Combining Aspect-orientation and UPPAAL Timed Automata

Dragos Truscan[1], Jüri Vain[2] and Martin Koskinen[1]
[1]*Åbo Akademi University, Turku, Finland*
[2]*Tallinn University of Technology, Tallinn, Estonia*

Keywords: Aspect-oriented Modeling, UPPAAL Timed Automata.

Abstract: We discuss an approach to combine aspect-oriented concepts with UPPAAL timed automata (UPTA) with the focus on providing a systematic constructive approach and automation tool support for model weaving. Our approach allows for decoupling the design of different aspects of the system and suggests the use of explicit composition patterns to weave the aspects together. We exemplify with an auto-off lamp example.

1 INTRODUCTION

Aspect-oriented modeling (AOM) (Clarke and Baniassad, 2005; France et al., 2004) is a paradigm inspired from *aspect-oriented programming* (Filman et al., 2005; Kiczales et al., 1997), which promotes the idea of *separation of concerns* in order to build more modular and easy to update specifications. An *aspect* describes a particular *concern* of the system from a particular *viewpoint*, allowing the developers to focus on individual features of the system in isolation.

An *aspect model* consists of an *advice model* (a model fragment describing a new functionality), a *pointcut model* (a model fragment specifying where the aspect can be composed to a base model) and a *composition protocol* (how the advice model is connected via the pointcut model). An aspect model can be woven with the base model in many places (called *join points*) and in different ways. The result of composing advice models and a *base model* is called *composite model*. The composition process is also called *model weaving*.

According to Sutton, *aspect-oriented software development* (AOSD) provides improved separation of concerns, ease of maintenance, evolution and customization, and greater flexibility in development (Sutton, 2006). Other researchers report in a survey of industrial projects (Rashid et al., 2010) that AOSD's main benefits are the substantial reduction in model size and the improved design stability. However, the main body of AOSD and AOM technologies provide a conceptual framework, leaving room for relatively loose semantic interpretation. Still, the main research challenge remains in hiding the complexities of the composition mechanisms from the user and in developing the associated tool support.

In this paper, we suggest the used aspect-oriented methods in the context of *UPPAAL timed automata* (UPTA) with the focus on providing a constructive approach accompanied by automated tool support for model weaving. Our suggestion would allow for decoupling the design of different aspects of the system and the use of explicit composition patterns to weave the aspects together. The approach will take advantage of the precise semantics of UPTA and their expressiveness when specifying behavioral aspects: incorporate timing constraints explicitly, multi-processes, synchronization and data structures. In addition, one can take advantage of the good tool support for model-checking in UPPAAL and of the available test generation tools using UPPAAL, which have been used in many industrial projects.

In the following, we discuss related works in Section 2. Section 3 will provide a short introduction to timed automata. Our aspect-oriented modeling approach is described in Section 4. Section 5 exemplifies our modeling approach. We conclude with a preliminary analysis of the approach and future work.

2 RELATED WORK

There is a large body of work applying AOM especially in the context of UML. The reader is deferred for more details to (Wimmer et al., 2011). However, aspect operation used in the context of UML lack clear semantics and constructive definitions. Al-

Truscan D., Vain J. and Koskinen M..
Combining Aspect-orientation and UPPAAL Timed Automata.
DOI: 10.5220/0005105101590164
In *Proceedings of the 9th International Conference on Software Paradigm Trends* (ICSOFT-PT-2014), pages 159-164
ISBN: 978-989-758-037-6

though UPTA is less expressive than UML, they allow more rigourous semantic definition. In our approach, we suggest the use of UPPAAL timed automata for specifying aspect models and their weaving without extending the formalism.

To our best of knowledge the only attempt to combine aspect-orientation and UPPAAL timed automata has been suggested by Sarna and Vain (Sarna and Vain, 2012). They provide an approach for including aspects in the construction of test models by formal refinements of UPTA specifications. The difference to our paper is that they used aspects for refining the system specification in place, whereas in this paper aspects for extending the functionality of the system with new features. Such an approach allows one to define features aspect-wise, while aspect weaving rules provide discipline for structural modeling.

3 PRELIMINARIES OF UPTA

An UPTA model M is a *closed network of extended time automata* $\mathcal{A}_1, \ldots, \mathcal{A}_n$, that are called processes. The processes are combined into a single system by the CCS[1] parallel composition $\mathcal{A}_1 \parallel \ldots \parallel \mathcal{A}_n$ (Bengtsson and Yi, 2004, Sec. 5.1, pp. 25). Synchronous communication between the processes is by handshake synchronization links that are called channels. Each channel has related to input (from the channel) and output (to the channel) actions. The action alphabet is assumed to consist of symbols for input actions denoted $ch?$, output actions denoted $ch!$, where ch denotes a channel name, and internal actions Act of $\mathcal{A}_1, \ldots, \mathcal{A}_n$. Asynchronous communication between processes is done by shared variables.

Each UPTA process $\mathcal{A}_i$ is given as a tuple $(L; E; V; Cl; Init; Inv; TL)$ where L is a finite set of locations, E is the set of edges defined by $E \subseteq L \times G(Cl;V) \times Sync \times Act \times L$, where $G(Cl;V)$ is the set of enabling conditions - guards. $Sync \subseteq \Sigma$ is a set of synchronisation actions over the channels the process $\mathcal{A}_i$ is linked to the network. In the graphical notation, the locations are denoted by circles and edges by arrows (see Figure 2). The set Act of internal actions is a set of sequences of assignment actions with integer and boolean expressions as well as with clock resets r. V denotes the set of integer and boolean variables. Cl denotes the set of real-valued clocks $(Cl \cap V = \oslash)$. $Init \subseteq Act$ is a set of assignments that assigns the initial values to variables and clocks. $Inv : L \rightarrow I(Cl;V)$ is a function that assigns an invariant I to each location, $I(Cl;V)$ is the set of invariants over clocks Cl and variables V. $T_L : L \rightarrow \{\text{ordinary, urgent, committed}\}$ is the function that assigns the type to each location of the automaton.

The semantics of UPTA $M = \mathcal{A}_1 \parallel \ldots \parallel \mathcal{A}_n$ is given in terms of *labelled transition systems* (LTS) (Bengtsson and Yi, 2004). A state of a network is a pair $\langle L, u \rangle$, where L denotes a vector of current locations of the network, one for each process $\mathcal{A}_1, \ldots, \mathcal{A}_n$ and u is a clock assignment reflecting the current values of the clocks in M. A network may perform two types of transitions, delay transitions and discrete transitions.

Besides clock variables UPTA may have boolean and integer variables, each with bounded domain and initial value. Predicates over these variables can be used as guards of the edges and they can be updated using resets on the edges. The semantics of the models that include such variables is extended in natural way, i.e. for an action transition to be enabled, the extended clock assignment must also satisfy all integer guards on the corresponding edges and when a transition is taken the assignment is updated according to the boolean, integer and clock resets.

To model atomic sequences of actions, e.g. atomic broadcast or multicast, UPTA support a notion of *committed locations*. A committed location is a location where no delay is allowed. In a network, if any process is in a committed location then only action transitions starting from such a committed location are allowed. Thus, processes in committed locations may be interleaved only with processes in a committed location.

4 INTRODUCING ASPECTS IN UPTA

As discussed in the introduction, the concerns of a system are developed in weakly related parts (models) called aspects. An aspect model is composed of a pointcut and an advice. Pointcuts identify points in the execution model referred to as join points.

With respect to UPTA, a pointcut can be a guard or set of guards applied to any combination of UPTA elements (model fragments) that are accessible via edges to which the pointcut guards are attached. Consequently, a join point is a place in the base UPTA model where the advice model is superimposed and the pointcut defines under what conditions the advice can be entered in the base model.

Both the base model and advice model are assumed to be UPTA and this model class is conservative under weaving operations described below. We suggest four types of advice weaving: *before*, *after*, *around*, and *conditional*. The first three follow the

[1] Calculus of Communicating Systems

same semantics as in AspectJ, while the fourth has a different one, as it will be discussed later on.

A composed model or woven model is a network of automata interacting via join points. The composition protocol is given by an adapter that implements the composition protocol. During the composition the same aspect can be woven in several places (join points) in the base model.

4.1 Generic Process

The generic weaving process is shown in Figure 1. In this figure, two independent UPTA models, *Base Model* and *Advice Model* implement two cross-cutting concerns, *Concern1* and *Concern2*, respectively. When the two models are composed a *woven model* is created.

Let *Base Model* be the base model to which the functionality of *Advice Model* is composed, resulting in a UPTA network. We define an *Adapter* as a model fragment which introduces weaving information in both models. Basically, the adapter introduces a *JoinPoint* in the base model, and the corresponding entry and exit points of the advice that matches to the join point. *JoinPoint* encodes one of the following composition rules: *before*, *after*, *around*, and *conditional*. These rules specify when the behavior introduced by the advice should be executed with respect to a join point and how the control should be returned to the base model. In our approach, we make several assumptions:

- Although the same advice model template can be shared between several join points of a base model, we assume that unique instances of that advice model are woven to each join point, i.e., no waiting or race for an advice.
- The execution of an advice is atomic w.r.t. its base model. That is, once an aspect advice model is entered from a join point, the base model waits for the aspect to complete before exiting the join point.
- An aspect model has one entry point and one or several exit points which return to the same join point.
- The base model and the advice model can be woven using UPTA-specific communication and synchronization assumptions, e.g. synchronizing the entry and exit of advice model with wait in the base model, sharing or refining data between base and advice model, etc.

4.2 Join Point Adapters

For the purpose of making weaving operators constructive, we suggest several join point adapters providing support for weaving before, after, around, and

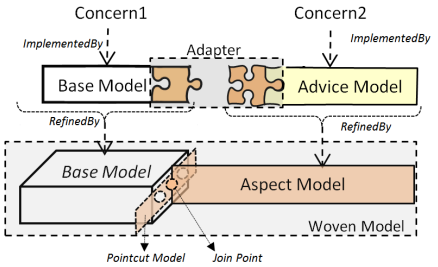

Figure 1: Generic weaving architecture for UPTA.

conditional advices, respectively. These adapters allow one to decide, based on the pointcut condition, whether the aspect should be invoked at a given join point. Our approach allows systematic and mechanized weaving of aspects into the base models.

The adapters we define can be applied for refining a channel synchronization, generically shown as the model fragment in Figure 2. The *channel* represents a synchronization between edges of parallel automata, whereas the direction of the synchronization is specified by suffixes of the channel name, e.g. *channel!* denotes the sending and *channel?* the receiving side of the channel. The synchronization can take place whenever both edges linked with a channel are enabled by their guards. During the synchronization, the variable updates specified on the synchronized edges are performed. In the following, in order to save space, we only present the adapters that can be applied to a receiving model fragment.

Figure 2: Model fragment with channel synchronization.

The *after adapter* (Figure 3) allows the execution of an advice after a channel synchronization. It refines the *End* location with two new locations *AspectStart* and *Call*, as well as with two new channels *enterAdvice!* and *exitAdvice?*. Whenever the *pointcut_expression* is true, the advice is executed, otherwise the base behavior is executed.

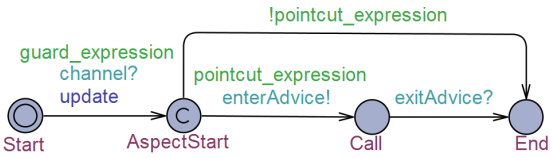

Figure 3: Generic *after* adapter.

The corresponding adapter introduced to the advice model during the weaving is shown in Figure 4. As one may notice, the execution of the advice model is triggered from the base model via the join point by receiving the *enterAdvice?* synchronization and, after

executing the advice functionality, it returns the control via the *exitAdvice!* synchronization.

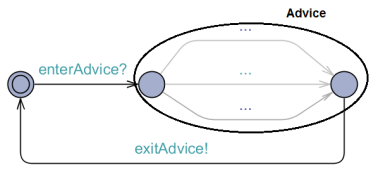

Figure 4: Generic advice.

The *before adapter* (Figure 5) allows the execution of the advice model before the base model reaches its fragment. The same generic advice model as in Figure 4 can be used.

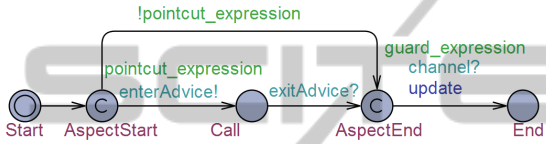

Figure 5: Generic *before* adapter.

The third adapter, the *around adapter* (Figure 6), allows the weaving of around advices by starting the execution of the advice model before the one of the base model fragment and returning from the advice model afterwards. The same generic advice model as in Figure 4 can be used. This is the most complex adapter type, and it can be used to both overload and override the functionality of the base model fragment.

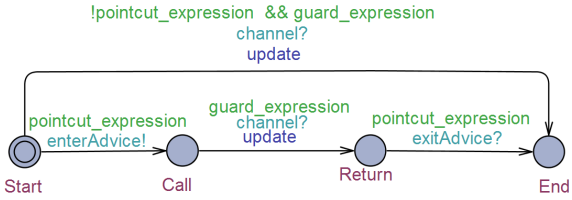

Figure 6: Generic *around* adapter.

Finally, the *conditional adapter* (Figure 7) introduces new functionality to the same base model fragment in Figure 2. The new functionality decides whether the execution of the base model continues after executing the advice or returns to a previous location. Compared to the previous adapters, the conditional adapter will allow the base model to consume the *channel?* synchronization, but the advice will decide if the same synchronization should be executed again via *exitAdviceRepeat* or the base model should proceed to the next location via *exitAdviceContinue*.

The corresponding generic aspect model for this advice is shown in Figure 8. As one may notice, this model can return to the join point via two different channels. If needed the adapter may be extended with

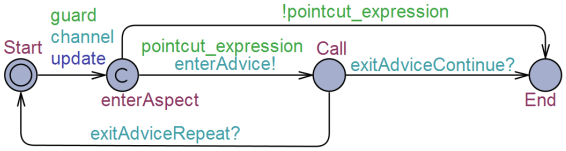

Figure 7: Generic *conditional* adapter.

more complex behavior, for instance with multiple exit points, which we defer for future work.

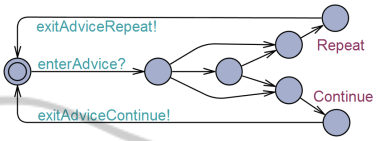

Figure 8: Generic conditional aspect.

Tool Support for Weaving. Since the composition mechanisms are specified explicitly via the generic adapters the weaving of aspects is completely automated via a Python-based tool. The tool has a graphical user interface in which the user can select from UPPAAL model files the template for the base model, the advice model template, the channel used as join point and the type of the advice. A new model file is created containing the woven aspect.

Verification of the Woven Model. Extending the base models with new functionality may imply the changing of timing behavior of the woven models. Since the weaving rules proposed do not pose restrictions on the advice models, there are no verification rules that could be specified generically, except the deadlock condition and the advice atomicity w.r.t. base model described in Section 4.1. Specific verification rules, including time-related ones, could be specified in TCTL (timed computation tree logic) (Alur et al., 1990) on a case-by-case basis.

5 CASE STUDY: AUTO-OFF LAMP

We exemplify our approach with an example originally found as a demo model in the documentation of the UPPAAL TRON tool (Hessel et al., 2008), under the name *auto-off lamp controller*.

5.1 Base Model-revisited

The purpose of the lamp controller is, that once it is tuned on, it will wait for a given time period before turning off, unless there are user inputs (“touches”)

which reset its auto-off function. The demo is composed of two models: a lamp and a user model. The lamp model (Figure 9-left) reacts to events, on the *touch* channel, and synchronizes to the user via the *done* channel. When the lamp is in the *OFF* location and the touch synchronization arrives, the internal clock *x* of the lamp is set to zero at the same time as the lamp transitions to the *switchON* location. The lamp is allowed to stay in this location for *tolerance* time units, during which it has to change the lamp-state modeling variable *n* to value 10 and synchronize its location to the environment on the *done* channel. After that, the lamp is in *ON* location, where it can accept new touch events.

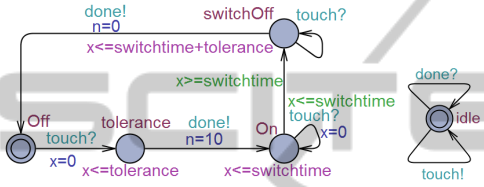

Figure 9: Original lamp (left) and user (right) models.

The lamp will stay in the *ON* location for *switchtime* time units, unless a touch event is received during its allowed stay and the clock is reset. When *switchtime* time units have elapsed the lamp transitions to *switchOFF* location and the *n* variable is set to zero. In this location the lamp will continue to accept touch events, even though these have no effect. A location change synchronization on *done* channel will take place from the *switchOFF* location to the *OFF* location within *switchtime* + *tolerance* time units. This implies that the lamp is allowed to stay in the *switchOFF* location for *tolerance* time units.

The model of the environment is presented in Figure 9-right. Its functionality is to emit touch events when the lamp is in a receiving state or to accept confirmation that the lamp level has changed.

5.2 Introducing New Functionality

We introduce two new orthogonal concerns to the lamp specification:
• Authentication: the user has to authenticate successfully before being allowed to change the lamp state from ON to OFF or reset the auto-off timer;
• Logging: log failed authentication attempts.

Each concern will be implemented separately as a stand-alone advice and woven in the base model of the lamp using the adapters described in Section 4.

5.2.1 Authentication Aspect

The first step is to create an advice model which implements the authentication. Since the authentication can result either in a successful or in a failed attempt, the advice model will have two exit points and thus should be compatible with a conditional adapter. The *authentication* advice (Figure 10) is entered via the *enAuth?* synchronization, after which, depending on the result of the authentication, it will exit via either *exAuthCon!* or *exAuthRep!*. For simplicity, the authentication is discriminated by a *pass* variable shared with the user model.

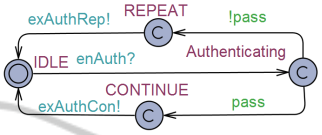

Figure 10: Advice handling authentication.

Intuitively, we would like to extend the behavior of our lamp model, to accept only touch events from authenticated users. If the user is not authenticated, the touch event is received but it has no effect on the lamp. In order to weave the authentication advice with the base model, the conditional adapter has to be applied to the base model in all possible locations. The target locations are all those edges having a *touch?* channel synchronization. The result of weaving authentication using the conditional adapter is shown in Figure 11.

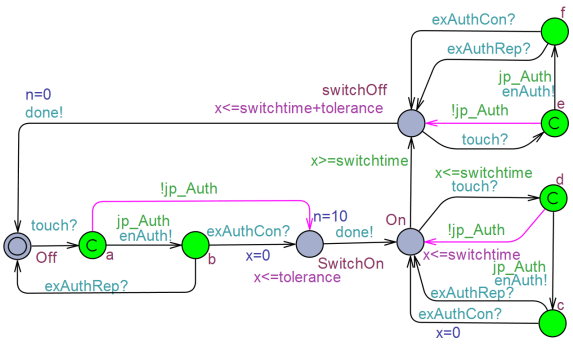

Figure 11: Lamp woven with authentication advice.

The behavior of the lamp model is the same with the base model whenever the authentication is successful. When the authentication fails, the control is returned to the location preceding *touch*, ignoring the user touch.

5.2.2 Logging Aspect

The logging aspect is introduced in a similar manner. Figure 12 shows and advice model for logging which has been already refined with an adapter. The advice is entered via the *enLog?* synchronization and it exits

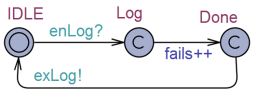

Figure 12: Logging advice.

via the *exLog!* synchronization after incrementing the number of failed authentications.

In order to weave this aspect with its base model, in this case the Authentication advice model, we refine *exAuthRep!* edge in Figure 10 using the before adapter defined in Figure 5, obtaining the model shown in Figure 13.

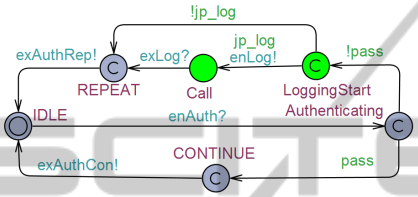

Figure 13: Weaving the Logging advice into Authentication.

Based on the new advice, whenever the authentication fails, the event is logged by invoking the Logging advice via the *enLog!* synchronization and after receiving *exLog?*, the original *exAuthRep!* is synchronized to the lamp model.

5.2.3 Environment With Authentication

The original user model in Figure 9 is updated to provide simple, both valid and invalid, authentication credentials via the global integer variable *pass*, as shown in Figure 14. Instead of having a complete set of correct and incorrect authentication credentials, we used only two values, one to represent all the successful cases and one to represent the unsuccessful cases. We also consider the authentication to take place at the same time as the touch event. Therefore the *touch* channel has not changed name. The new functionality could have been also introduced via a new aspect, but we decided to keep the example simple.

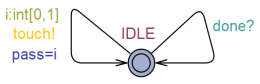

Figure 14: Environment model with authentication.

6 CONCLUSIONS AND FUTURE WORK

We suggested the introduction of aspects-oriented concepts in the context of UPTA models with the purpose of creating more modular, manageable and easy to update specifications. More specifically, we have proposed a set of generic adapters that can be used for systematic and tool-supported weaving of UPTA-based aspect models. Our approach allows to employ aspect-oriented concepts without modifying the underlying UPTA formalism. In addition, it allows one to take advantage of the verification engine of UPPAAL to ensure the validity of the resulting models.

Preliminary evaluation shows that creating and updating aspect models becomes easier following our approach and that weaving of models does not have a dramatic effect on the state space of the resulting models. However a more thorough evaluation is subject to future work.

REFERENCES

Alur, R. et al. (1990). Model-checking for real-time systems. In *Logic in Computer Science, 1990. LICS'90, Proceedings., Fifth Annual IEEE Symposium on e*, pages 414–425. IEEE.

Bengtsson, J. and Yi, W. (2004). Timed automata: Semantics, algorithms and tools. In Desel, J., Reisig, W., and Rozenberg, G., editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 87–124. Springer Berlin Heidelberg.

Clarke, S. and Baniassad, E. (2005). *Aspect-Oriented Analysis and Design. The Theme Approach*. Addison-Wesley.

Filman, R. E. et al. (2005). *Aspect-Oriented Software Development*. Addison-Wesley, Boston.

France, R. B. et al. (2004). An aspect-oriented approach to design modeling. *IEE Proceedings - Software, Special Issue on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, 151(4).

Hessel, A. et al. (2008). Testing Real-Time systems using UPPAAL. In *Formal Methods and Testing*, pages 77–117. Springer-Verlag.

Kiczales et al. (1997). Aspect-Oriented Programming. In *ECOOP '97 - Object-Oriented Programming*, volume 1241 of *LNCS*, pages 140–9. Springer-Verlag.

Rashid, A. et al. (2010). Aspect-Oriented Software Development in Practice: Tales from AOSD-Europe. *Computer*, 43(2):19–26.

Sarna, K. and Vain, J. (2012). Exploiting aspects in model-based testing. In *Proceedings of the Eeleventh workshop on Foundations of Aspect-Oriented Languages*, FOAL '12, pages 45–48, New York, NY, USA. ACM.

Sutton, StanleyM., J. (2006). Aspect-Oriented Software Development and Software Process. In Li, M., Boehm, B., and Osterweil, L., editors, *Unifying the Software Process Spectrum*, volume 3840 of *Lecture Notes in Computer Science*, pages 177–191. Springer Berlin Heidelberg.

Wimmer, M. et al. (2011). A Survey on UML-based Aspect-oriented Design Modeling. *ACM Comput. Surv.*, 43(4):28:1–28:33.