JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2005**

Vol. 6, No. 7, Special Issue: Aspect-Oriented Modeling, August 2007

Motorola WEAVR: Aspect Orientation and Model-Driven Engineering

Thomas Cottenier, Motorola Software Group / Illinois Institute of Technology
Aswin van den Berg, Motorola Software Group
Tzilla Elrad, Illinois Institute of Technology

Abstract

This paper presents an Aspect-Oriented Software Development (AOSD) language and methodology for Model-Driven Engineering (MDE) of large distributed applications, and a tool that implements these concepts, the Motorola WEAVR.

MDE technologies and development practices have been used for a long time in the industry, for the development of large critical infrastructure systems, especially in the telecommunication and avionics domains. While the benefits of MDE are established, these technologies still suffer from important technical limitations that hinder their adoption and reduce their potential benefits in terms of software development productivity.

MDE and AOSD exhibit some complementary properties. Modeling enables systems to be specified at higher level of abstraction but suffers from difficulties with respect to the refinement and integration of system perspectives. On the other hand, aspect technologies focus on the modularization and composition of concerns, but lack appropriate abstraction mechanisms.

The paper presents the WEAVR aspect-oriented modeling language and identifies three directions along which aspect models can help bridge the gap between system specification and implementation, while overcoming some of the weaknesses of Aspect-Oriented Programming languages.

1 INTRODUCTION

Modeling languages focus on partial abstractions of the product under development that highlight properties that are relevant from the perspectives of the different stakeholders, architects or developers. Model-Driven Engineering (MDE) [Bast03, Frankel03] technologies aim at automatically or semi-automatically translate those partial specifications into executable artifacts. Models defined at different levels of abstraction are iteratively expanded into executable artifacts through a series of manual refinements, mappings, and automated transformations.

Cite this article as follows: Thomas Cottenier, Aswin van den Berg and Tzilla Elrad: "Motorola WEAVR: Aspect Orientation and Model-Driven Engineering", in Journal of Object Technology, vol. 6, no. 7, Special Issue: Aspect-Oriented Modeling, August 2007, pp. 51-88 http://www.jot.fm/issues/issue_2007_08/article3.

Model-Driven Engineering for Communication Systems

The particular MDE environment discussed in this paper focuses on large, complex distributed software systems deployed as part of the global telecommunication infrastructure. These software systems typically have a very long life time, and their development may involve hundreds of software engineers. There is therefore a strong emphasis on architecture modeling and interface modeling to support independent development across different development teams.

Architecture modeling emphasizes specification, simulation and testing of the observable behavior of system components and their interactions. The architecture is an executable, skinny version of the system under development [Jacobson04]. It includes models that simulate the observable behavior of the system components for different use cases. These models act as behavioral contracts or interfaces between components developed by different development teams.

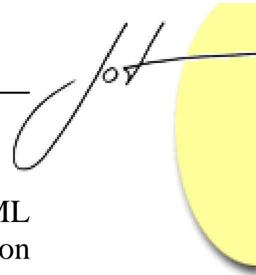
The implementation of the system is obtained by refining these behavioral models down to models that contain sufficient information for automated code generation. In the development environment deployed at Motorola, these models can then be compiled into executable code, usually C or C++ using an optimizing code generator [Weigert03]. The code generator injects platform-specific dependencies and adapts the model to particular deployment topologies. Code optimization destroys the structural and syntactical correspondence between the generated code and the model. Generated code is therefore not supposed to be manually inspected or refined. Applications developed in this fashion are therefore completely specified at the level of the models.

This type of MDE development environment is characterized in the literature as translation-oriented [Mellor02], where models are automatically translated into code, as opposed to round-trip environments. Round-trip modeling environments make a clear distinction between model and code and require manual refinement of generated code skeletons.

Translation-oriented MDE environments require a heavy investment in code generators that are specifically tuned for a particular domain. These environments are adapted to domains where large product families need to be maintained over long periods of time, as in the telecom infrastructure domain.

In this paper, we focus on the refinement from abstract architectural models of the system to more detailed, complete models of the system. Refinement from behavioral models includes functional refinement but also the composition of features, the integration of alternative uses cases, the handling of exceptions and faults and various optimizations.

This refinement process, including behavior testing, is typically the most problematic phase of the development lifecycle. In this paper, we identify three major issues that impede the automation of these development tasks: mismatch between problem structure and modeling language abstractions, difficulties in isolating the implementation of different use cases and features from each other, and difficulties in abstracting from application-specific low-level implementation details.



1. Model abstractions and decompositions: Modeling languages such as the UML provide complementary specification languages. Yet, these specification languages only support hierarchical and orthogonal decomposition mechanisms. Many system features and use cases do not map either of these decompositions completely, especially when they need to be integrated with low-level behavior of other use cases/features. This causes discrepancies between architectural models and detailed models.
2. Use case and feature interdependencies: The difficulty in cleanly modularizing different use cases and features from each other makes it hard to explicitly declare their interactions and dependencies. They become hard to define independently of their low-level behavior. The interactions between use cases/features become buried inside of the implementation instead of being declared explicitly.
3. Application-specific implementation concerns: Code generators can be tuned to systematically handle domain-specific implementation details and platform specificities. Yet, most systems also exhibit application-specific implementation details. Many of these concerns do not map cleanly to the main decomposition of the system and need to be integrated manually at multiple locations in the system.

Section 3 illustrates these problems through a classic example from the telecom domain. The formulation of those limitations suggests that they can be addressed by Aspect-Oriented Software Development (AOSD) technologies. Features that exhibit such characteristics have been categorized in the literature as crosscutting concerns [Kiczales97]. A concern is an area of interest or focus in a system, such as a requirement, a feature or a use case. Crosscutting concerns are concerns that are hard to modularize using the dominant decomposition mechanisms of the language used because they follow different decomposition rules.

Aspect-Oriented Software Development

Aspect-Oriented Software Development (AOSD) [Elrad04] is a development paradigm that focuses on the modular implementation of crosscutting concerns. Aspect-Oriented Programming (AOP) languages provide explicit language-level support for localizing crosscutting concerns into separated modules, called aspects. AOP languages use predicates over the system implementation, called *pointcuts*, to capture *joinpoints*, points in the system, such as method calls, where aspects inject behavior through *advice* bodies. Aspects encapsulate pointcuts, advice bodies, attributes and methods that pertain to the implementation of a crosscutting concern.

Yet, it is generally agreed on in the literature that aspects are hard to reason about in isolation [Kiczales05]. Even as crosscutting concerns are modularized, their integration with the system requires coordination among different developers. This problem has been characterized as the *Fragile Pointcut* problem [Gybels03][Ostermann05].

Pointcut descriptors introduce strong coupling between aspects and the modules they apply to. Aspects depend on specific points in the execution of the system to be exposed, according to particular signatures, such as method call signatures. Small refactorings are

therefore susceptible to modify the way an aspect interacts with a module, breaking the semantics of the aspect. Modules that are advised by aspects become hard to evolve independently. Developers need to be aware of the aspects defined in the system when modifying the implementation of a component. In large development environments, this situation might be worse than problems introduced by code tangling and scattering because it involves coordination between developers, as the behavior introduced by the aspect is not directly visible in the components.

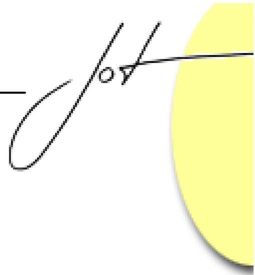
The fragile pointcut problem is partially caused by the lack of stable semantic abstractions for representing pointcuts. One of the goals of Aspect-Oriented Modeling (AOM) techniques is to provide ways to express pointcuts in terms of system behavioral specifications, rather than the system implementation. Such pointcuts are likely to be more robust with respect to refactorings.

In MDE, architectural behavioral models can be used as interfaces with respect to aspects. As long as the behavioral model of the component is maintained, changes in its implementation will not affect the correctness of the aspects applied to the component. This technique requires the ability to infer joinpoint locations in the implementation of a component from pointcuts that are expressed in terms of its behavior specification.

The Motorola WEAVR [Cottenier07] [Cottenier05] is an AOM engine developed at Motorola, as an add-in to the Telelogic TAU [TAU] modeling tool. It provides language constructs to capture aspects in UML 2.0 and performs weaving of state machines through model transformation before code generation. The WEAVR tool performs a particular type of joinpoint inference that targets the detection of decisions in the system. Decisions are conditional statements that have an important impact on the future behavior of the system. Decisions also tend to be locations where crosscutting occurs; they are locations where alternative use cases interact.

MDE and AOSD exhibit some complementary properties. Modeling enables systems to be specified at higher level of abstraction but suffers from difficulties with respect to the refinement and integration of system perspectives. On the other hand, aspect technologies focus on the modularization and composition of concerns, but lack appropriate abstraction mechanisms.

The limitations of MDE with respect to decomposition and separation of concerns will be illustrated by an example from the telecom domain. The next section presents this example and introduces the modeling concepts use in the paper, as well as the behavioral specification language used by WEAVR. Development issues that arise when extending the behavior of the system will be presented in Section 3. This section also introduces the WEAVR aspect language and proposes aspect solutions to the identified problems. Section 4 details the joinpoint model of WEAVR and discusses some of the more advanced features of the language, including realization mappings, decision inference and state introductions. Section 5 presents a discussion on the proposed approach and discusses further work. Section 6 discusses related work and Section 7 concludes this paper.



2 MODEL-DRIVEN ENGINEERING

Architecture and Behavioral Specification

It is desirable to validate the system design and architecture as early as possible in the development lifecycle. For large systems, validation is essentially performed through model simulation and testing. System requirements are mostly expressed using use cases, sequence diagrams and textual descriptions. Architectural models specify the logical components of the system, their observable behavior and their interactions. In the telecom domain, system architecture is usually modeled using composite-structure diagrams, class diagrams, and state machine diagrams. Typically, system simulation produces traces in the form of sequence diagrams that can be validated automatically with respect to the system requirements using automated testing.

Composite-structure diagrams are used to identify and model the different subcomponents of the system. These subcomponents become the basic units of independent development of the system, with respect to a use case. Composite-structure diagrams define a hierarchical run-time decomposition of the system. They define the internal run-time structure of an *active class* (a process definition), in terms of other active classes instances, referred to as *parts*. A *connector* specifies a medium that enables communication between parts of an active class or between the environment of an active class and one of its parts.

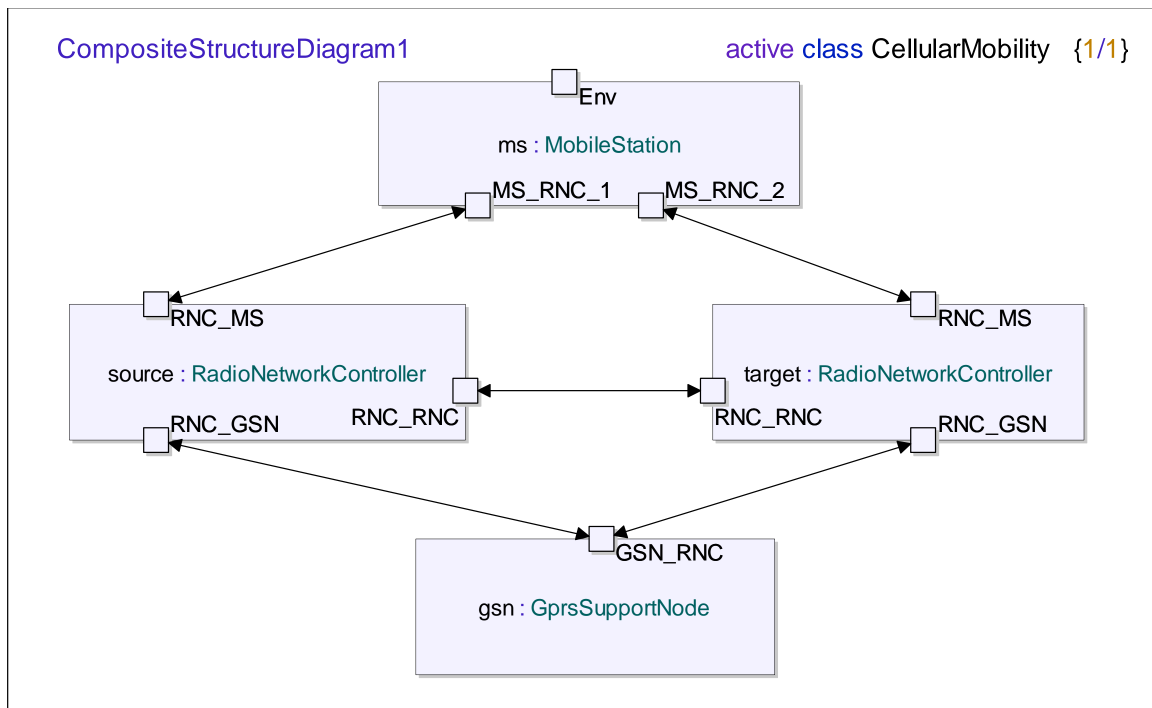


Figure 1. Composite-structure diagram for a classic cellular communication application

Figure 1 illustrates a simple composite-structure diagram for a classic mobile communication problem, the relocation of signaling communication channels between two cellular radio network controllers (RNC) managed by one gateway support node (GSN) and initiated by a mobile station (MS). The specific example presented in this paper is inspired from the UMTS handoff protocol [Pang04] [Lin01]. The handoff procedure is responsible for maintaining a session connection with the MS, while ensuring that no data is lost during handoff.

The diagram contains four parts: an MS, a GSN, a serving RNC which holds a connection from the GSN to the MS and a target RNC which is the RNC that takes over the connection after handoff. Figure 1 also shows the connectors between ports of the parts of the system. The RNCs and the GSN are connected through a private network, whereas the MS has wireless connections with the RNCs.

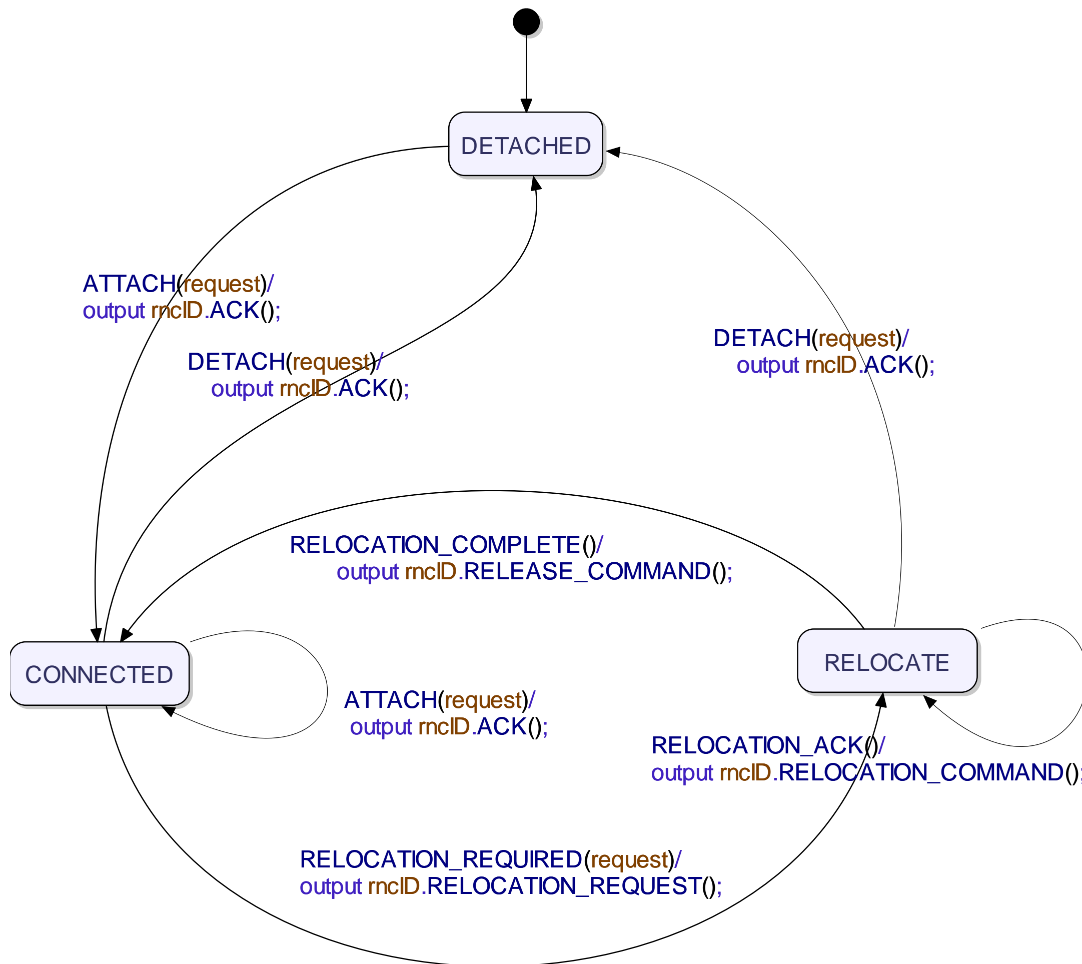
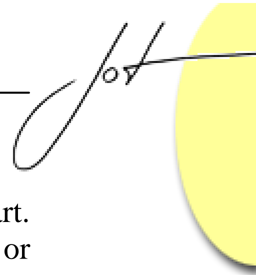


Figure 2. Behavioral specification of the connection handoff procedure from the perspective of the GSN

In this paper, a particular type of state machine-based behavioral specifications is used.



WEAVR system behavioral specifications define the observable behavior of each part. They specify what the state transitions of the system are, their triggers and what output or actions they may produce. The specification does not detail how these actions are produced or how the decisions regarding the execution of a specific path are made. These state machines are therefore not imperative and are not fully executable. They define possible execution paths of the system, usually paths that correspond to the main use cases.

Figure 2 illustrates a behavioral specification of a cellular handoff procedure, from the perspective of the GSN. It defines the different states of a connection, the events that trigger transitions between those states, and actions that *may* be executed along those transitions.

The states defined in Figure 2 correspond to the states of the connection between the network and the MS. When no link is established in the system, the connection is *DETACHED*. When a connection is established with an RNC, the connection becomes *CONNECTED*. During relocation, a new connection is established while the old connection is being revoked. The connection is then in the *RELOCATION* state.

The transition from the *DETACHED* state to the *CONNECTED* state corresponds to an MS network entry. The transition from *CONNECTED* to *CONNECTED* corresponds to a connection renewal. The transition from *CONNECTED* to *RELOCATE* corresponds to handoff initiation. Finally, the transition from *RELOCATE* to *CONNECTED* corresponds to handoff completion.

Figure 4 presents the corresponding behavioral specification of an RNC:

- Figure 4.a specifies the transitions that occur during a network entry. The MS issues a registration request by sending the *REGISTRATION* signal. When an RNC receives a registration request, it sends an *ATTACH* request to the GSN and waits for a response. Once the connection is established, the RNC *actively* transmits downlink packets from the GSN to the MS.
- Figure 4.b describes the behavior of a serving RNC during relocation. After handoff initiation, the serving RNC *forwards* packets from the GSN to the target RNC. Upon relocation completion, the serving RNC becomes *Idle*.
- Figure 4.c describes the behavior of the target RNC during relocation. During handoff initialization, the target RNC *buffers* packets received from the serving RNC. Upon relocation completion, the target RNC *actively* delivers packets to the MS.

The handoff procedure needs to guarantee that no packets get lost during relocation. The GSN therefore implements a handshaking protocol (*RELOCATION_REQUEST*, *RELOCATION_COMMAND*, *RELOCATION_COMMIT*) between the RNCs to ensure that the target RNC is ready to buffer signals before the serving RNC start forwarding packets. The detail of the handshaking protocol is not essential to this discussion but is presented for illustrative purposes.

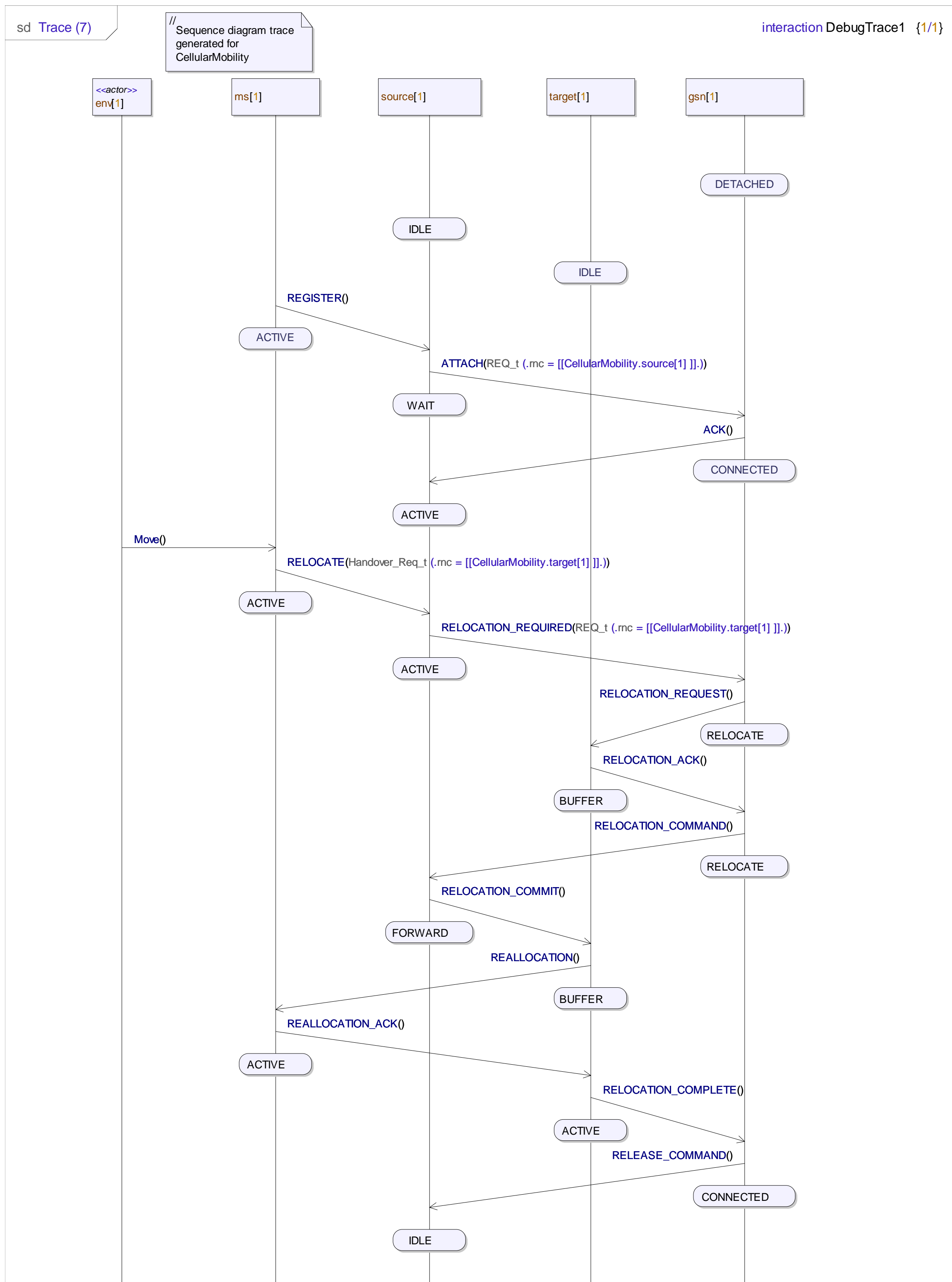
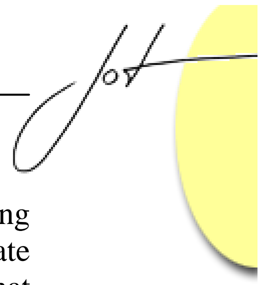


Figure 3. Trace generated by the model verifier during a simulation of the relocation behavioral specifications of Figure 2 and Figure 4



A transition defined in a WEAVR behavioral specification expresses the following behavior. From the transition start state, there exists a path to the transition target state along which the actions defined along the transition are executed. It is not guaranteed that these actions will be executed or that the transition will complete after it has been fired, as the system might follow a different path that is not defined in the specification. These specifications therefore exhibit non-determinism. A transition defined in the specification may follow an alternative path, which is not explicitly declared, bringing the system in a *FAILURE* state. These semantics allow specifications to simulate alternative uses cases early in the lifecycle. A complete formal treatment of WEAVR behavioral specifications is beyond the scope of this paper and will be subject for further publications.

While WEAVR behavioral specifications are not fully executable, they can be simulated. In its initial phases, the simulation is performed by stepping through the execution of the model, while manually taking decisions about the execution of the system (such as whether to execute connection renewal or relocation initiation or whether a particular action should be executed). During this phase the initial architecture is refined as to satisfy the requirement use-cases. Once the architecture converges, different execution paths can be encoded into tests to drive the simulation automatically and perform analysis.

The simulator generates a trace of the system as a sequence diagram. Figure 3 illustrates a trace generated by the simulator for a network entry followed by a successful connection handoff between a serving RNC and a target RNC.

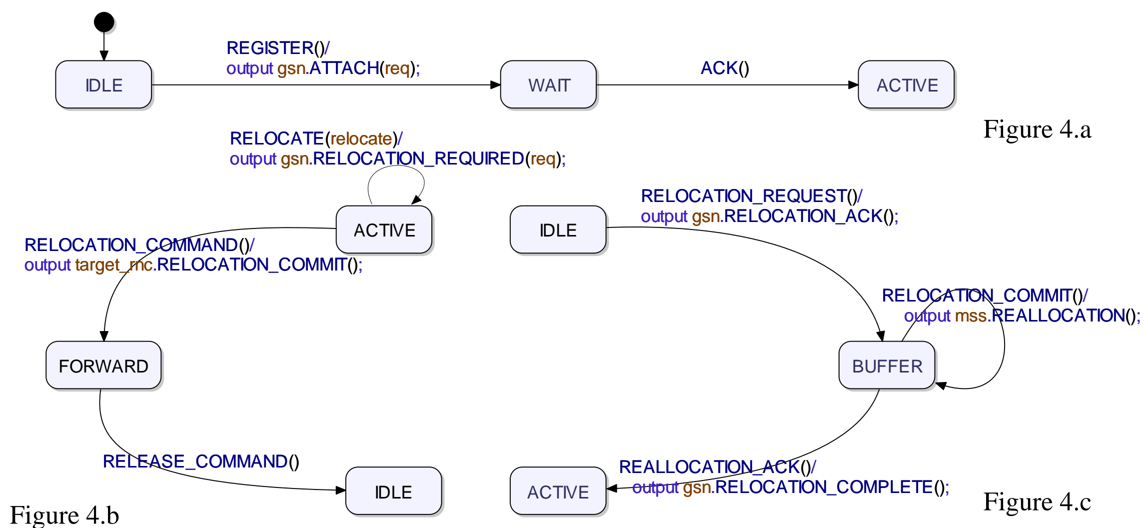


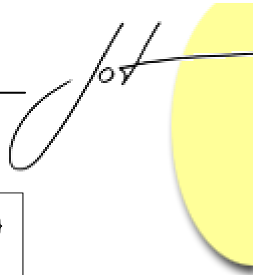
Figure 4. Behavioral specification of the connection handoff procedure from the perspective of the source RNC (Figure 4.b) and the target RNC (Figure 4.c)

System Refinement and Implementation

System implementation is defined using imperative *transition-oriented* state machines, which are characteristic of the Specification and Description Language (SDL) and are now part of the UML 2.0 standard. Transition-oriented state machines provide a better view of the control flow and the communication aspects of a transition than a state-oriented view. They are used for defining the detailed internal behavior of a reactive component. Transition-oriented state machines use explicit symbols for different actions that can be performed during the transition. They also make the control flow explicit using decision actions, represented as diamonds.

Figure 5 illustrates an implementation of a GSN *Connection* manager. The *Connection* manager implementation is a refinement of the behavioral specification of Figure 2. As opposed to Figure 2, this state machine is fully executable. It can non-ambiguously be compiled by a code generator, as it explicitly details the individual actions executed along transition, and how they are produced.

The *Connection* manager implementation also contains actions that are not represented at the level of the behavioral specification, such as the *setRecord* action. As this action might fail and throw an exception, a complete implementation would need to handle this exception and recover the system to a consistent state. Both the implementation presented in Figure 5 and an implementation that handles *setRecord* exceptions would conform to the WEAVR specification of Figure 2.



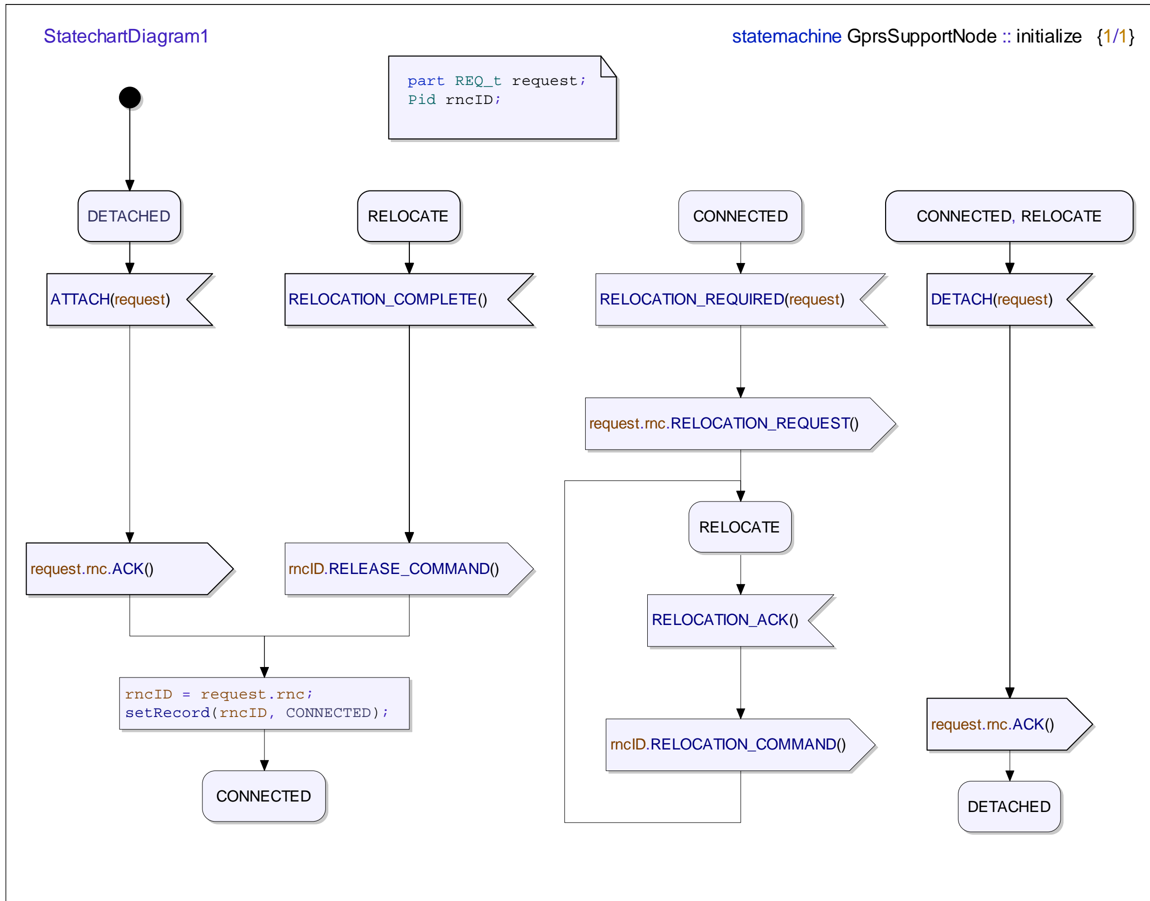


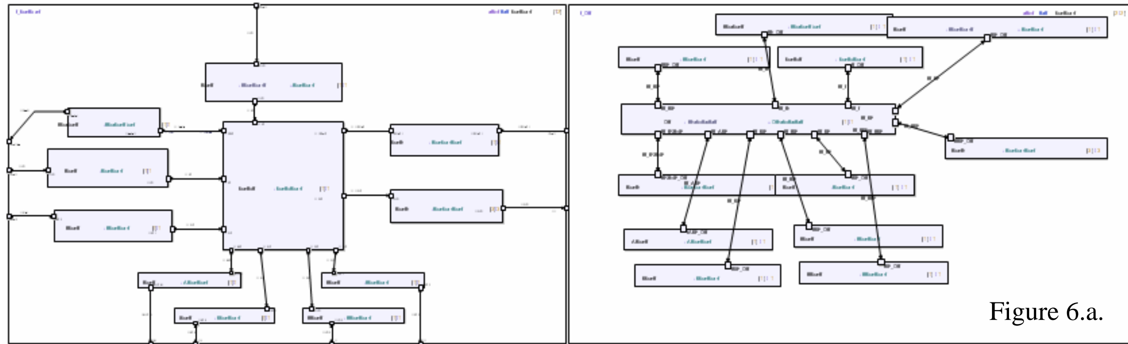
Figure 5. An example implementation of a GSN connection manager. The connection manager implements the specification of Figure 2

The refinement relationship between implementation models and the behavioral specifications used by WEAVR can be declared explicitly through WEAVR realization mappings, which are defined in Section 4.

3 ASPECT ORIENTATION AND MODEL-DRIVEN ENGINEERING

This section illustrates some of the shortcomings of the modeling language and the model-driven engineering process, especially regarding development tasks that involve the refinement from specification and architecture to detailed, complete, executable models. Aspect-oriented solutions to these problems are proposed, expressed using the WEAVR aspect-oriented modeling language.

Figure 5 gives a high level view of the composite-structure diagrams of two sub-components of a large communication system. Each of those sub-components is itself composed of multiple parts. This system is subject to a requirement that states that *"When the system starts up, all parts must initialize successfully, otherwise the system must shutdown"*.



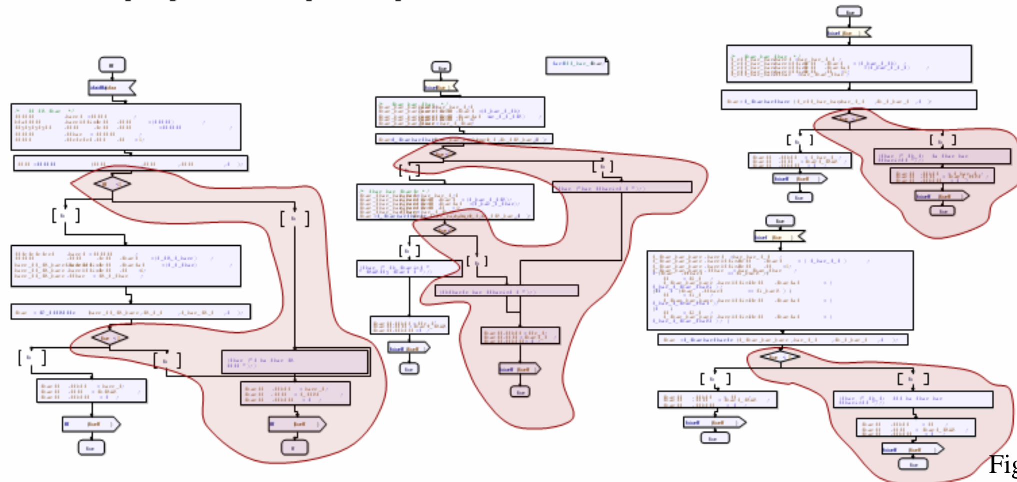


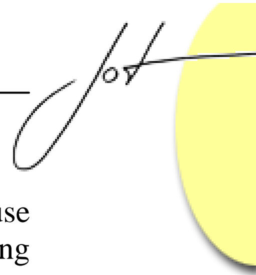
Figure 6. Two system sub-component composite structure diagrams and illustration of the impact of an initialization failure requirement upon the implementation state machines of its parts

This requirement captures a transactional all-or-nothing behavior. With respect to the system implementation, it implies that each part of the system must be able to detect a failure condition upon initialization, and notify a coordinator. It also requires each part to wait for the coordinator confirmation before entering its active state.

The execution state machines of Figure 6.b illustrate the impact of the initialization failure detection and notification requirement upon the implementation of some parts. As the system contains hundreds of parts, the impact of this requirement on the implementation is massive. The integration of the initialization coordination concern poses the following problems:

The implementation of the failure detection and notification concern cannot be modularized in a separate state machine because it interacts with the control flow of the state machine implementation of each part. The abstraction and composition mechanisms of the UML do not allow the concern to be specified and implemented in a separate module.

The interaction between the failure detection concern and the main behavior of the parts needs to be defined in terms of the low-level behavior of the main use case. The implementation of the main use case is tangled with the implementation of the



initialization failure concern. The interaction between main use case and exceptional use case becomes buried inside of the implementation state machines instead of being declared explicitly.

The implementation of the failure detection includes a logging statement that is executed upon detection of the failure condition (right after the detection decision action, represented as a diamond). The code generator can automatically introduce customized domain specific logging statements. Yet, in this case, logging is highly dependent on a particular use case. Application-specific implementation details can not be handled systematically by the code generator and need to be added manually in the models.

The following sections illustrate each of those issues through the introduction of an additional feature to the cellular system example presented in Section 2, and propose aspect-oriented solutions expressed in the WEAVR aspect-oriented modeling language. The basic WEAVR language features are introduced through those examples. The WEAVR joinpoint model and more advanced language features are detailed in Section 4.

Model Decomposition and Superimposition

The connection handoff process presented in Section 2 constitutes the basic relocation scenario. This procedure is very expensive in terms of communication overhead. Whenever a mobile station (MS) changes cell location, it scans for nearby base transceiver systems (RNC) and initiates a handoff. This procedure consumes bandwidth between base stations and base station controllers (GSN) and consumes MS battery power. In practice, full handoff is only required when the MS is actively communicating with the network.

In order to save bandwidth and MS battery power, a lightweight handoff mechanism can be implemented whenever the MS is operating in *standby* signaling mode. In standby mode, the MS become periodically available for downlink broadcast traffic without having to register at a specific RNC. Transmission is performed through a connection held by the GSN, which periodically pages the MS.

Figure 7 illustrates the behavioral specification of the signaling mode activation logic, within the GSN component. Whenever the MS releases the signaling connection, through the *CONNECTION_RELEASE* signal, the GSN prepares for handling lightweight handoff operations, by entering the *STANDBY* state. In *STANDBY*, the GSN takes full control over handoff operation when it receives an *ATTACH* or a *RELOCATION_COMPLETE* signal. Instead of entering the *CONNECTED* state, the GSN transitions to a new state, called the *IDLE* state in the *Connection* state machine. Full handoff operations are only resumed when the GSN receives an *ATTACH* signal, while the *Signaling Mode* state machine is in the *READY* state. The connection also can be released, when receiving a *DETACH* signal in the *STANDBY* state.

The standby mode handoff behavior can be expressed as illustrated in Figure 8. When the *Signaling Mode* state machine is in the *STANDBY* state, the *ATTACH* and *RELOCATION_COMPLETE* signals trigger transitions to a new state, called *IDLE* in the *Connection* state machine. In this state, all downlink packets are redirected to a

component that performs paging instead of being forwarded to the serving RNC. The *Signaling Mode* state machine therefore not only modifies the connection relocation behavior, but also the signal routing behavior.

Both the state machine of Figure 2 and the state machine of Figure 7 refer to behavior that needs to be implemented by the same instance of the GSN connection management component. The instance needs to respond to connection related signals in the *DETACHED*, *CONNECTED*, *RELOCATE* and *IDLE* states, but also need to respond to standby mode activation/deactivation requests in the *READY* and *STANDBY* states.

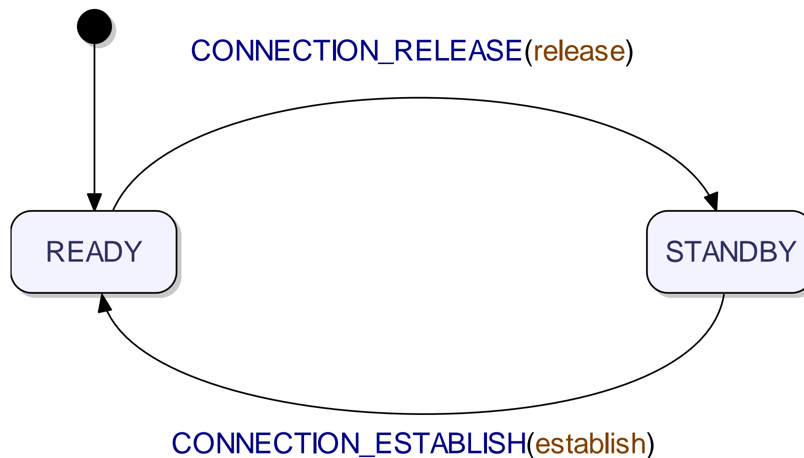


Figure 7. Behavioral specification of the *Signaling Mode* activation logic, at the GSN

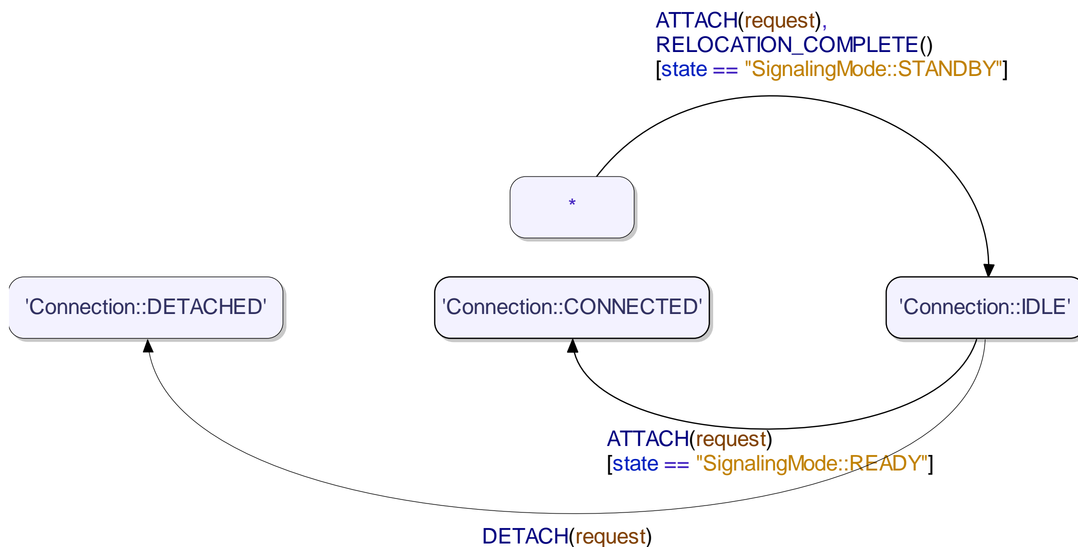
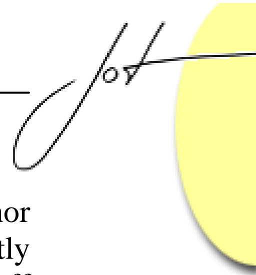


Figure 8. Behavioral specification of the relocation procedure at the GSN, when operating in *Standby Mode*



The state machines of Figure 2 and Figure 7 are neither completely hierarchical nor completely independent. In standby mode, handoff operations are performed differently than in ready mode. Yet, the system might enter the standby mode while handoff operations are still under way at the level of the RNC. Standby mode handoff operations therefore only start when the GSN receives a signal that causes a transition to the *CONNECTED* state, such as *ATTACH* or *RELOCATION_COMPLETE*. The state machine of Figure 2 is therefore not a sub state machine of the *READY* state, in the *Signaling Mode* state machine.

The state machines of Figure 2 and Figure 7 are clearly not independent either. They could be represented using Harel statechart orthogonal regions [Harel87] by directly introducing the standby handoff behavior of Figure 8 directly into the state machine of Figure 2. Yet, this has the drawback that the models that implement the base handoff behavior can not be maintained independently of the standby mode behavior. Also, it obscures the interaction between the base and standby mode handoff behavior. Use case interdependencies are further discussed in the next section.

In practice, few problems are independent enough to be concisely captured by the orthogonal region concept. This is one of the reasons why many UML modeling tools, such as Telelogic TAU, do not support this concept at all. Consequently, semi-independent state transitions can only be implemented by manually projecting one state machine onto the decomposition of the other one, which dramatically increases system complexity.

One solution is to complement the Harel orthogonal region mechanism by a superimposition [Bougé88] [Katz93] [Kurki-Suonio03] relationship between state machines. Figure 9 illustrates this concept.

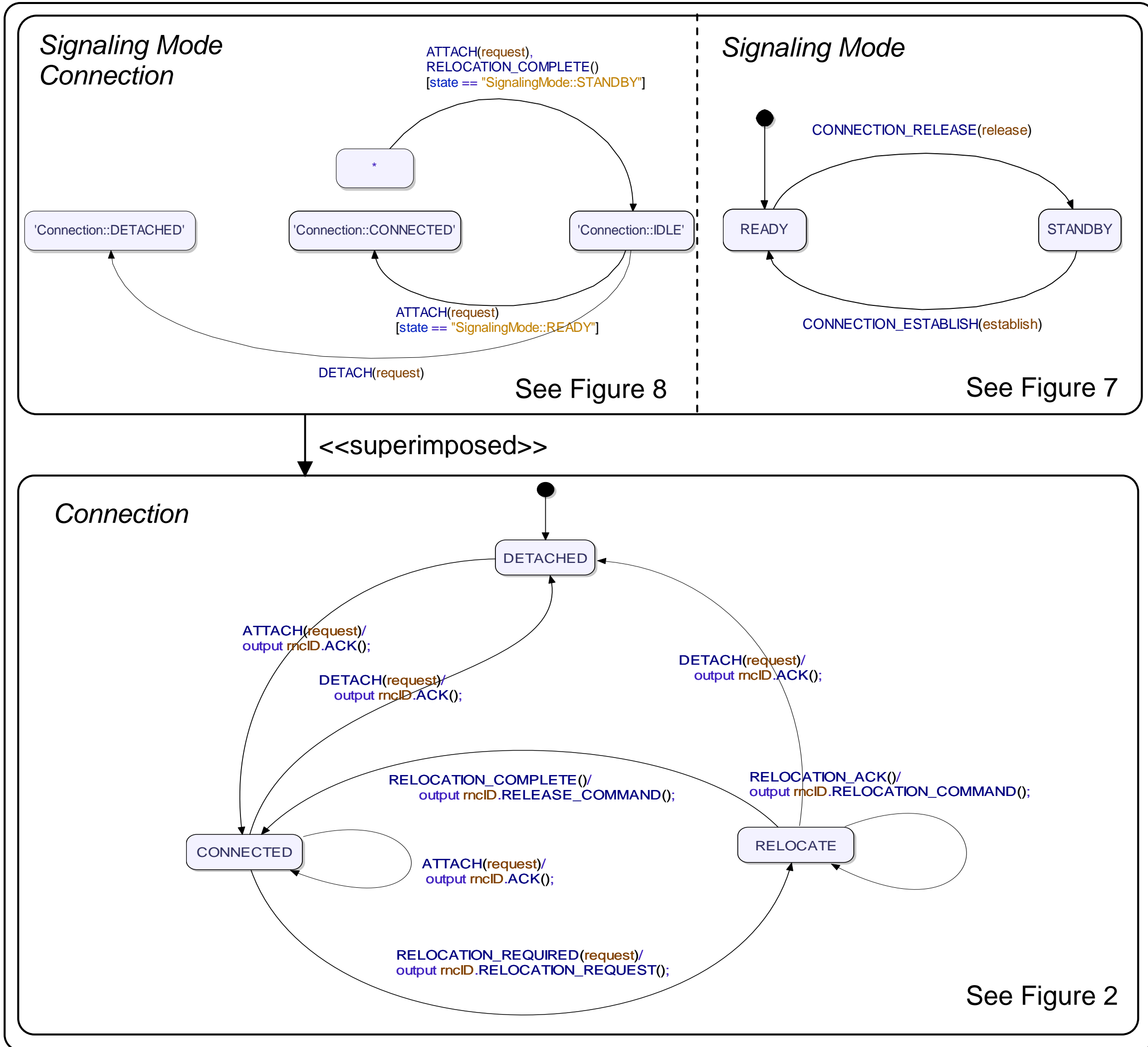
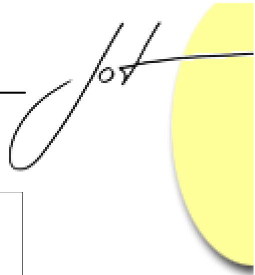


Figure 9. Superimposition of the standby *Signaling Mode* operations on the base handoff operations. The standby handoff operations and the signaling mode activation logic are declared in orthogonal regions

The *Signaling Mode* activation state machine of Figure 7 is maintained in a different orthogonal region than the standby mode handoff state machine of Figure 8.

The standby mode handoff state machine is superimposed on the base handoff state machine. In standby mode, the *ATTACH* and *RELOCATION_COMPLETE* signals are handled by the state machine of Figure 8. The result of the superimposition of state machines of Figure 8 and the state machine of Figure 2 becomes a co-region of the *Signaling Mode* activation state machine.

This solution avoids the pitfalls mentioned before, as the base handoff behavior is maintained separately from the standby mode handoff behavior. Furthermore, the <<superimposed>> dependency makes it clear that the signaling mode directly impacts the relocation behavior.



ClassDiagram1 <<Aspect>>class SignalingModeConnection {1/1}

<<operation,TransitionPointcut>>
StandbyConnect

<<binds>>

<<operation,Advice>>
SetIdleConnection

Figure 10.a

StatechartDiagram1 <<TransitionPointcut>> void StandbyConnect() {1/1}

*

*

'Connection::CONNECTED'

Figure 10.b

StatechartDiagram1 <<Advice>> void SetIdleConnection() {1/1}

proceed();

state == "SignalingMode::STANDBY"

else

==true

setRecord(rncID, IDLE);

'thisStateMachine<IDLE>'

'thisStateMachine<IDLE>'

ATTACH(attach) [state == "SignalingMode::READY"]

attach.rnc.ACK()

rncID = attach.rnc;
setRecord(rncID, CONNECTED);

'Connection::CONNECTED'

Figure 10.c

Figure 10. Aspect-oriented implementation of the standby mode handoff operations. The superimposition dependency is implemented by a transition pointcut

This solution can be implemented using the WEAVR aspect-oriented modeling language as illustrated by the aspect of Figure 10. The *SignalingModeConnection* aspect implements the specification of Figure 8. An aspect is represented as a class annotated by the *<<Aspect>>* stereotype. The aspect of Figure 10 contains one pointcut, the *StandbyConnect* transition pointcut, as indicated by the *<<TransitionPointcut>>* stereotype, and one advice, the *SetIdleConnection* advice, as indicated by the *<<Advice>>* stereotype. The *SetIdleConnection* advice is bound to the *StandbyConnect* pointcut by a *<<binds>>* dependency.

WEAVR supports two types of pointcuts: action pointcuts and transition pointcuts. While action pointcuts match actions executed along a transition, according to action signature, the current state and the transition it the context of which it executes, transition pointcuts match transitions or sections of transitions depending on their trigger signature, the current state, the next reachable state and the actions that are executed along the paths to next reachable states. The semantics of action and transition joinpoint selection will be detailed in Section 4.

The transition pointcut illustrated in 10.b matches all transition from any state, to the *CONNECTED* state, triggered by any signal in the *Connection* state machine. The *SetIdleConnection* advice introduces one new state, the *IDLE* state in the *Connection* state machine, as indicated by *'thisStateMachine<IDLE>'*. State and label introductions will be clarified in Section 4. The advice also introduces two transitions. The first one brings the connection state machine in the *IDLE* state, whenever an *ATTACH* or a *RELOCATION_COMPLETE* signal is received, while the *Signaling Mode* state machine is in the *STANDBY* state. This transition initiates the standby mode handoff behavior. If the *Signaling Mode* state machine is not in state *STANDBY*, the transition *proceeds* to the *CONNECTED* state, as indicated by the *proceed* keyword.

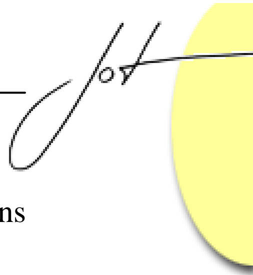
The second transition introduced by the advice brings the *Connection* state machine back to the *CONNECTED* state when an *ATTACH* signal is received in the *IDLE* state, while the *Signaling* state machine is in the *READY* state.

Model Interactions and Dependencies

The previous section shows that aspect-oriented modeling techniques can be used to alleviate problems caused by the lack of appropriate modeling language mechanisms for decomposing and isolating the implementation of different uses cases and features. The ability to isolate those use cases/features has two main advantages.

First, it allows the base behavior of different use cases or features to be reused in different contexts. Most alternatives to the decomposition presented in the previous section require combining the handoff state machines of the standby and ready signaling modes into a single state machine. These solutions over-specialize the base relocation state machine, impeding its reuse in contexts where other modes of operations are deployed. Another alternative would be to use state machine inheritance and transition redefinition. Yet, this solution tightly binds the standby mode behavior to the base handoff behavior as the transition that is refined needs to be completely re-implemented in the sub state machine. Furthermore, it requires intimate knowledge of the implementation of the base state machine.

The second advantage of the solution presented is that it makes the dependencies and interactions between different use cases, features or modes of operation explicit. These interactions and dependencies are declared in the pointcut expressions of the aspects. Pointcuts enable the behavioral specifications to act as interfaces between different use-cases/features. System decompositions that combine the implementations of the different



state machines into a single state machine obscure the dependencies and interactions between different concerns at the level of the implementation.

Figure 11 shows the structure of the GSN active class, in terms of three independent base parts, the *Connection*, *SignalingRelay* and *SignalingMode* state machines. The *Connection* state machine handles the base handoff behavior, and is specified by the behavioral specification of Figure 2. The *SignalingRelay* state machine is responsible for forwarding signaling packets to the serving RNC. The *SignalingMode* state machine defines the signaling mode activation behavior specified in Figure 7.

The *Mode* state machine is contained in an orthogonal region, as a co-region of two distinct aspects, the *SignalingModeConnection* aspect illustrated in Figure 10, and the *SignalingModeRelay* aspect. The *SignalingModeConnection* aspect extends the behavior of the *Connection* state machine for standby mode handoff operations. This interaction is defined explicitly by the <<crosscuts>> dependency from the *SignalingModeConnection* aspect to the *Connection* state machine. Similarly, the *SignalingModeRelay* aspect extends the base *SignalingRelay* state machine for standby mode signal buffering and paging. The impact of the standby mode operations on the *Connection* and *SignalingRelay* state machines is explicitly defined in the model, and characterized by the aspect pointcuts.

As mentioned before, the orthogonal region concept is not directly supported by Telelogic TAU. Typically, completely independent state machines are defined in different active classes. Orthogonal regions as illustrated in Figure 11 need to be simulated. In practice, parts that correspond to orthogonal regions of a same state machine are annotated with a co-region stereotype. These state machines can then be merged into an equivalent single state machine before code generation.

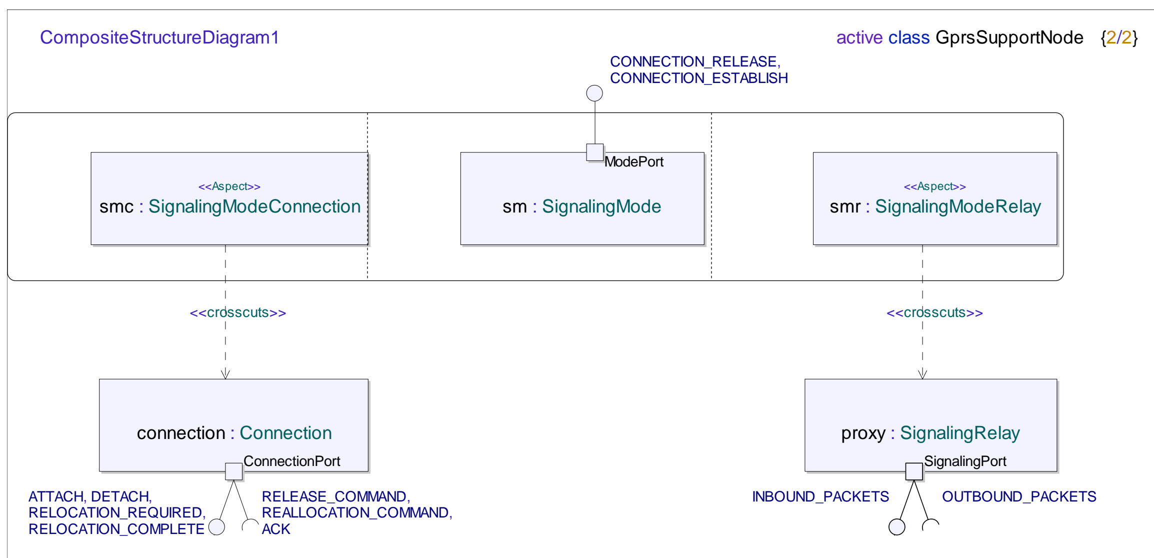


Figure 11. Decomposition of the GSN handoff and standby mode operations into three base state machines and two aspects. The standby mode operations crosscut both the connection operations and the signaling operations.

It is important to note that aspect pointcuts are fully expressed in terms of the state machine specifications rather than their implementation [Cottenier07]. The aspect state machine of Figure 8 only depends on entities that are explicitly declared in the architectural behavior specification of Figure 2, rather than the implementation described in Figure 5. WEAVR explicitly forbids aspects to be defined in terms of actions or transitions that are not declared in behavioral specifications. This restriction is fundamental to avoid strong coupling between aspects and particular implementations. Coupling between aspect and base state machines is further discussed in Section 4.

Application-Specific, Implementation-level Crosscutting Concerns

The previous sections discussed the use of aspect-oriented modeling technique to overcome some of the limitations of the decompositions provided by the modeling language. These techniques allow the implementation of different features or use-cases to be isolated from each other, while explicitly declaring their dependencies and interactions in terms of specification entities. This section discusses the use of aspects to modularize implementation-level concerns such as logging, exception handling or timing constraints.

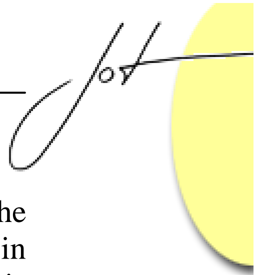
Code generation can automatically integrate crosscutting concerns such as tracing or recurring platform specificities with the base system. These concerns are activated and deployed through the configuration of the code generator. Yet, there are concerns, such as exception handling or timing constraints that are highly dependent on the application logic and cannot be handled in a systematic way through code generation. The implementation of these concerns depends on the application. These concerns could be handled through code generation by defining application-specific transformation rules that inject the required behavior automatically, directly into the intermediary code representation.

Yet, the process of extending the code generator requires expertise in automated code generation. The efforts spent in deploying application-specific transformation rules often overweighs the effort required to manually introduce the implementation of crosscutting behavior in the models.

Aspect-oriented languages provide developers with a simpler interface that can be used to define the implementation of crosscutting concerns directly in the model, rather than in code generator configuration files.

Figure 12 illustrates a connection timeout aspect. The *ConnectionTimeOut* aspect imposes a timing constraint on the duration of connection held by the GSN. This aspect is deployed directly in the *Connection* state machine of Figure 5. The *SetConnectionTimer* advice is responsible for setting a timer whenever a connection is initialized or renewed in the GSN. It also introduces a transition that terminates the connection instance upon expiration of the timer. The *ResetConnectionTimer* advice resets the connection timer whenever a connection is terminated.

The pointcut of Figure 12.b explicitly declares the DETACHED state. This declaration is required because the advice of Figure 12.c refers to this state. States that are not defined at the level of the pointcut can not be referred to directly by the advice.



The pointcut of Figure 12.d also illustrates the use of joinpoint inference. The ResetConnectionTimer advice introduces a timer reset statement at the first locations in the state machine for which the DETACHED state is the only reachable state. This is performed by placing the advice body before the proceed keyword, for an advice bound to a transition pointcut. These locations are either transition starting points, or points that immediately follow a decision action. Joinpoint inference is discussed in Section 4.

ClassDiagram1
active <<Aspect>>class ConnectionTimeOut {1/1}
<<operation,TransitionPointcut>>
Connect
<<binds>>
<<operation,Advice>>
SetConnectionTimer
<<operation,TransitionPointcut>>
Disconnect
<<binds>>
<<operation,Advice>>
ResetConnectionTimer
Figure 12.a

StatechartDiagram1
<<TransitionPointcut>> void Connect() {1/1}
DETACHED
*
*
CONNECTED
Figure 12.b

StatechartDiagram1
<<Advice>> void SetConnectionTimer() {1/1}
```
proceed();
set (CONNECTION_TIMER(), now + CONNECTION_DURATION);
```
CONNECTED
CONNECTION_TIMER()
rncID.DISCONNECT()
DETACHED
Figure 12.c

StatechartDiagram1
<<TransitionPointcut>> void {1/1}
Disconnect()
*
*
DETACHED

<<Advice>> void {1/1}
ResetConnectionTimer()
```
reset(CONNECTION_TIMER());
proceed();
```

Figure 12. A connection timeout aspect. The aspect ensures that a connection expires after a period of time, if it is not explicitly renewed.

Figure 12.d

Figure 12.e

The transition joinpoint selection semantics allow the *ResetConnectionTimer* pointcut to insert the timer reset advice as early as possible in sections of the transitions for which it is certain that only the *DETACHED* state is reachable. The semantics of transition joinpoint matching are further detailed in Section 4.

The ConnectionTimeOut aspect is representative of some of the most commonly used aspects in the modeling environment. These aspects allow repetitive implementation tasks to be systematically handled, while crosscutting behavior is cleanly encapsulated in a separate module. The aspect pointcuts precisely capture the conditions upon which timers should be set or reset, in terms of the behavior specification of the *Connection* state machine.

The behavior captured by the *ConnectionTimeOut* aspect is generic enough to be deployed as a canned aspect. Developers do not need to be familiar with the implementation of the advice bodies. They can directly reuse the aspect in multiple state machines, by redefining its pointcuts or by declaring state machine realization mappings, as detailed in Section 4. Examples of reusable aspects are aspects that implement a blocking pair of asynchronous messages, logging and tracing aspects, as well as part instantiation and management aspects.

4 MOTOROLA WEAVR: WEAVING ASPECTS INTO MODELS

The previous section introduced some of the WEAVR basic aspect-oriented modeling language constructs, and briefly mentioned some of the more advanced features:

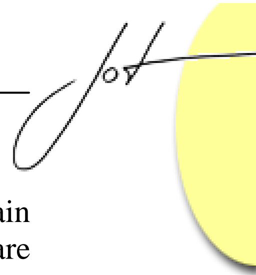
- State machine realization mappings.
- Abstract aspects.
- Transition joinpoint selection and decision action inference.
- State, transition and label introductions.

These concepts are important to maintain low coupling between aspects and the implementation of state machines they advise, without compromising on the expressiveness of aspects. WEAVR explicitly restricts pointcuts entities to be defined in terms of the architectural behavior specifications rather than their implementation. Yet, the weaving is performed at the level of the system implementation. The WEAVR therefore needs to infer implementation-level joinpoints from specification level pointcuts. This section details the WEAVR joinpoint model and its pointcut descriptors.

WEAVR Joinpoint Model and Pointcut Descriptors

As mentioned before, WEAVR supports two distinct types of pointcut descriptors: action pointcuts and transition pointcuts. The two types of pointcuts are distinguished by a different stereotype.

A pointcut is always represented as a transition from a set of source states to a set of target states, triggered by a trigger expression. The transition can contain one action expression. Wildcards can be used to quantify over both the source and target states of



the transition. As opposed to standard UML transitions, pointcut descriptors may contain a multistate (a set of states) as the target of a next state action. The trigger expressions are used to match the signatures of transition triggers and action expressions are used to match the signatures of actions executed in the control flow of a transition.

source states ---------trigger expression/action expression----------> target states

The use of static analysis during joinpoint selection is central in WEAVR. As for other aspect-oriented languages, WEAVR supports the notion of *cflow* between pointcut descriptors. WEAVR also supports a generalization of cflow, applied to transitions. An action pointcut always declares an action expression in the cflow of a source state and a transition trigger (which might be wildcards). The actions matched by WEAVR will only include actions that are executed while the source state is on the stack (the state machine, or one of its parent state machines is in a state that matches the pointcut source state), while a transition triggered by a trigger that matches the pointcut trigger expression is currently executing, either directly in the state machine or in one of its parent.

With respect to transition pointcuts, WEAVR performs action and state reachability analysis. Transition joinpoints are sections of transitions that are in the cflow of states that match the pointcut source states and that are in the cflow of transitions that match the pointcut trigger expression. The matched transition joinpoint starts at the first location for which it is statically certain that an action that matches the pointcut action expression will be executed, and for which the only reachable states match the target states of the pointcut. As illustrated in the previous section, state expressions can include scope qualifiers to distinguish between the states of different state machines.

A formal definition of the pointcut selection mechanism would be best defined in terms of temporal logic, but is beyond the scope of this paper. Yet, the selection mechanism is illustrated in the next sections.

Actions joinpoints in WEAVR include:

- call expression actions (a call to a method)
- output actions (sending a signal)
- create expression actions (a call to a constructor)
- timer set/reset actions

Transitions joinpoints in WEAVR include:

- start transitions (the initialization of a state machine implementation)
- operation bodies (the execution of a method)
- triggered transitions (transitions triggered by a signal, a timer or a guard)
- decision answer transitions (a section of a triggered transition guarded by a decision action)

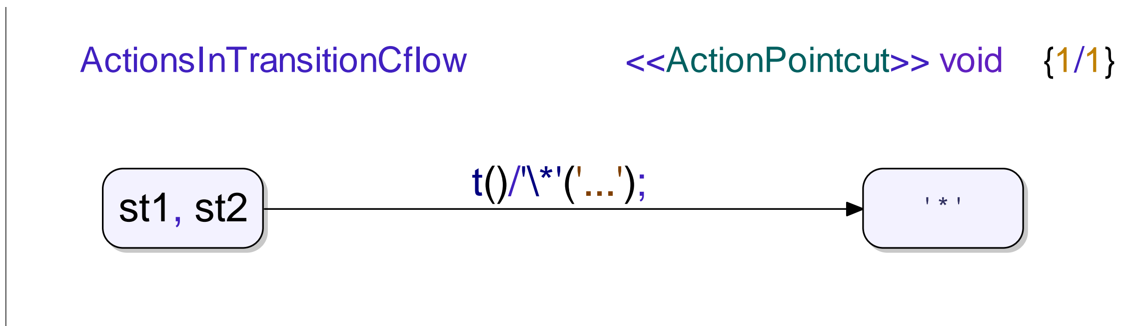


Figure 13. An action pointcut that matches all actions in the cflow of a transition triggered by t, from either st1 or st2

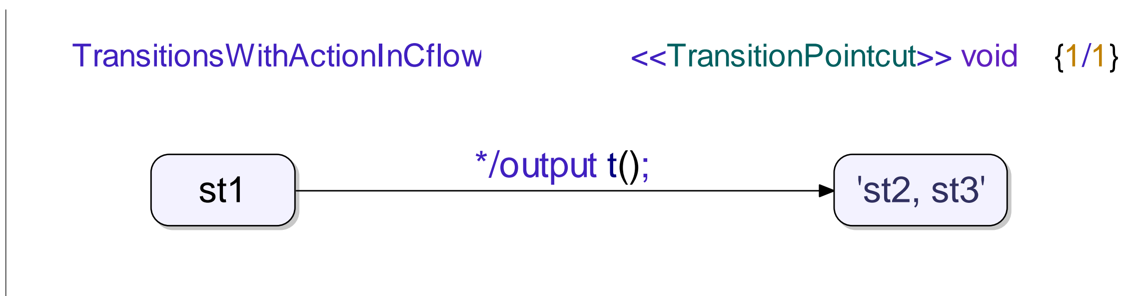


Figure 14. A transition pointcut that matches all sections of a transition from a state st1 to either st2 or st3, and, for which the signal t is send.

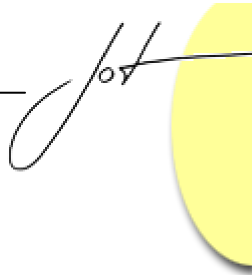
Figure 13 represents an action pointcut that matches all actions executed in the control flow of a state *st1* or *st2*, while a transition triggered by *t* is executing. Figure 14 defines a transition pointcut that matches all transition sections that execute in the control flow of a state st1, and that start from the first location for which only either state st2 or state st3 are reachable through paths that output the signal *t*.

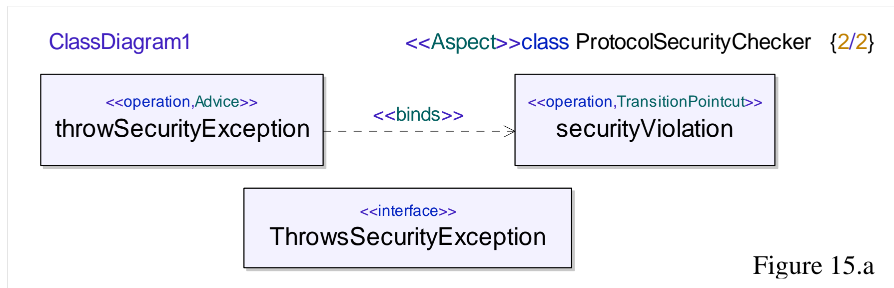
The next section introduces state machine realization mappings and illustrates state reachability analysis performed by WEAVR through an example.

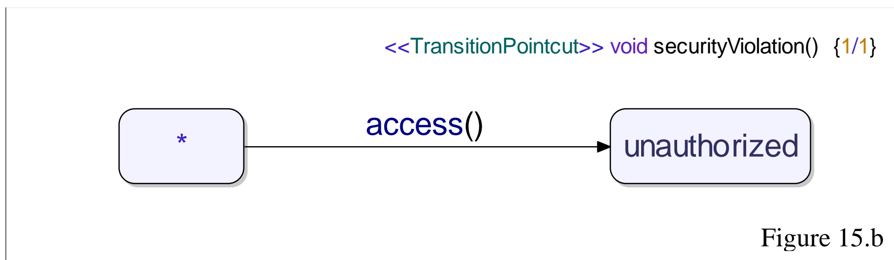
Abstract Aspects

As mentioned before, many aspects contain advice bodies that can be reused in different contexts. The deployment of reusable aspects requires redefining its pointcuts, or mapping the behavioral specification of the target state machine to a specification that matches the aspect pointcuts.

Figure 15 illustrates an aspect that throws a security exception, aborts the execution of the current transition and brings the state machine in a *FAILURE* state, whenever a transition triggered by an *access* trigger leads to an *unauthorized* state. It replaces transitions to unauthorized states by transitions to a new failure state.







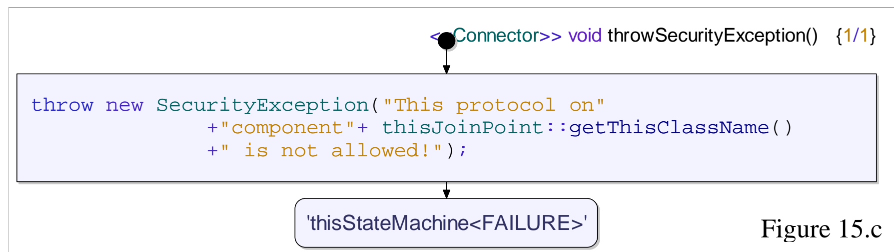


Figure 15. A WEAVR aspect that implements an access control concern

The aspect of Figure 15 is defined independently from base modules, and is meant to be reused for different applications. Its pointcut is therefore abstract as it not written with respect to a concrete behavioral specification.

The advice of Figure 15.c is instantiated before the transition from some state to an unauthorized state, at the first point for which only the unauthorized state is reachable. These transitions are aborted by the advice, as it does not invoke the *proceed* keyword.

The advice invokes the reflective API of the WEAVR through the *thisJoinPoint*, *thisTransition* and *thisStateMachine* keywords. These keywords are used to retrieve information about the context of a particular joinpoint. The advice of Figure 15.c. also introduces a new state, whose scope is the state machine of the joinpoint, as indicated by the *thisStateMachine* keyword.

WEAVR Realization Mappings

In order to deploy the abstract aspect of Figure 15, we need to map the specification of the system to a *perspective* that defines resource accesses and unauthorized states. This mapping is performed using realization relationships between state machines. WEAVR realization mappings map transitions from a concrete state machine to transitions of a more abstract state machine.

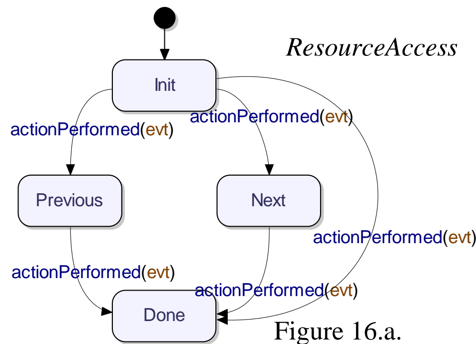


Figure 16.a.

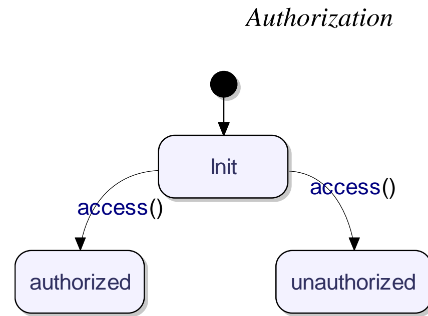


Figure 16.b

```
map (ResourceAccess): Authorization {
  {Init} actionPerformed {Previous}
                                   realizes {Init} access {unauthorized}
  {Init} actionPerformed {Next, Done}
                                   realizes {Init} access {authorized}
}
```

Figure 16. A realization mapping that maps the state machine of Figure 16.a onto the more abstract state machine of Figure 16.b

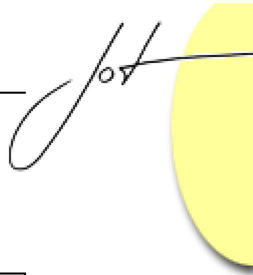
Realization mappings can be used during general development to declare the intent to conform to a behavioral specification. A realization mapping defines a refinement relationship: a concrete implementation state machine is declared to be the refinement of a more abstract state machine.

The problem of verifying that this relationship is indeed a refinement involves bi-simulation between state machines, and is beyond the scope of this paper. As the correspondence between transitions is explicitly declared in the mapping, an engine can easily detect simple conflicts and enforce that a realization mapping is maintained during development.

For example, the *Connection* state machine of Figure 5 is an implementation of the relocation specification of Figure 2. The state machine of Figure 5 is more specialized. Whereas the relocation specification exhibits non determinism, the state machine of Figure 5 imperatively details the control flow of the state machine and the exact actions that are executed along its transitions.

Figure 16 illustrates state machine realization by mapping the *ResourceAccess* state machine of Figure 16.a to the more abstract *Authorization* state machine of Figure 16.b. The mapping declares *realization* relationships from transitions of the *ResourceAccess* state machine to transitions of the *Authorization* state machine. The mapping of Figure 16 defines the *ResourceAccess* transition from *Init* to *Previous* as being an *unauthorized* transition, whereas the transitions from *Init* to *Done* and *Init* to *Next* are defined as *authorized*.

A realization mapping defines a particular view of the system that is of interest from the perspective of a particular concern, in this case authorization.



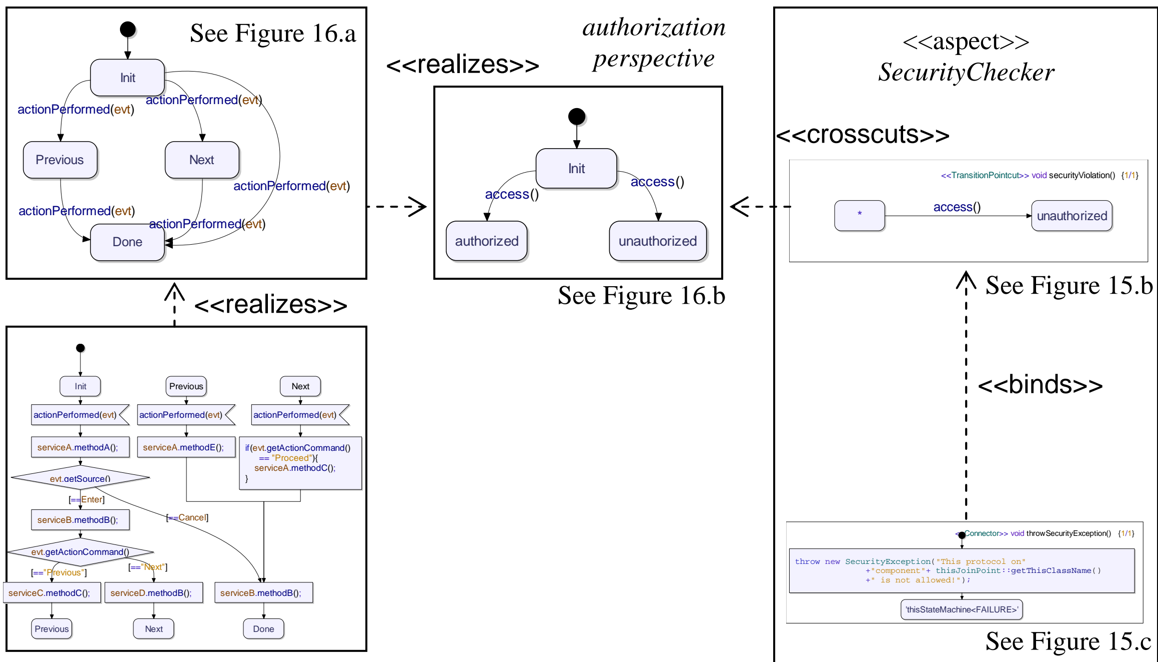


Figure 17. Representation of the deployment of the abstract SecurityChecker aspect on the implementation of the specification of Figure 16.a, through the perspective of the realization mapping of Figure 16

Once the authorization perspective is defined by the mapping of Figure 16, the abstract aspect of Figure 15 can be deployed on a concrete system as illustrated in Figure 17. The pointcut of Figure 15.b matches transitions of the specification of Figure 16.a through the authorization perspective defined by the realization mapping. Whereas the pointcuts are expressed in terms of the behavioral specifications, the joinpoints that are selected by the pointcuts are located at the level of the implementation, in the state machine represented at the left bottom of Figure 17.

Transition Joinpoint Selection

Figure 18 represents an implementation-level state machine that realizes the behavioral specification of Figure 16.a. The pointcut of Figure 15.b matches a section of one of its transitions. The transition joinpoint is delimited in the figure by two small shadowed boxes. The pointcut descriptor selects the portions of the transition for which the only reachable state is an *unauthorized* state (the *Previous* state). This location corresponds to the starting point of a *Decision Answer Transition*, right after a decision is performed. This matching method is powerful because it can localize the important decision points in the execution of a state machine.

The *before* location on a transition section always corresponds to a branch of a decision action. In the case of Figure 18, the decision selected is the *getActionCommand()*

== *"Previous"* expression. The decision itself is a point in the implementation, but it can be characterized in terms of the module specification.

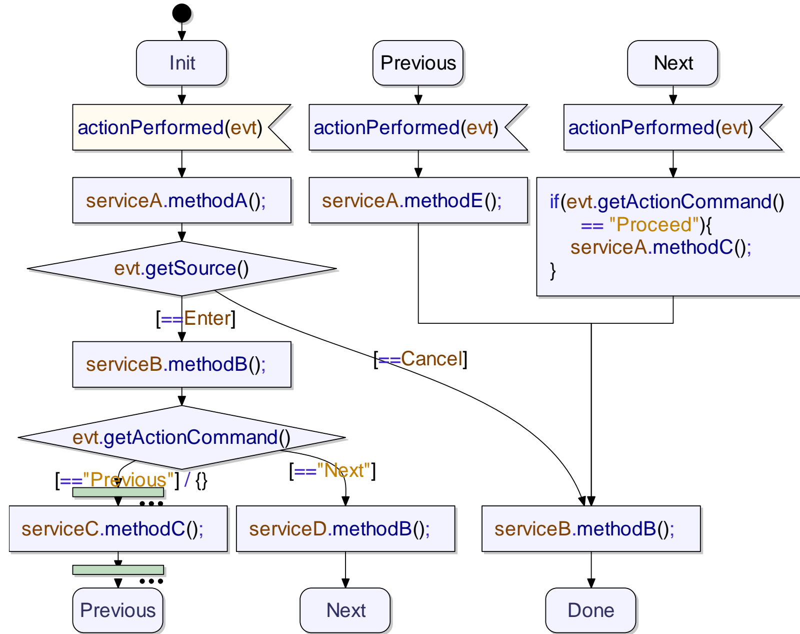


Figure 18. Section of a transition selected by the pointcut descriptor of Figure 15.b, through the mapping of Figure 16

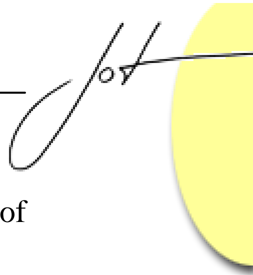
The transition selection mechanism enables the WEAVR to capture decisions in the implementation of modules, in terms of their behavioral specifications.

Realization mappings also enable pointcuts to be flexible with respect to modification of the views of a system. Suppose that the application requires both the *Previous* and *Next* states to become inaccessible in a certain mode of operation. This modification can be implemented by modifying the state machine specification realization mapping as follows.

```
{Init} actionPerformed {Previous, Next}
                       realizes {Init} access {unauthorized}
{Init} actionPerformed {Done}
                       realizes {Init} access {authorized}
```

Figure 19 shows the corresponding transition sections in the state machine implementation. The decision action matched is now the *getSource()* == *"Enter"* expression instead of the *getActionCommand()* == *"Previous"* expression. This is the first location in the state machine for which only unauthorized states are reachable.

The technique presented illustrates the concept of joinpoint *inference*. The exact locations of the joinpoints (the decision actions) are not specified explicitly in the pointcut expressions. The decision actions could have been located directly, by defining action pointcuts that match the *getActionCommand()* or the *getSource()* method calls. Yet, such pointcuts require intimate knowledge about the implementation of a module.



Joinpoint inference allows the same locations to be derived automatically in terms of the states of the system, and its state transitions.

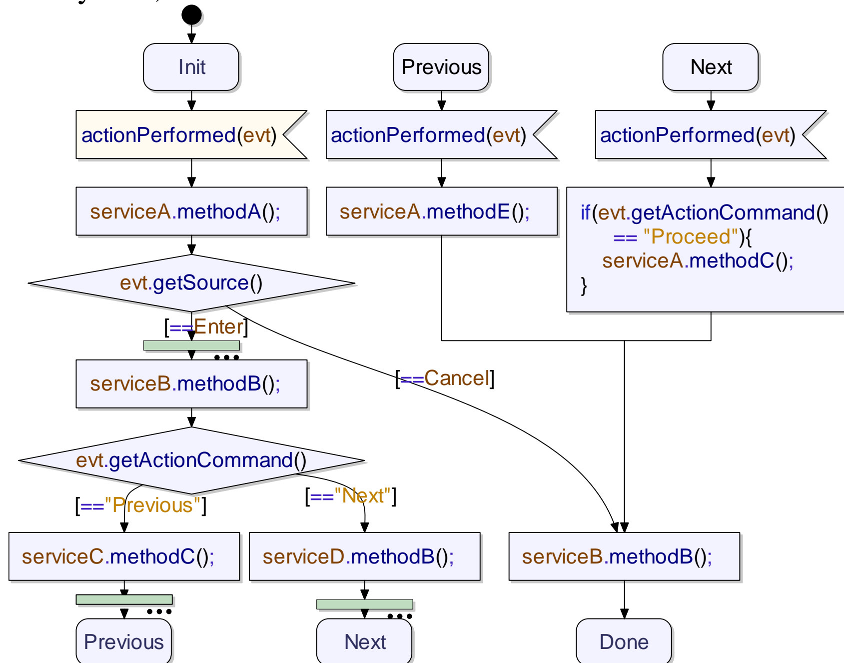


Figure 19. Section of the transition selected by the pointcut descriptor of Figure 15.b after modification of the mapping

Joinpoint inference allows pointcut to match points in the implementation of a module in terms of the module specification, without requiring potential joinpoints to be explicitly declared. This concept is key in addressing the fragile pointcut problem. Pointcuts are expressed in terms of a module specification, but still hold the power to match locations that are not explicitly exposed in the interface of a module.

State and Label Introductions

As mentioned before, aspects can introduce new states and new transitions in a state machine. State introduction requires the scope of the introduction to be defined. Introduction defined in an advice can be introduced per joinpoint state machine, per joinpoint transition or per joinpoint action.

These different levels of introduction granularity make it possible to introduce complex control flow structures in the system. Advices can not only introduce states, but also labels and join actions. A state or label introduction occurs when an advice refers to a state or a label that is not declared in its pointcut descriptors. When introducing states and labels, the scope of the introduction is specified through the *thisJoinPoint*, *thisTransition* and *thisStateMachine* keywords.

A state that is introduced per Joinpoint, *thisJoinPoint<state_name>*, is unique to the advice instance bound to a specific joinpoint. A state that is introduced per transition, *thisTransition<state_name>* is shared by all instances of the advice bound to joinpoints

of the same transition. A *thisStateMachine<state_name>* state is shared by all instances of the advice bound to joinpoints of the same state machine.

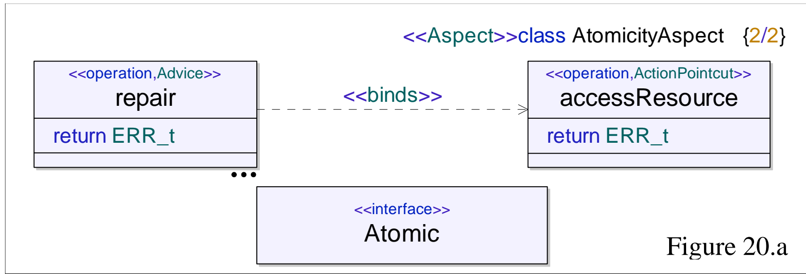


Figure 20.a

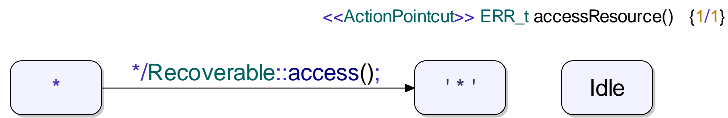


Figure 20.b

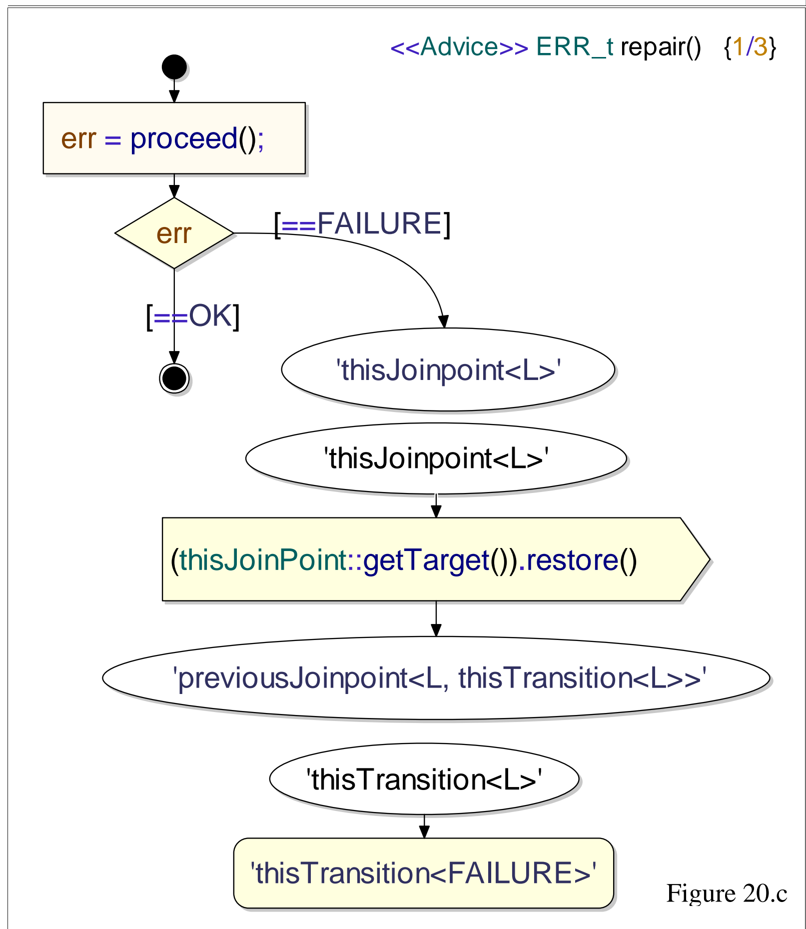
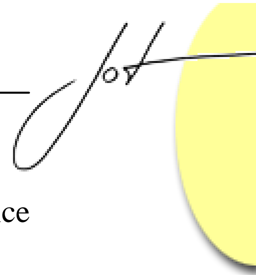


Figure 20.c

Figure 20. An aspect that makes all accesses to recoverable resources atomic along a transition

State and label introductions make it possible to define behavior that spans across different instantiations of an advice. The action joinpoints within a transition are partially



ordered. An advice can therefore reference states and labels introduced by other advice instances using the *previousJoinPoint <state_name, default_state>* notation.

The *previousJoinPoint* state resolves to a default state in the case the previous joinpoint is not defined.

Figure 20 illustrates a transaction atomicity aspect that makes use of label and state introductions. The pointcut descriptor of Figure 20.b. matches invocations of methods that implement the *Recoverable* interface and return an error code of type *ERR_t*. Resource access failure is detected through the return value of the invocation.

Whenever a resource access fails, the aspect advice of Figure 20.c. ensures that all the recoverable resources accessed along this transition are restored to their previous state. Resource recovery is performed by invoking the *restore* method on the resource. The advice builds a control structure that rolls back resources previously accessed along the transition, by referring to the label introduced by the advice at the previous joinpoint along the transition. The advice also introduces a state that characterizes the failure of the transition, *thisTransition<FAILURE>*. The default failure state becomes the target of the transition whenever a resource access failure is detected.

Figure 21 illustrates a state machine that is equivalent to the state machine obtained by weaving the aspect of Figure 20 into the *Init* transition of the state machine of Figure 18. The first joinpoint encountered along the transition is a call to the *methodA* method. This joinpoint does not have a previous joinpoint along the transition. The *previousJoinpoint* label therefore resolves to the default label for this transition, *thisTransition<L>*. The *thisJoinpoint<L>* label introduced by the advice resolves to a generated label, named *L_Init_1*. Whenever *methodA* fails, the transition joins the *L_Init_1* label and restores the resources accessed by *methodA* to its previous state.

The next joinpoints of the transition are calls to the *methodB* method that execute in two distinct decision answer transitions. For both these joinpoints *previousJoinpoint* resolves to the *methodA* joinpoint. The *previousJoinpoint* label therefore resolves to the label introduced at the first joinpoint, *L_Init_1*. Whenever *methodB* fails, the state machine restores the resource accessed by *methodB* to its original state and joins the *L_Init_*1 label. The state machine then proceeds by restoring the previous resource accessed along the transition, the *methodA* resource, and completes by entering the *FAILURE_Init* state introduced by the aspect in the scope of the transition.

In the case of Figure 21, the ordering relationships between joinpoints can be resolved statically. In the more general case, the runtime environment needs to maintain a jump table to resolve the precedence relationships between joinpoints at runtime.

The aspect has the effect of making successive accesses to methods that implement the Recoverable interface atomic along a state transition. At the code level, the implementation of the corresponding aspect would be problematic [Kienzle02]. State machines provide aspects with additional abstractions that can be used at the pointcut level to abstract from implementation details, but also at the level of the advices, to augment the compositional expressiveness of aspects.

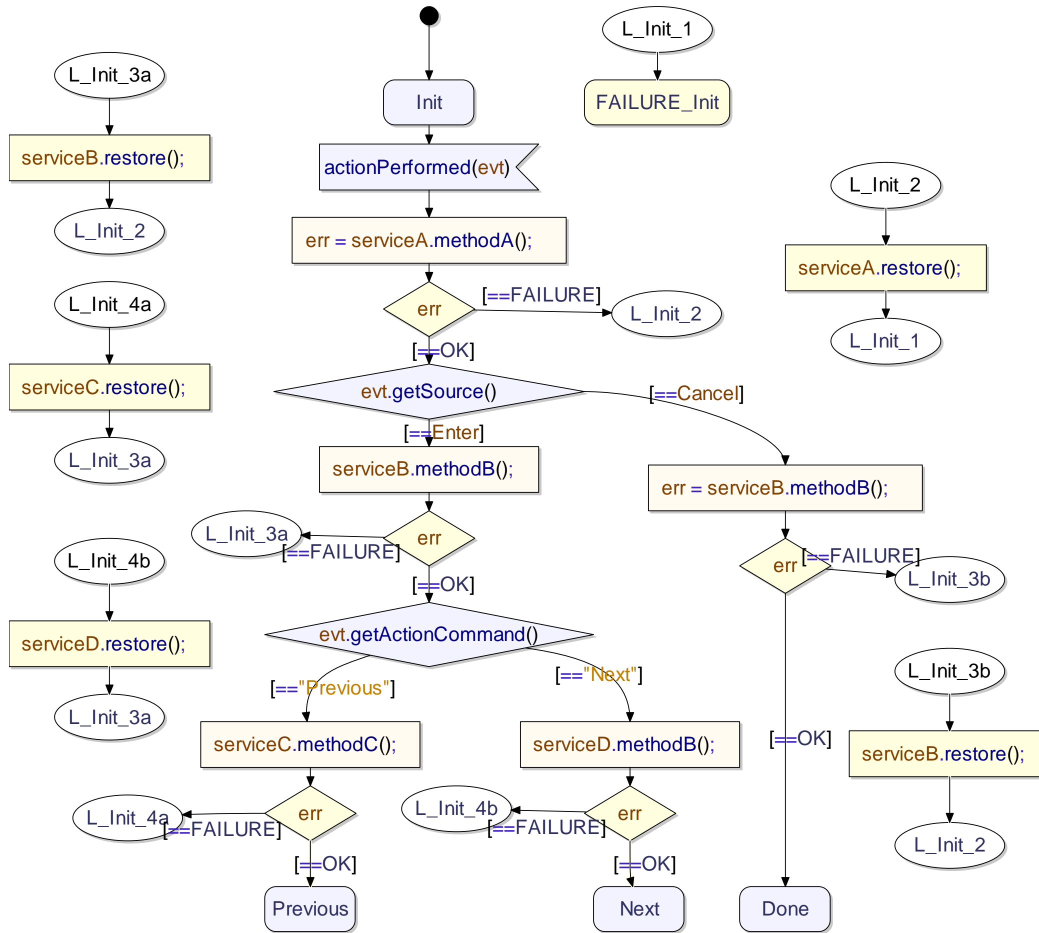


Figure 21. Partial representation of the result of weaving the atomicity aspect of Figure 20 in the state machine of Figure 18

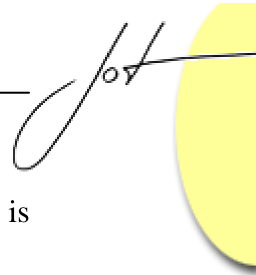
Pointcut Composition and Aspect Dependencies

Pointcut composition and the resolution of dependencies between aspects in WEAVR are discussed in [Zhang07].

5 DISCUSSION AND FURTHER WORK

The current version of the WEAVR tool fully supports the action and transition joinpoint model that was presented in the first part of the previous section. The tool implements transition joinpoint selection through state and action reachability analysis and action joinpoint selection through control flow analysis over states and transitions.

Other features, such as scoped state introductions, abstract aspects and the composition of state machines according to orthogonal regions semantics are at the stage of prototypes. The aspects that are currently used in production are aspects that fall in the category of application-specific, implementation-level aspects such as tracing aspects or



the connection timer aspect presented in Section 3. The use of the WEAVR tool is therefore still limited to aspects that handle simple, implementation-level concerns.

We believe that most of the benefits from aspect orientation are to be obtained from the deployment of aspects that maintain different use cases or features isolated at the implementation level. This allows base use cases or features to be reused across applications, and encourages the definition of explicit specifications of the use-cases/features interdependencies.

The deployment of such aspects requires fundamental changes in the development process. It requires development teams to be assigned responsibilities in terms of use cases rather than in terms of the physical components of the system such as the RNC or the GSN. Such a change requires the definition of precise behavioral interfaces between features and components, such as WEAVR behavioral specifications.

Further development includes the full implementation of some of the prototype features discussed earlier in the paper. Other important features concern the implementation of more robust pointcut composition and joinpoint interference resolution mechanisms. This topic covered in [Zhang07].

A large development effort is also dedicated to the development of joinpoint visualization and joinpoint selection validation mechanisms, as well as the development of a simulation engine for aspect-oriented models. These topics where not covered in the paper but are important with respect to further adoption of the WEAVR tool, and to establish trust in aspect-oriented technologies.

Further development also includes the prototyping of aspect-oriented versions of some of the uses cases of the communication systems currently in production, such as the relocation component discussed in Section 2. This process will produce metrics that will help in the evaluation of the approach.

Finally, further research includes the formalization of the WEAVR behavioral specification language and WEAVR realization mappings using temporal logic. This will also be directly applicable to the transition selection mechanism, which was informally presented in Section 5.

6 RELATED WORK

The WEAVR aspect-oriented modeling language and tool integrates results from different research areas of Model-Driven Engineering and Aspect-Oriented Software Development. Research areas related to this work include:

- work on the conceptual representation of aspects and crosscutting concerns
- work on aspect-oriented and general purpose model transformation
- work on system specification and model refinement
- work on expressive pointcut descriptors, such as stateful aspects
- work on semantically-based pointcut descriptors

The Aspect-Oriented Modeling (AOM) [AOM] community aims at identifying common, language independent aspect-oriented concepts that can be represented at different levels of abstraction, to support aspect orientation at different stages of software development, including requirements engineering, software architecture, design and implementation.

Approaches that focus on the conceptual representation of aspect-oriented techniques include the work on Join Point Designator Diagrams (JPDDs) [Stein06], the work on Theme/UML [Clarke05], and the work on aspect-oriented software development with use cases [Jacobson04]. Join Point Designation Diagrams are an abstract visual means to render aspect-oriented join point selections. Theme/UML provides an UML notation for the design of aspect-oriented programs and an approach to map aspect-oriented designs to code. The AOSD use case approach focuses on the modularization of UML use cases using aspects, and is particularly relevant to this work. Yet, these approaches are oriented towards the manual implementation of systems from designs expressed as informal models, rather than model-driven engineering.

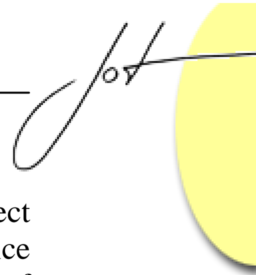
AOM approaches that focus on Harel statecharts include the Aspect-Oriented Statechart Framework (AOSF) [Elrad02]. The AOSF supports the weaving of independent state charts into a composite state chart, where each of the original state charts resides in its own orthogonal region.

Approaches that emphasize code generation from models generally use model transformation techniques to weave aspect-oriented models directly at the model level, before code generation. Aspect weaving is considered as a particular form of model transformation. Examples of model weavers are C-SAW [Gray01] and the ATL ModelWeaver [Bezevin04].

Approaches that target the refinement of action systems through superposition [Kurki-Suonio03] allow independent refinements of common base systems to be composed in an aspect-oriented fashion. This work is very relevant to the behavioral specification method described in this paper and supports very similar concepts. The approach adopts Temporal Logic of Actions (TLA) as the programming logic to verify fairness properties. The WEAVR approach has different aims, and rather focuses on the modular specification of use cases and features. WEAVR specification would better be described using computational tree logic rather than TLA.

The work on Stateful aspects [Douence04] and more advanced control flow pointcut composition operators is also relevant to this work. Stateful aspects can capture a sequence of events in a system. WEAVR pointcuts include state based conditions, which implicitly capture the history of the system execution. We consider the need for stateful pointcut designators as a symptom that the system implements reactive behavior. As such, the system is better decomposed using the natural decomposition for reactive systems, state machines.

Finally, this work is relevant to research that aims at providing solutions to the fragile pointcut problem [Gybels03]. There are two main research directions in addressing this problem. The first direction of research advocates restricting the expressiveness of aspects by preparing modules for specific, anticipated aspects. Approaches such as Open



Modules [Aldrich05] propose to move aspect pointcut descriptors from the aspect definition to the interfaces of modules. Aspects are therefore only allowed to advice joinpoints that are explicitly declared in the module interface. A second direction of research focuses on methods that allow pointcut descriptors to be defined at a higher level of abstraction, in terms of the program semantics [Ostermann05].

WEAVR restricts pointcut expressions to entities that are explicitly defined in the behavioral specification of a module. It therefore combines both approaches. Yet, WEAVR is able to maintain some of the expressiveness of unrestricted aspect-oriented programming language by allowing joinpoints that are not directly exposed in the specification of the module to be inferred from behavioral specifications.

7 CONCLUSIONS

We presented three major issues that impede the automation of development tasks during the refinement of system architectural models and specifications to system implementation.

First, there is a mismatch between problem structure and modeling language abstractions which causes discrepancies between system specification and system implementation.

Second, interdependent use cases and features are hard to implement in isolation from each other, which forces these dependencies to be declared implicitly in the system implementation rather than being declared explicitly at the level of their interfaces. Base use cases thereby become overspecialized for a specific context and are hard to reuse.

Third, many application-specific implementation details can not be handled systematically through code generation, which forces their manual integration in the system, at multiple locations.

We illustrated these problems through an example from the telecom domain and proposed aspect-oriented solutions to these problems, expressed in the WEAVR aspect-oriented modeling language.

Finally, we discussed some of the more advanced WEAVR features and showed how the use of specification-level pointcuts have the potential to address some of the weaknesses of aspect-oriented programming languages through inference of implementation-level joinpoints from pointcuts expressed in terms of their behavioral specification.

This technique might be essential for maintaining different use cases or features isolated at the implementation level. The modularization of use case implementations allows base use cases or features to be reused across applications while encouraging the definition of explicit specifications of the use-cases/features interdependencies.

REFERENCES

[Bast03] Bast, W., Kleppe A., Warmer,J.: MDA Explained: The Model Driven Architecture: Practice and Promise, Addison-Wesley, 2003.

[Frankel03] Frankel, D.S.: Model Driven Architecture: *Applying MDA to Enterprise Computing*, John Wiley & Sons, 2003.

[Jacobson04] Jacobson, I. Ng, P-W.: *Aspect-Oriented Software Development with Use Cases*, Addison-Wesley, 2004.

[Weigert03] Weigert, T., Dietz, P.: *Automated Generation of Marshalling Code from High-Level Specifications*, In SDL 2003: System Design, LNCS 2708, Springer Verlag, 2003.

[Mellor02] Mellor, S.J., Balcer, M.J. *Executable UML: A Foundation for Model Driven Architecture*, Addison-Wesley, 2002.

[Kiczales97] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J.-M. Loingtier, J. Irwin: *Aspect-Oriented Programming*, in Proceedings of the 11th European Conference on Object-Oriented Programming, Jyväskylä, Finland, LNCS 1241, pp. 220-242, Springer Verlag, , 1997.

[Elrad04] Elrad, T., Filman, B., Aksit, M., Clarke, S.: *Aspect Oriented Software Development*, Addison Wesley, 2004.

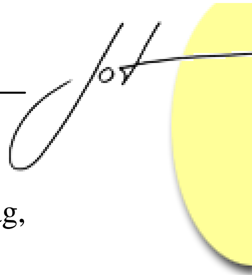
[Kiczales05] Kiczales, G., Mezini, M.: *Aspect-Oriented Programming and Modular Reasoning*. In proceedings of the International Conference on Software Engineering, St. Louis, USA, pp 49–58, ACM Press, 2005.

[Gybels03] Gybels, K., Brichau, J.: *Arranging Language Features for More Robust Pattern-Based Crosscuts*. In proceedings of the International Conference on Aspect-Oriented Software Development, , Boston, USA, pp 60–69, ACM Press, 2003.

[Ostermann05] Ostermann, K., Mezini, M., Bockisch, C.: *Expressive Pointcuts for Increased Modularity*. In proceedings of the 19th European Conference on Object-Oriented Programming, Glasgow, Scotland, LNCS 3586, pp. 214-240, Springer Verlag, 2005.

[Cottenier07] Cottenier, T., van den Berg, A., Elrad, T.: *Joinpoint Inference from Behavioral Specification to Implementation*, To appear in proceedings of the 21st European Conference on Object-Oriented Programming, Berlin, Germany, LNCS, Springer Verlag, 2007.

[Cottenier05] Cottenier T., Van Den Berg A., Elrad T.: *Modeling Aspect-Oriented Compositions*, In satellite proceedings of the 8th International Conference on Model Driven Engineering Languages and Systems,



Montego Bay, Jamaica, LNCS 3844, pp. 100-109, Springer Verlag, 2005.

[Pang04] Pang, A. C., Chen, J. C., Chen, Y. K., Agrawal, P.: *Mobility and session management: UMTS vs. cdma2000*, IEEE Transaction on Wireless Communications, Volume 11, Issue 4, pp. 30-43, IEEE Press, 2004.

[Lin01] Lin, Y. B., Haung, Y. R., Chen, Y. K, Chlamtac, I.: *Mobility Management: From GPRS to UMTS*, Wireless Communications and Mobile Computing, Volume 1, Issue 4, pp. 339-60, John Wiley & Sons, 2001.

[Harel87] Harel, David: *Statecharts: A Visual Formalism for Complex Systems*. Science of Computer Programming 8, pp. 231-274, Elsevier, 1987.

[Bougé88] Bougé, L., Francez, N., A *compositional approach to superimposition*, In proceeduings of the 15th Symposium on Principles of Programming Languages, San Diego, California, pp. 240-249, ACM Press, 1988.

[Katz93] Katz, S., *A superimposition control construct for distributed Systems*, In ACM Transactions on Programming Languages and Systems, Volume 15, Issue 2, pp. 337-356, ACM Press, 1993.

[TAU] Telelogic, *TAU*, http://www.telelogic.com/corp/products/tau/g2/

[Kienzle02] Kienzle, J., Guerraoui, R.: *AOP - Does It Make Sense? The Case of Concurrency and Failures*, In Proceedings of the 16th European Conference on Object-Oriented Programming, Malaga, Spain, LNCS 2374, pp. 37-61, Springer Verlag, 2002.

[Zhang07] Zhang, J., Cottenier, T., van den Berg, A., Gray, J.: *Aspect Composition in the Motorola Aspect-Oriented Modeling Weaver*, in Journal of Object Technology, Volume. 6, Issue 7, ETH Zurich, 2007.

[AOM] AOM Workshop, *AOM*, http://www.aspect-modeling.org/

[Stein06] Stein, D., Hanenberg, S., Unland, R.: *Expressing Different Conceptual Models of Join Point Selections in Aspect-Oriented Design*, In Proceedings of 5th International Conference on Aspect-Oriented Software Development, Bonn, Germany, pp. 15-26, ACM Press, 2006.

[Clarke05] Clarke, S., Baniassad, E.: *Aspect-Oriented Analysis and Design. The Theme Approach*, Addison-Wesley, 2005.

[Elrad02] Elrad T., Aldawud O., Bader A.: *Aspect-Oriented Modeling - Bridging the Gap Between Design and Implementation.* In Proceedings of the 1st International Conference on Generative Programming and Component Engineering, Pittsburgh, USA, LNCS 2487, pp. 189-201, Springer, 2002.

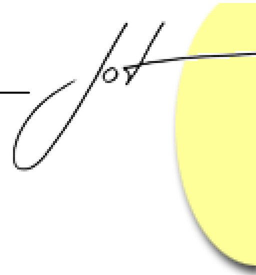
[Gray01] Gray, J., Bapty, T., Neema, S., Tuck, J.: *Handling crosscutting constraints in domain-specific modeling*. In Communications of the ACM, Volume 44, Issue 10, pp.87-93, ACM Press, 2001.

[Bezevin04] Bézivin, J., Jouault, F., Valduriez, P.: *First Experiments with a ModelWeaver*, Workshop on Best Practices for Model Driven Software Development held in conjunction with the 19th Conference on Object-Oriented Programming, Systems, Languages, and Applications, Vancouver, Canada, 2004.

[Kurki-Suonio03] Kurki-Suonio, R.: *Action systems in incremental and aspect-oriented modeling,* In Distributed Computing, Volume 16, Issue 2-3, pp. 201-217, Springer-Verlag, 2003.

[Douence04] Douence, R., Fradet, P., Sudholt, M.: *Composition, Reuse and Interaction Analysis of Stateful Aspects*, In proceedings of the 3rd International Conference on Aspect-Oriented Software Development, Lancaster, UK, pp. 141-150, ACM Press, 2004.

[Aldrich 05] Aldrich, J. *Open Modules: Modular Reasoning about Advice*. In Proceedings of the 19th European Conference on Object-Oriented Programming, Glasgow, Scotland, LNCS 3586, pp. 144-168, Springer Verlag, 2005.



About the authors

Thomas Cottenier works as a researcher for the Motorola Software Group and is a PhD student at the Computer Science department of the Illinois Institute of Technology, Chicago. Thomas holds a Communication Engineering degree from the Université Libre de Bruxelles, Brussels and an MS in Computer Engineering from the Illinois Institute of Technology. Thomas' interests include Aspect-Oriented Software Development and Model-Driven Software Development applied to telecommunication and distributed computing systems. He can be reached at thomas.cottenier@motorola.com.

Aswin van den Berg has a PhD in Computer Science from Cornell University and has been working with the Software and System Engineering Research Lab at Motorola Labs from 2000 until 2007 and for the Motorola Software Group in Schaumburg, USA since 2007. His graduate work focused on Program Transformation Systems and Higher Order Attribute Grammars. At Motorola he has been working on automatic code generation from SDL/UML software models and is now the project leader of the Motorola WEAVR project. He can be reached at aswin.vandenberg@motorola.com.

Tzilla Elrad is a research professor at the Computer Science Department, Illinois Institute of Technology. Tzilla received her BS in mathematics and physics from the Hebrew university, Jerusalem, her MS in Computer science from Syracuse University, NY and her PhD from the Technion, Haifa. Tzilla's current interests are in the design of concurrent programming languages and aspect-oriented software development. She can be reached at elrad@iit.edu