Modeling and Composing Scenario-Based Requirements with Aspects

João Araújo [†], Jon Whittle[‡], Dae-Kyoo Kim [Ψ]
[†]*Dept. Informática, FCT, Universidade Nova de Lisboa, 2829-516 Caparica, Portugal*
[‡] *QSS Group/NASA Ames Research Center, Moffett Field, CA 94035, USA,*
[Ψ]*Colorado State University, Fort Collins, CO 80523, USA*
ja@di.fct.unl.pt, jonathw@email.arc.nasa.gov, dkkim@cs.colostate.edu

Abstract

There has been significant recent interest, within the Aspect-Oriented Software Development (AOSD) community, in representing crosscutting concerns at various stages of the software lifecycle. However, most of these efforts have concentrated on the design and implementation phases. We focus in this paper on representing aspects during use case modeling. In particular, we focus on scenario-based requirements and show how to compose aspectual and non-aspectual scenarios so that they can be simulated as a whole. Non-aspectual scenarios are modeled as UML sequence diagrams. Aspectual scenarios are modeled as Interaction Pattern Specifications (IPSs). In order to simulate them, the scenarios are transformed into a set of executable state machines using an existing state machine synthesis algorithm. Previous work composed aspectual and non-aspectual scenarios at the sequence diagram level. In this paper, the composition is done at the state machine level.

1. Introduction

Requirements that cut across other requirements may result in tangled representations. Consequently, the reaction to requirements change is more difficult, since the impact of the change is more complicated to handle. It is therefore important to consider crosscutting requirements early in the software lifecycle.

The best way to deal with crosscutting requirements is to separate them from other requirements and model them independently. This modularization avoids tangled representations in the requirements document and facilitates requirements evolution. On the other hand, if no attention is paid to how the crosscutting requirements interact with other requirements, there is a danger that the nature of these interactions will only become clear during later stages of software development when problems are more costly to rectify. Hence, it is necessary *at the requirements stage* to have both a means of modeling crosscutting concerns independently but also a means of composing crosscutting concerns with other requirements in a way that will allow the entire set of requirements to be validated.

Aspect-oriented software development (AOSD) [4, 8] advocates the separation of crosscutting concerns (aspects) during development. However, most research in this area has focused on the design and implementation phases of the software lifecycle. In this paper, we consider aspects at the requirements level. In particular, we concentrate on scenario-based requirements. A simple example of an aspectual requirement scenario would be the description of the steps taken in response to a communications failure. The communications failure scenario is an aspect because it applies to all scenarios using the particular communication mechanism. Modeling the failure scenario without aspects requires that the failure steps must be considered in all affected scenarios. Our approach will model the failure scenario separately and only later will the failure behavior be composed with the affected scenarios.

A scenario is an example trace of desired or existing system behavior. Scenarios are commonly used in requirements engineering [1] because they are easily understood by all stakeholders. A complete and consistent set of scenarios can be difficult to specify, however, because there are a lot of non-nominal scenarios (e.g., exception, failure cases) to consider. Many of these are aspectual in the sense that they crosscut other scenarios.

Our process provides a way to describe aspectual and non-aspectual scenarios independently and then to merge them together for the purpose of validating the complete set of scenarios. In particular, we show how to separate aspectual scenarios and how to compose the behaviors represented by these scenarios so that the aspectual behavior can also become part of the validation effort.

Scenarios will be modeled, in this paper, as UML interactions (in particular, sequence diagrams) [15]. UML interactions, when used to model requirements, show the required behavior of several system components communicating towards a common goal. Interactions are a good way of modeling early requirements because they show global exchanges between system components and are very natural and intuitive to write down. Interactions give a *global* view of the requirements, but to simulate the requirements, a local view of each system component is necessary[1]. Finite state machines (FSMs) can be used to model the local, internal behavior of system components.

We will show how to model aspectual scenarios – that is, scenarios that crosscut other scenarios. Furthermore, we will provide a way of generating finite state machines from these scenarios in a manner that composes the aspectual and non-aspectual behavior so that the set of scenarios can be validated as a whole. As previously stated, scenarios will be modeled as UML sequence diagrams. Aspectual scenarios will be modeled as Interaction Pattern Specifications (IPSs) [6]. Finite state machines will be modeled as UML state machines and aspectual finite state machines will be modeled as State Machine Pattern Specifications (SMPSs) [9].

We present the modeling and composition of aspectual scenarios in the context of an iterative validation process. An initial set of scenarios is translated automatically into a set of UML state machines – aspectual scenarios (i.e., Interaction Pattern Specifications) become aspectual state machines (i.e., SMPSs). The aspectual and non-aspectual state machines are then composed by instantiating the aspects. The result is a new set of state machines representing the complete specification. These state machines can be simulated thus providing a convenient and easy way for the scenario-based requirements to be validated. During simulation, it is likely that new scenarios will be discovered or inconsistencies and ambiguities will be found in the way that the aspects interact with the non-aspects.

Previous results [17, 18] showed a similar iterative validation process in which aspects and non-aspects are composed at the scenario level. In this paper, the composition is done at the state machine level. More precisely, in the former approach, aspects and non-aspects are combined before state machines are generated from them. The separation of concerns is maintained at the state machine level by generating separately state machines from the aspects and state machines from the non-aspects. There are advantages and disadvantages to both approaches.

With composition at the scenario level, the state machines need never be seen by the requirements engineer. Composition is specified purely in terms of scenario relationships and the executable state machines that are generated can be hidden. This has advantages for requirements engineers not trained in state-based techniques.

On the other hand, composition at the scenario level tends to be rather coarse-grained. The user must provide composition operators that describe how to interleave messages from different scenarios. By composing instead at the state machine level, there is additional flexibility in describing the nature of the composition because composition can be defined in terms of states that are not specified in the scenarios.

Note that we do not advocate either of these solutions, but expect that each one will be appropriate in different contexts.

This paper is organized as follows. Section 2 presents the overall approach. Section 3 describes aspectual scenarios and state machines. Section 4 presents the process of composition of aspectual and non-aspectual state machines. Section 5 shows the application of the approach to an example. Section 6 gives some related work and finally, section 7 presents some conclusions and points directions to future work.

2. Overview of the approach

To situate our approach, we define a high-level process for developing and composing aspectual and non-aspectual behavior – see Figure 1. Use cases (i.e., functional requirements) are refined to a set of scenarios (also called interactions in this paper). Aspectual scenarios, i.e., scenarios that crosscut other scenarios, are represented as Interaction Pattern Specifications and non-aspectual ones as UML sequence diagrams.

Each aspectual or non-aspectual scenario is translated into a set of aspectual or non-aspectual state machines (one for each entity involved in the interaction). This is done using the Whittle & Schumann state machine synthesis algorithm [19].

The result of the synthesis algorithm is a set of state machines – each entity will have an aspectual and a non-aspectual state machine. The next stage of the process composes the aspectual and non-aspectual state

[1] at least, using current technology

machine for each entity. The result is an executable set of state machines that completely describe the requirements and in which aspectual and non-aspectual behavior has been merged. Validation of these state machines can now take place using either a simulation harness or a code generator and the results can be fedback into the overall process.

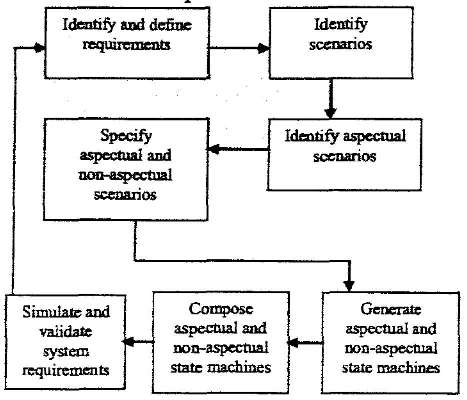

Identify and define requirements
Identify scenarios
Specify aspectual and non-aspectual scenarios
Identify aspectual scenarios
Simulate and validate system requirements
Compose aspectual and non-aspectual state machines
Generate aspectual and non-aspectual state machines

Fig. 1: Process model

Composing (or weaving) aspectual and non-aspectual state machines helps the requirements engineer grasp the full picture. To compose aspectual and non-aspectual state machines, we use the notion of binding and State Machine Pattern Specification. This approach requires that there is a binding statement for each application of the aspectual state machine. Interaction Pattern Specifications and State Machine Pattern Specifications are described in the next section.

3. Background

In this section, we present the required technical background for the rest of the paper.

3.1 Pattern Specifications

Pattern Specifications (PSs) [6, 9] are a way of formalizing the structural and behavioral features of a pattern. The notation for PSs is based on the Unified Modeling Language (UML) [15].

A Pattern Specification describes a pattern of structure or behavior and is defined in terms of roles. A PS can be instantiated by assigning modeling elements to play these roles. The abstract syntax of UML is defined by a UML metamodel. A role is a UML metaclass specialized by additional properties that any element fulfilling the role must possess. Hence, a role specifies a subset of the instances of the UML metaclass. A PS can be instantiated by assigning UML model elements to the roles in the PS. A model conforms to a pattern specification if its model elements that play the roles of the pattern specification satisfy the properties defined by the roles.

Pattern specifications can be defined to show static structure or dynamic behavior. In this paper, we will only be concerned with specifications of behavior but it should be noted that any class roles participating in pattern specifications must be defined in a Static Pattern Specification (SPS), which is the PS equivalent of a class diagram.

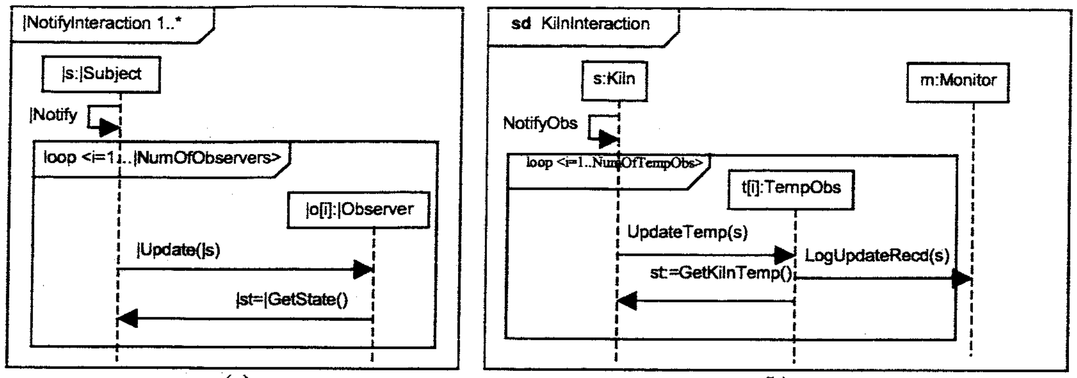
An Interaction Pattern Specification defines a pattern of interactions between its participants. It consists of a number of lifeline roles and message roles which are specializations of the UML metaclasses *Lifeline* and *Message* respectively. Each lifeline role is associated with a classifier role, a specialization of a UML classifier. Figure 2 shows an example of an IPS and a conforming sequence diagram (taken from [6]). The IPS in Figure 2(a) formalizes the Observer pattern. Role names are preceded by a vertical bar to denote that they are roles. An IPS can be instantiated by assigning concrete modeling elements to the roles. A conforming sequence diagram (see Figure 2(b)) must instantiate each of the roles with UML model elements satisfying the partial ordering on the message roles, multiplicity and other constraints (e.g., given in the Object Constraint Language [16]). In Figure 2, (b) conforms to (a) if we define instantiations as follows:

1. Bind |*NotifyInteraction* to *KilnInteraction*
2. Bind |*s* to *s*
3. Bind |*Subject* to *Kiln*
4. Bind |*o[i]* to *t[i]*
5. Bind |*Observer* to *TempObs*
6. Bind |*Notify* to *NotifyObs*
7. Bind |*Update* to *UpdateTemp*
8. Bind |*GetState* to *GetKilnTemp*
9. Bind |*st* to *st*
10. Bind |*NumOfObservers* to *NumOfTempObs*

Note that any number of additional model elements may be present in a conforming sequence diagram, namely, *m*, *Monitor* and *LogUpdateRecd* in Figure 2(b), as long as the role constraints are maintained.

A State Machine Pattern Specification (SMPS) defines a pattern of state-based behavior between its participants. It consists of a number of state roles and transition roles, which are specializations of the UML metaclasses *State* and *Transition* respectively.

An SMPS can be instantiated by assigning concrete modeling elements to the roles. Figure 3 shows an example of an SMPS and a conforming state machine



(a) (b)

Fig. 2: An IPS (a) and a Conforming Sequence Diagram (b)

where *S1*, *S2*, *T1* and *T2* have been instantiated to *A1*, *A2*, *P1* and *P2* respectively.

Notice again that additional modeling elements are allowed in the conforming diagram, Figure 3(b) – namely, *A3*, *P3* and *P4*. A state machine pattern specification captures the fact that a state machine diagram is an instance of an SMPS if the states and relative ordering of the transitions in the SMPS are preserved in the conforming state machine diagram.

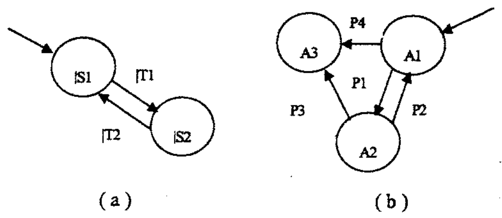


Fig. 3: An SMPS (a) and a Conforming State Machine (b)

An alternative way of representing patterns in UML is to use UML templates [2]. However, PSs are more flexible in terms of instantiation.

3.2 State Machine Synthesis

The following is a brief description of the algorithm used in this paper to synthesize state machines from a collection of scenarios (represented as UML sequence diagrams). The algorithm is a variant of the one described in [19].

Any algorithm that translates a set of scenarios into state machines must transition from a global scenario-based view (in which interactions between all system components are considered) to local component-based views (in which a state machine is given for each component). In general, a set of state machines can be executed whereas a set of scenarios (e.g., UML sequence diagrams) cannot because local models are needed for execution.

There are many algorithms for transforming scenarios into executable state machines. The interested reader is referred to [20] for examples. We merely give the flavor of the technique here. Synthesis of state machines is performed in two steps. First, each sequence diagram is converted into a set of state machines, one for each object [2] involved in the interaction. Next, the individual state machines derived for each object (from different sequence diagrams) are merged into a single state machine for that object.

In the first step, an individual sequence diagram is translated into a collection of finite state machines (FSMs). Messages directed towards a particular object are considered events in the FSM for that object. Messages directed away from an object are considered actions. The synthesis algorithm starts by generating an initial state for each FSM. It then traverses the sequence of messages. Messages have a unique sender object and a unique receiver object. For each message, a transition is added to the FSM for the receiver of the message where the transition is labeled with an event having the same name as the message. Similarly, a transition is added to the FSM for the sender with an action defined, where the action is to send the message.

We allow state labels in sequence diagrams[3] that explicitly label a state of an object in the interaction. State labels become named states in the generated FSM. State labels can lead to loops in the generated FSM if a state label occurs in multiple places.

[2] When describing the synthesis algorithm, we will refer to objects in scenarios. More generally, however, scenarios may represent interactions between components or subsystems and hence all discussion referring to objects equally applies to components and subsystems.

[3] State labels are similar to continuations in Message Sequence Charts and UML2.0 sequence diagrams.

Figure 4 shows an example of synthesis for a single sequence diagram. The state machine on the right is generated for B. *a/b* is the standard event/action notation for state machines (i.e., if event *a* occurs, then the transition fires which results in action *b* being taken, where for the purposes of this paper, actions always involve sending messages). The black boxes are state labels.

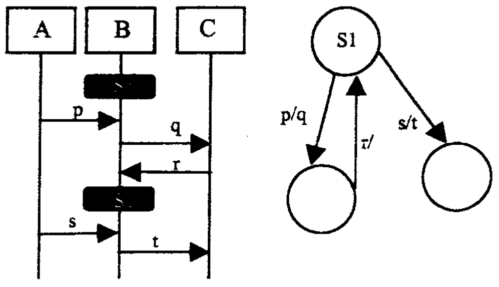


Fig. 4: Synthesis for a single sequence diagram.

Once finite state machines have been created for the individual sequence diagrams, the finite state machines generated from different sequence diagrams for a particular object are merged together. Merging state machines derived from different sequence diagrams is based upon identifying similar states in the FSMs. Our notion of similarity is based on identifying common incoming and outgoing transitions to states – see [19] for more details.

4. Composition of Aspectual and Non-Aspectual Finite State Machines

In the same way as [2, 7], we regard aspects as patterns. In particular, we represent aspectual scenarios as Interaction Pattern Specifications and aspectual state machines as State Machine Pattern Specifications. In [6, 9], an IPS or SMPS consists solely of role elements. We extend this definition to allow an IPS or SMPS to contain both role elements and non-role (i.e. concrete modeling) elements. An example of this in Figure 2(a) would be if the |*Subject* role were replaced with the concrete modeling element, *Kiln*, in the IPS.

Allowing non-role elements in an IPS and SMPS gives much greater flexibility in specifying aspects. For example, a security aspect might specify that any new user to a system must have their password checked. The actions to check the password will be the same for any user and hence should be represented directly as concrete modeling elements rather than role elements that must be instantiated. The inclusion of concrete modeling elements in the pattern specifications reduces the number of instantiation steps since roles that would be instantiated to the same elements in all contexts can be directly represented as those elements in the pattern specification.

Figure 5 describes the composition of aspectual and non-aspectual finite state machines using the synthesis algorithm and instantiation.

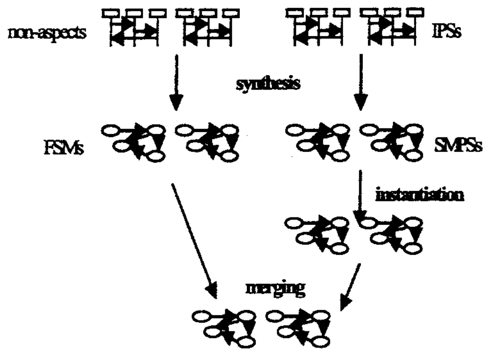


Fig. 5: Compose aspectual and non-aspectual FSM.

Non-aspectual scenarios are specified as sequence diagrams. Aspectual scenarios are specified as IPSs. These scenarios are merged together into state machines (one aspectual and one non-aspectual for each participating object) using the synthesis algorithm. The result is two sets of FSMs: one originating from non-aspectual scenarios and another obtained from aspectual scenarios. For each aspectual FSM, an instantiation is given within the context of those FSMs that it crosscuts. This yields a new set of FSMs (instantiated SMPSs). However, there will now be multiple state machines for the same object (i.e., one aspectual and one non-aspectual FSM) and so these state machines must be merged to yield a single state machine for the object that represents the combination of the aspectual and non-aspectual behavior.

As shown in Figure 5, we identify two distinct stages in the state machine aspect composition process. The first is *instantiation* in which SMPSs are given a binding statement. The second is *merging* in which the bound SMPS is woven with existing non-aspectual FSMs. These two stages are described in the following sections.

4.1 Instantiation and Merging

In previous work [17, 18], instantiation and merging were considered at the scenario level, i.e., the IPSs were instantiated and merged with the non-aspect scenarios *before* synthesis took place. In what follows, we describe a method for instantiation and merging at the state machine level (i.e., *after* synthesis takes place).

An important issue surrounding instantiation and merging is the level of input required from the user. Instantiation is a manual process because the bindings have to be provided for each case in which an aspect crosscuts a non-aspect. Merging, however, can be partially automated. Our approach is to define an algorithm for automating the process of merging but to allow fine-tuning of this algorithm by user input.

4.2 Codification of Instantiation

Instantiation is the process of binding role elements to concrete modeling elements. For state machines, the binding consists of mapping state roles and transition roles. The transition role mapping is a one-to-one mapping of the labels on the transitions. For state roles, we allow a many-to-many mapping. In some cases, a one-to-one mapping is enough but greater flexibility in the way merging is done is achieved via a many-to-many mapping. In order to support automatic merging in the presence of a many-to-many state role mapping, additional information will be required from the user, as follows:

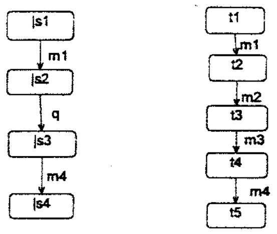
1. If the state role mapping is one-to-one, no further input is needed from the user.
2. If the state role mapping is many-to-one, then multiple state roles are mapped to the same image state. This is interpreted as meaning that the image state becomes a composite state with the state roles as sub-states. Any transitions directed to (or from) the image state must be re-directed towards (or away from) one of its new substates. The user must give the substate that the transitions will be directed towards or away from.
3. If the state role mapping is one-to-many, then a single state role maps into multiple image states. Because of this, for a transition directed towards (or away from) the state role, it is not clear which image state the transition should be directed towards (or away from). Hence, this additional information must be provided by the user.

We will refer to the additional user input required in cases (2) and (3) as *merging directives*.

The result of instantiation is defined by a mapping θ, that binds transition roles to concrete transitions and state roles to concrete states, and merging directives if θ is many-to-one or one-to-many. For the rest of this section, we assume, for simplicity, that the transition role mapping is the identity.

4.3 Codification of Merging

We will refer to the example in Figure 6 during the description of the merging algorithm.



(a) *A* (b) *X*

Fig. 6: SMPS (a) crosscuts FSM (b).

The left hand side of Figure 6 is an aspect SMPS and the right hand side is a non-aspectual FSM. To simplify things, we have assumed that the event roles on the transitions of the SMPS have already been instantiated to *m1* and *m4*. *q* is a concrete modeling element that occurs in the SMPS. Note that not all role elements in the SMPS need be given bindings (e.g., |*s3* will not be instantiated in our first example). The merging algorithm proceeds as follows. Denote the SMPS by *A* and the non-aspectual FSM by *X*. A new state machine, *Z*, is created with states as follows:

- Each state of *X* becomes a state of *Z*.
- For A, if θ maps multiple state roles of *A* to a single concrete state of *X*, then *X* becomes a composite state in *Z* and the state roles of *A* become substates of *X* in *Z*.
- For *A*, if θ maps a state role of *A* to a concrete state of *X* such that no other state role of *A* maps to that concrete state, then the state role of *A* is discarded (i.e., it does not become a state in *Z*).
- In all other cases, state roles of *A* become states of *Z*, where the name of the state role, |*a*, is mapped in *Z* to $\theta(|a)$.

The transitions of *Z* are created as follows:

- All transitions of *X* become transitions of *Z*. However, if a transition of *X* has a target state[4] in *X* that becomes a composite state in *Z*, then the transition must be redirected such that its target state in *Z* is a substate of the composite state. The merging directives tell the algorithm which substate the transition should be redirected to. Similarly, if the transition of *X* has a source state mapped to a composite state

[4] The originating state of a transition will be called its source state. Similarly, the destination state of a transition is its target state.

in *Z*, then the source is redirected to a substate of the composite state.

- A transition of *A* whose source state and target state roles are each mapped to zero or one state under θ, becomes a transition in *Z*.
- If a transition of *A* has a source state role that is mapped to multiple states under θ, the transition of *A* becomes a transition of *Z*, but it is redirected so that its source state becomes the state specified by the merging directives. Similarly for a transition of *A* that has a target state role mapped to multiple states under θ.

Note how the merging directives required in the case of a many-to-many mapping θ are reflected in how transitions are created in *Z*. The many-to-one mapping of state roles is taken care of by the first bullet point, which ensures that transitions in *Z* are connected to the correct states in the composite states of *Z*. The one-to-many mapping of state roles is taken care of by the third bullet point – transitions connected in *A* to a state role involved in a one-to-many mapping must be told which states they should connect in the final state machine *Z*.

As written, the rules for constructing *Z* may result in duplicate transitions in *Z*. We leave it up to the algorithm implementer to remove copies of transitions.

Returning to Figure 6, suppose that the SMPS on the left-hand side is bound to the FSM on the right-hand side by the mapping $\theta(|s1)=t1$, $\theta(|s2) = t2$, $\theta(|s4) = t5$. Following the rules just defined for creating a new state machine *Z* results in Figure 7.

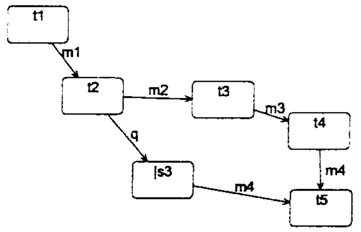


Fig. 7: Merged FSM and SMPS from Fig. 6.

The merging algorithm has matched the prefix and suffix of the two paths defined in the state machines and has created a branch in the centre.

We provide another example of merging at the state machine level that interleaves the events from the aspectual and non-aspectual state machine. In this case, the mapping θ is defined by $\theta(|s1)=t1$, $\theta(|s2) = t2$, $\theta(|s3) = t2$ & $t4$, $\theta(|s4) = t5$. Note that two state roles (*|s2* and *|s3*) in the aspect state machine have been mapped onto a single composite state (*t2*). As discussed above, the user must provide merging directives that re-route transitions into or out of this composite state. In this case, there are two transitions – *m1* and *m2*. The target state for *m1* is given by a merging directive as *|s2*. The source state for *m2* is given as *|s3*. Note also that *|s3* maps to multiple states. Hence, the user must define, for each incoming (and outgoing) transition of *|s3* which target (source) state the transition should be directed towards. In this example, we define that *q* should go to *|s3* in the merged state machine, whereas *m4* should leave from *t4*. The result of applying the merge algorithm is shown in Figure 8. Note that θ gives the user a way to control the ordering of *q* with respect to the other events. A way to provide such control is not obvious when merging at the interaction level. At the state machine level, however, we can redefine θ as follows: $\theta(|s1)=t1$, $\theta(|s2) = t3$, $\theta(|s3) = t3$ & $t4$, $\theta(|s4) = t5$. The result is a state machine similar to Figure 8 except that *t3* is a composite state rather than *t2* and the order of the events is *m1, m2, q, m3, m4*. This ordering satisfies the constraints of the original SMPS and hence the merged state machine continues to conform to the SMPS.

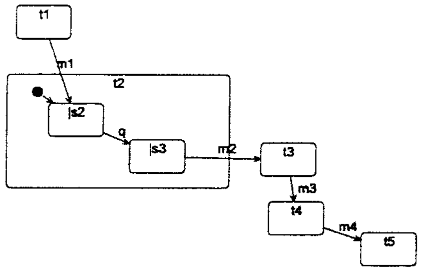


Fig. 8: Event interleaving during composition.

5. Example

We will illustrate our approach using a simple car parking example. The top-level requirements for the car parking system are as follows:

"To use a car parking system, a client has to get a ticket from a machine after pressing a button. Afterwards, the car is allowed to enter and park in an available place. The system has to control if the car parking is full or if it still has places left. When s/he wants to leave the parking place, s/he has to pay the ticket obtained (described above) in a paying machine. The amount depends on the time spent. After paying the client can leave by inserting the ticket in a machine which will open the gate for her/him to leave. Regular users of the parking system may pre-purchase time and enter/exit by inserting a card and PIN number which will result in money being deducted automatically from the user's account."

5.1 Identify use cases, aspectual and non-aspectual scenarios

By analyzing the requirements above, we identify the use cases Enter Lot, Exit Lot and Pay. Figure 9 shows a use case diagram for the example. Interaction scenarios can easily be identified based on the Use Case diagram.

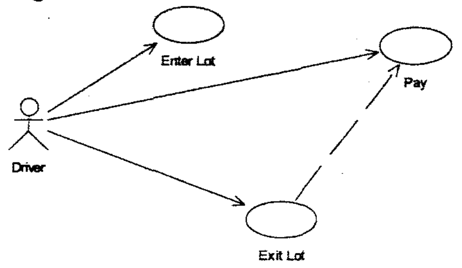


Fig. 9: Use Case Diagram for the Car Parking System

We refine each use case into a number of scenarios. Among these, some crosscut other ones. For example, some error-handling scenarios – i.e., how to react in the case of broken machinery, incorrect PIN etc. – can be modeled as aspectual scenarios if crosscutting is identified. This leads to the scenarios given in Tables 1 and 2, where I1-I11 are non-aspectual and A1-A3 are aspectual. For example, A1 is an aspectual scenario as it crosscuts the non-aspectual scenarios I3, I4, and I10.

Table 1: Non-Aspectual Scenarios

I1	Enter Lot, parking lot has space
I2	Enter Lot, parking lot has no space
I3	Enter Lot, regular user types in PIN and enters
I4	Exit Lot, driver inserts ticket; ticket paid
I5	Exit Lot, driver inserts ticket; ticket not paid
I6	Exit Lot, driver has no ticket
I7	Exit Lot, grace period from paying ticket exceeded
I8	Exit Lot, regular user types in PIN and exits
I9	Exit Lot, driver types in PIN but insufficient funds in account
I10	Pay, driver inserts ticket and correct money
I11	Pay, driver adds money to PIN card

Table 2: Aspectual Scenarios

A1	Machine is broken
A2	Ticket cannot be read
A3	PIN incorrect

5.2 Describe aspectual and non-aspectual scenarios

We will give a representative example for modeling the aspects using an Interaction Pattern Specification (IPS). Figure 10 shows the IPS for interaction aspect A1. If the machine cannot respond for any reason the supervisor is alerted and the driver receives an error message. The IPS contains four role names that must be instantiated to compose the aspect with UML sequence diagrams. The non-aspectual scenario I4 is given by the sequence diagram in Figure 11.

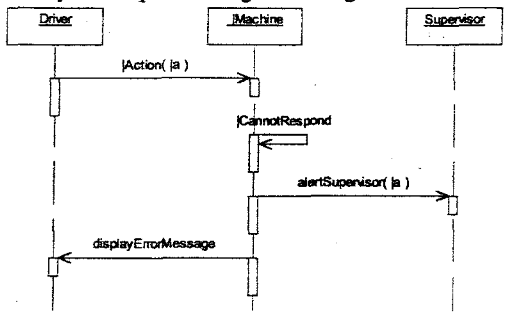


Fig. 10: IPS for the scenario "Machine is broken".

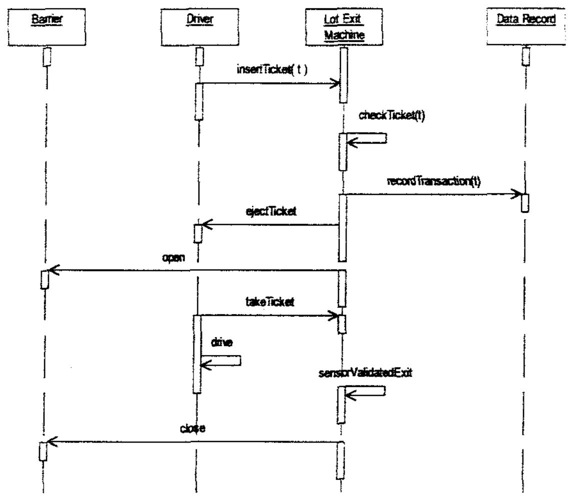


Fig. 11: Sequence diagram for exiting with paid ticket.

After the driver inserts the ticket, the Lot exit Machine checks it and if it is valid the transaction is recorded. Then the ticket is ejected and the barrier opens. Once the driver collects the ticket and leaves, the barrier is closed.

5.3 Generating SMPSs and FSMs

We now apply the synthesis algorithm to the scenarios in Figures 10 and 11. The algorithm produces an SMPS for |*Machine* (Figure 12) and a state machine for the *Lot Exit Machine* (Figure 13).

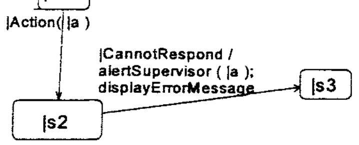

|s1
|Action(|a)
|CannotRespond / alertSupervisor (|a); displayErrorMessage
|s3
|s2

Fig. 12: State machine for |*Machine*.

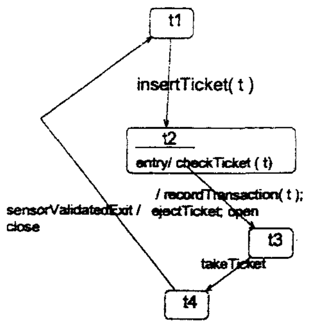


Fig. 13: State machine for *Lot Exit Machine*.

5.4 Instantiating SMPSs

Let us follow the process of instantiation for the SMPS for |*Machine* and state machine for *Lot Exit Machine*. There are five role elements in A1 (see Table 2) which must be given a binding, as follows:

1. |*s1* binds to *t1*
2. |*s2* binds to *t2*
3. |*Action* binds to *insertTicket*
4. |*a* binds to *t*
5. |*CannotRespond* binds to *timeout*

Note that bindings (1)-(4) bind role elements to concrete modeling elements in *Lot Exit Machine*'s FSM. Binding (5), however, binds to a modeling element that is not part of the *Lot Exit Machine*'s FSM. Given the bindings, composition is done automatically. The instantiated SMPS of |*Machine* is compared to the FSM of *Lot Exit Machine* and composition produces a new FSM that combines the behavior from instantiated |*Machine* and *Lot Exit Machine* in such a way that the new FSM contains all behavior from instantiated |*Machine* and *Lot Exit Machine* and, in addition, conforms to the original SMPS |*Machine*. The resulting new FSM is shown in Figure 14.

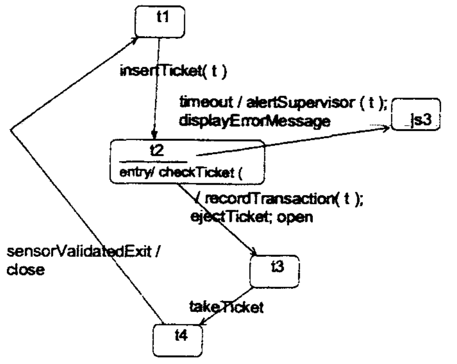


Fig. 14: Composed FSM.

State machines can also be generated for the other objects in the interactions and, after instantiation, the whole system can be simulated by injecting events using commercially available tools and the requirements validated by the users.

6. Related work

Some requirements approaches, such as viewpoints [5] and goals [11], address separation of functional and non-functional requirements. PREView [14] is a viewpoint-oriented requirements engineering method, which helps separate functional and non-functional properties of a system. Non-functional requirements are also separated from functional ones in goal-oriented approaches such as KAOS [3]. However, the identification of crosscutting requirements and their composition are not addressed explicitly.

Rashid et al. [12, 13] support separation of crosscutting properties at the requirements level. Composition rules are defined using XML. They use a list of constraint actions and operators, which are used to specify how an aspectual requirement influences or constrains the behavior of a set of non-aspectual requirements. Moreover, a conflict resolution scheme is presented, which is not addressed in our approach.

Georg et al. [7] propose an aspect-oriented design approach that defines an aspect through role models to be woven into UML diagrams. The approach is similar to ours in that aspects are treated as patterns. In particular, interaction aspects may be modeled as interaction role models. However, [7] does not allow concrete modeling elements in the role models. The addition of concrete modeling elements may be useful in practice to reduce the number of instantiations that the user must provide. In addition, [7] only considers instantiation for interaction role models, not composition of role models with non-aspectual interactions.

Clarke and Walker [2] use UML templates to define aspects. Interaction pattern specifications provide a much more precise way of defining aspects. [2] also is concerned more with how to specify the aspects rather than weaving aspects into non-aspectual models. Clarke and Walker compose static structural properties of aspects with non-aspectual class models, but do not compose interaction properties of aspects with interaction models.

There is also the work by S. Konrad and B. Cheng [10]. They focused on requirements patterns for embedded systems. However, pattern composition is not addressed in that work.

7. Conclusions

This paper presented an approach to modeling scenario-based requirements using aspect-oriented principles. Aspectual scenarios were modeled using Interaction Pattern Specifications (IPSs). A technique was described to compose aspectual and non-aspectual scenarios and to transform them into a set of executable state machines. In this way, we showed how to separate aspects during scenario development but also how to generate a composed behavioral description for simulating the scenarios.

The advantages of the approach are common to aspect-oriented software development in general: better modularization and traceability. This is reflected in the flexible and simple way that the merging algorithm is defined. Future work will address how to use the result of the simulation step to augment or correct the scenario models.

One issue that has not been directly addressed is scalability. The developer must provide binding statements for each aspect and for each scenario that the aspect crosscuts. We expect there to be ways to manage this complexity, for example, by providing default bindings. The scalability of the synthesis algorithm has already been evaluated on real-world examples and a Rational Rose add-in has been developed for creating and instantiating Pattern Specifications.

8. References

[1] I. Alexander and N. Maiden (eds.) *Scenarios, Stories, Use Cases*. John Wiley, 2004.

[2] S. Clarke and R. J. Walker, "Composition Patterns: An Approach to Designing Reusable Aspects", *Proceedings of International Conference on Software Engineering*, 2001.

[3] A. Dardenne, A. Lamsweerde, and S. Fickas, "Goal-directed Requirements Acquisition". Science of Computer Programming, Vol. 20, No., pp. 3-50, 1993.

[4] T. Elrad, R. Filman, and A. Bader (eds.), "Theme Section on Aspect-Oriented Programming", CACM, 44(10), 2001.

[5] A. Finkelstein and I. Sommerville, "The Viewpoints FAQ" BCS/IEE Software Engineering Journal, Vol. 11(1), 1996.

[6] R. France, D. Kim, S. Ghosh and E. Song, "A UML-Based Pattern Specification Technique", *IEEE Transactions on Software Engineering*, Vol. 30(3), 2004.

[7] G. Georg, I. Ray and R. France. "Using Aspects to Design a Secure System", *Proceedings of the 8th IEEE International Conference on Engineering of Complex Computer Systems*, Greenbelt, Maryland, 2002.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. "Aspect-oriented programming", *Proceedings of the European Conference on Object-Oriented Programming*, Vol. 1231, 1997.

[9] D. Kim, R. France, S. Ghosh and E. Song, "A UML-Based Metamodeling Language to Specify Design Patterns", *Proceedings of Workshop on Software Model Engineering (WiSME), at UML 2003*, San Francisco, 2003.

[10] S. Konrad, B. Cheng. Requirements Patterns for Embebed Systems. *IEEE Joint International Conference on Requirements Engineering*, Essen, Germany, 2002.

[11] A. van Lamsweerde, "Goal-Oriented Requirements Engineering: A Guided Tour". 5th International Symposium. on Requirements Engineering, pp. 249-261, 2001.

[12] A. Rashid, A. Moreira and J. Araújo, "Modularisation and Composition of Aspectual Requirements", *Proceedings of* the *2nd International Conference on Aspect-Oriented Software Development*, ACM Press, pp. 11-20, 2003.

[13] A. Rashid, P. Sawyer, A. Moreira and J. Araújo, "Early Aspects: A Model for Aspect-Oriented Requirements Engineering", *Proceedings of IEEE Joint International Conference on Requirements Engineering*, IEEE CS Press, pp. 199-202, Essen, Germany, 2002.

[14] I. Sommerville and P. Sawyer, "Requirements Engineering - A Good Practice Guide". John Wiley and Sons, 1997.

[15] Unified Modeling Language Specification, version 2.0, January 2004, In OMG, http://www.omg.org

[16] J. Warmer and A. Kleppe, *The Object Constraint Language: Getting Your Models Ready for MDA*, 2nd Edition, Addison-Wesley, 2003.

[17] J. Whittle and J. Araújo, "Scenario Modeling with Aspects", *IEE Proceedings Software*, to appear.

[18] J. Whittle, J. Araújo and D. Kim, "Scenario Modeling with Aspects", *Proceedings of Workshop on Aspect-Oriented Modeling with UML at UML2003*, San Francisco, 2003.

[19] J. Whittle and J. Schumann, "Generating Statechart Designs from Scenarios", *Proceedings of the International Conference on Software Engineering*, pp. 314-323. Limerick, Ireland, 2000.

[20] Workshop on Scenarios and State Machines: Models, Algorithms and Tools, *Proceedings of 25th International Conference on Software Engineering*, Portland, Oregon, 2003. http://www.doc.ic.ac.uk/~su2/SCESM