expression. For that, we must declare the variable as local.

2) MethodInvocation2Edge&Location: Such as variable assignment, we define an outgoing edge and a new location. The method can be attached to edge as so-called updates. However, the architect must write the method algorithm in the declarative part of template.

3) ExternalSynchronisation2Templates: If the action node is used to describe an external synchronization on gates g1/g2 in order to exchange data value (Fig. 4), then we must apply the following sub rules.

a) BlackboardTemplate: If the blackboard template (Fig. 5) is not created yet in previous transformation steps, then we must define it. The communication in blackboard architectural style passes through a shared blackboard template. However, this communication protocol implies some issues such as starvation, deadlock and mutual exclusion. For that, we use write lock that either allows multiple templates to access the blackboard objects in a read-only way, or it allows one, and only one, template at any given time to have write access to the blackboard object. When the user template needs to communicate with others, it must obtain a write lock that allows to put data in the blackboard object. After, the user template could add their data, by using two global variables obj[tid][j] and syncval[tid][j], and thus releases the lock. tid is the template identifier ranging 0 to NT-1 where NT is the total number of processes (instance of templates). j specifies the maximum number of objects that the template can be defined on the blackboard. To prevent a change request when a blackboard is empty, the auxiliary functions setBEmpty(), isEmpty() and ModifyBEmpty() are defined. All of them use the variable empty initialized at true. The variable is set (setBEmpty()) when an addObject is performed and unset (ModifyBEmpty()) when all objects data are read.

b) ControllerTemplate: The focus of the controller template (Fig. 6) is to handle the best user template for reading blackboard object (trigger channel and best() function) when a change occurs in blackboard process (change channel).
c) **SynchronisationWithValueEmission2SubTemplate**: The write lock (WLock channel) is performed before the critical section is entered and the unlock (unlock channel) is performed after the critical section is left. By critical section, we mean any writing of data in the blackboard object. Only one process at any given time has a write access. We define a variable WriteLockedBy whose initial value is NT to check whether a process has write access rights. Infact, if the value of WriteLockedBy is different from the value of process identifier then the process must wait. The next transition checks the number of processes that gained a write lock. If the number is equal to zero then we assign the identifier value to WriteLockedBy variable. This operation is done by auxiliary function testWLock() (Fig. 7). The write operation ends at LeaveWL.

![Figure 7. T1 Template](image)

**d) SynchronisationWithValueReception2SubTemplate**: The controller template informs T2 template (Fig. 8) that blackboard contains new data. Then, the T2 template is synchronized with blackboard via read channel to retrieve interesting data.

![Figure 8. T2 Template](image)

D. **Rule 4: DeterministicDelay2SubTemplate**

Fig. 9.a. depicts that something is interpreted after 5 time units. The corresponding UPPAAL template (Fig.9.b) is composed by two locations and transitions. The first transition updates a clock (c) to zero. Hereafter, we define an invariant which stipulates that a system is not allowed to stay in the same state more than 5 time units. The transition guarded by \(c \geq 5\) has to be taken into consideration.

![Figure 9. Deterministic Delay with its transformation](image)

E. **Rule 5: Non-DeterministicDelay2SubTemplate**

Same as deterministic delay, the only difference lies in the transition guard (\(c \geq 0\). This change allows us to model that something is interpreted at most after 5 time units (Fig. 10.b).

F. **Rule 6: TimeLimitedOffer2SubTemplate**

In TURTLE, this operator means that the action \((g!x)\) is offered during a period which is less to 5. If the offer happens, the final node is interpreted. Otherwise, AD is interpreted (Fig.11.a). The first transition of corresponding UPPAAL template (Fig.11.b) updates a clock (c) to zero. Hereafter, the invariant \((c < 5)\) stipulates that a system is not allowed to stay in the state more than 5 time units, so that only one of the two transitions can be taken. Either, the transition guarded by \(c = 5\) has to be taken and the subtemplate associated to AD must be interpreted or the T1 subtemplate defined in rule 3.d must be occurred.

![Figure 10. Non-Deterministic Delay with its transformation](image)

![Figure 11. Time Limited Offer with its transformation](image)

VI. **VALIDATION OF TRANSFORMATION MODEL**

The presented model transformation method has been practically validated through a Cruise Control System case study. At the AIM layer, the system is designed with TURTLE profile. Then, it is transformed into a component diagram for a structural point of view and a network of timed automata from behavioral perspective. To validate the transformation rules, we classify properties into two categories. The first category deals with properties that are specific to blackboard style like deadlock and starvation free. The second category treats the properties which are specific to application domain like safety and liveness requirements.

We define a priority for channel to prevent a starvation problem. The Write Lock and read channels have the same priority, but the change channel has a higher priority. This results in the possibility of writer starvation. For that, the change channel is guarded by a race condition which implies that the transition being fired only when the condition becomes true. This condition requires the invocation of reader in case the buffer is either full or not empty. The absence of deadlock is checked by a single query \((\forall I \neg deadlock)\). As an example, we describe one bounded liveness requirement that the cruise control has to respect: when the brake pedal is pressed; the cruise control should be turned OFF within a period of time not exceeding 50 ms. This requirement is checked by a
query (obj[writeLockedBy][0]==PedPress) Reg.OFF and Reg.b<=50). The result is OK.

VII. RELATED WORK

Many different methodologies for the platform-independent design and the development of RTS have been proposed in the literature. For that, the adoption of executable UML has been widely studied. For example, xtUML [10] defines an executable and translatable UML subset for embedded RTS, allowing the simulation of UML models and the C code generation. However, there is no support to formal verification tools in xtUML. Similarly, the Accord/UML methodology [8] uses executable modeling which suggests that the validation process is based on simulation. HIDOORS [15] encompasses the possibility to automatically generate Java code from a PIM and supports the analysis and validation of real-time properties by using WCET analysis. The approach of SAE, based on the emerging standard AADL (Avionics Architecture Description Language) [9], provides a mean to specify both the software and hardware architectures, to map software into hardware elements (operational architecture) and to produce component implementations. Also, this approach lacks formal support to define behavior model. The MARTE profile doesn’t introduce or target new specific analysis techniques. Rather, it aims at representing well-known models from schedulability and performance analysis theories into a UML framework, based on transformation tools [16]. However, to our best knowledge, no experiences of architectural styles modeling using these approaches are currently published yet.

VIII. CONCLUSION

In this work we have presented ArchMDE approach, which focuses on the RTS development and verification. The combination of architecture-centric, model-driven paradigms and formal approaches are employed as an RTS development process and constitutes the contribution. We have proposed a way to transform the TURTLE elements into blackboard architecture elements in order to address the challenge of designing and adapting software architecture for RTS. The introduction of the architectural viewpoint has several advantages, which mainly are:

- The architectural style for RTS provides means for representing components and leads to a rigorous and clear software architecture.
- The structure and the behavior of software architecture are automatically generated by executing a mapping process.
- The software architecture allows for design decisions expression and enables the comprehension of the system at a higher level of abstraction.
- The formal behavior of software architecture increases the quality of RTS by providing a way to check logical and time consistency.

REFERENCES