The Motorola *WEAVR*:
Model Weaving in a Large Industrial Context

Thomas Cottenier
Motorola Labs
Illinois Institute of Technology

Aswin van den Berg
Motorola Labs

Tzilla Elrad
Illinois Institute of Technology

thomas.cottenier@mot.com   aswin.vandenberg@mot.com   elrad@iit.edu

ABSTRACT

This paper reports on the development of an Aspect-Oriented Modeling engine and its initial deployment within the Model-Driven Engineering environment used in production at Motorola. The development environment is presented in detail, through a small example, and the current state of Aspect-Oriented Software Development technologies are discussed in this context. The report presents the particular decision made concerning the design and the deployment of the Motorola **WEAVR**[1] Aspect-Oriented Modeling engine in light of the particular needs of the telecom system engineering industry. First, we motivate a *model weaving* approach as opposed to the more traditional aspect modeling, code generation and code-level weaving approaches. Second, we present a novel join point model for transition-centric state machines, and discuss its use within a large industrial context. Finally, we report on the initial adoption of the weaving engine within production teams and its impact on the development process.

Keywords

Aspect-Oriented Modeling, Model-Driven Engineering, UML 2.0, Model Weaving

1. INTRODUCTION

This paper reports on the successful initial adoption of Aspect-Oriented Software Development technologies within one of the core business units at Motorola, the Networks and Enterprise Business Unit. The Networks and Enterprise unit is a provider of integrated voice and data communication end-to-end infrastructure. It delivers secure two-way radio, cellular and wireless broadband systems to government, service providers and enterprise customers worldwide. One of its core research and development activity deals with the development of the forthcoming WiMAX (802.16e) infrastructure to fulfill the demand for mobile broadband wireless solutions and take operators to the 4th generation of mobile wireless networks.

The telecom infrastructure software industry has a long history of Model-Driven Engineering (MDE) practices. It pioneered model-driven software development techniques starting in the 70's, with the Specification and Description Languages (SDL) ITU recommendation [1]. The SDL was initially conceived as a specification language to unambiguously describe the behavior of reactive, discrete systems in terms of communicating extended finite state machines. Since then, it was extended with mechanisms supporting object-orientation and has adopted a formal semantics described in terms of Abstract State Machines.

The formal base of SDL, its support for object-orientation, its easily understood finite state machine basis and its graphical representation have driven an important engineering investment in tools such as graphical editors, static analyzers, code generators and model simulators. The use of SDL has rapidly expanded from the area of system specification and documentation to the realm of system design and implementation.

The unambiguous semantics of SDL has enabled the industry to develop powerful code generators that take as input models and deliver highly optimized platform specific code, mostly in C and C++. The performance needs of the industry have secured a heavy investment in code generators that guarantee a performance overhead within 5% of the performance of the equivalent manually written code. These optimizing code generators have had an important effect of the system development process. The structure of the generated code is typically pretty different from the structure of the system models and prohibits the manual refinement of the system at the level of the code. This constraint has pushed more and more of the system implementation directly into the models.

Today, at Motorola, 50% to 85% of the telecom infrastructure systems are fully automatically generated from SDL and UML models depending on the divisions. The remaining 15% to 50% of the system typically deals with hardware interfacing software such as drivers, algorithmic code for signal processing or legacy code. The impacts of Model-Driven Engineering techniques on productivity, reduction in development effort and reduction in defects across the business unit are reported in [2].

The SDL community has profoundly impacted the standardization of the UML. The UML 2.0 has adopted many of the language features of SDL, including the composite-structure architecture diagrams, support for transition-centric state machines (see Section 3.2), which are characteristic of SDL behavior designs, and parts of the SDL action language semantics. While UML 2.0 models do not have precise semantics, they can be interpreted as SDL-like specifications using a lightweight profile. Precise UML 2.0 models can therefore unambiguously specify a system and be fully automatically translated into executable artifacts. The alignment of UML 2.0 and the latest SDL recommendations has enabled the industry to migrate towards the OMG standards while leveraging the investments in the SDL tools. Today, at Motorola, telecom infrastructure development is performed using UML 2.0 compliant modeling notations used along with fully automated, optimizing code generation.

---

[1] Motorola **WEAVR** is an add-in to Telelogic TAU [15]. We are pleased to provide free of charge licenses of the add-in to academia, for research purposes.

2. WEAVING ASPECTS IN MODELS

Telecom infrastructure software typically has a long life-time, which can span from 20 to 40 years. In the past, these systems provided few added services, and where pretty stable in their implementation.

Today, service providers have very rapidly changing requirements and desired features. For the same technology, different operators compete on the added-value services they provide. The increasing number of features to be supported by the infrastructure puts a tremendous strain on the architecture of the system. Many of those features impact the implementation of the system in terms of security policies, fault-tolerance capabilities, quality of service or session movement (seamless mobility), at multiple locations in the models.

In order to guarantee the maintainability of the system over a long period of time, in the presence of frequently changing requirements, it is absolutely essential to address the modularization of these concerns. These constraints have pushed an internal research effort towards developing better techniques and practices for the modularization of crosscutting concerns within the business unit.

Once the modularization capabilities of the UML 2.0 had been exhausted, this research effort naturally turned towards Aspect-Oriented Software Development technologies. Yet, it rapidly became clear that the existing technologies were not adapted to the Model-Driven Engineering environment deployed in production. Aspect-Oriented Programming languages are not applicable in this environment because the generated code artifacts do not exhibit the required structure on which aspects can hook onto. The code optimization process destroys the structural and syntactic correspondence between the system models and the generated code.

What is required is the full coordination of crosscutting concerns with the base system at the level of the models, especially behavioral models such as state machines. More specifically, we needed a model weaver for transition-centric state machines.

The literature on Aspect-Oriented Modeling using the UML appeared of limited help. Approaches to Aspect-Oriented design such as Jacobson's use case approach [3] or Theme/UML [4] adopt a *models as blue-prints* approach [5] where the focus is on specification and documentation rather than system implementation through automatic code generation. As a result, these approaches advocate a mapping between models describing crosscutting concerns and aspects written in an Aspect-Oriented Programming language, rather than the weaving of behavioral models. For the reasons described in the previous section, this option is not viable either for our purposes.

There is a little literature on model weaving, notably the C-SAW constraint weaver for the GME [6], and the ATLAS ModelWeaver [7]. While C-SAW does not support UML models, the ModelWeaver focuses on the static structure of systems rather than precise behavior. The only work on the modularization and coordination of crosscutting concerns using state machine is the work of Aldawud [8]. Aldawud's framework uses the concurrent region feature of Harel Statecharts to isolate crosscutting concerns. The "weaving" is then performed by manually correlating transition triggers and output events. This approach is not scaleable over large models and is not practical in the case of transition-centric state machines.

Interestingly, there is some more work being done on the weaving of sequence diagrams [9]. Yet, sequence diagrams are mainly used for specification and testing purposes. They are not appropriate for precise behavior modeling, especially when developing distributed or concurrent systems.

The Motorola Software and System Engineering Research Lab therefore dedicated a R&D effort to deliver an industrial strength Model Weaver adapted to the requirements of system engineering within the Networks and Enterprise Business Unit. This research effort resulted in a completely novel join point model for transition-centric state machines.

This paper describes and motivates the architecture and design of the Motorola **WEAVR**. Section 3 introduces the SDL style of UML 2.0 modeling and presents the development process and the Model-Driven Engineering stack deployed in production. Section 4 introduces the basic language constructs adopted to capture and deploy aspects at the modeling level. Section 5 presents a new join point model for transition-centric state machines and motivates its use through small examples of crosscutting appearing in models. Section 6 discusses the initial deployment of the weaver tool and its impact on the development process. Finally, Section 8 concludes this paper.

3. MDE AT MOTOROLA

This section describes the style of UML modeling used for system development and the Model-Driven Engineering process deployed in production.

3.1 UML Models for Communication Systems

Telecom systems can be characterized as being reactive discrete systems [10]. A reactive system is a system whose behavior is dominated by interactions between actions input to the system, and the reactions output by the system. A discrete system is a system whose interaction appears a discrete points and by mean of discrete events. These events are mostly represented as asynchronous signals that are exchanged between component instances and the environment. The natural decomposition for reactive behavior is the Harel Statechart [10] or state machine diagrams.

The style of UML modeling used for telecom infrastructure system development is highly influenced by the SDL. Beyond the use of class diagrams, this style of modeling is characterized by composite-structure architecture diagrams, transition-centric state machines and the use of an action language for the complete implementation at the model level.

3.1.1 Composite-Structure Architecture Diagrams

During architecture modeling, the internal structure of active classes is described from a communication point of view. This decomposition attributes responsibilities to class instances. Architecture modeling typically takes place after, or in parallel with, class modeling during the design phase.

Composite-structure diagrams define a hierachical decomposition of the system. They are used early in the development process to attribute implementation responsibilities to teams of developers with respect to the requirements of a system, and to individual

developers within a team, with respect to the requirements of a sub-system.

A composite structure diagram defines the internal run-time structure of an active class (a type of process or thread), in terms of other active classes instances. These building blocks are referred to as parts. Parts are also restricted to be instantiations of active classes.

Composite structure diagrams also express the communication within the active class by visualizing connectors between the communication ports of the 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.

Composite structure diagrams are pretty stable. They are unlikely to change once they have been defined. They specify the interfaces of the system and its components in terms of required and realized signals.

Figure 1 illustrates the composite structure diagram of a simple resource server. An instance of a server is composed of two subcomponents, one dispatcher and request handlers. The number of request handlers is unbound, and initially 0.

The dispatcher is responsible for forwarding external requests to a request handler. It therefore maintains a table of sessions. The session index identifies the process id (Pid) of a request handler through a context id (CID_t).

A request handler is responsible for granting access to a resource in a globally fault tolerant way. Access to the resource is only granted is all the resources required for the interaction are accessible. This is performed using a distributed transaction protocol, such as two-phase commit (2PC). The request, commit and abort Signals are typical of such protocols.
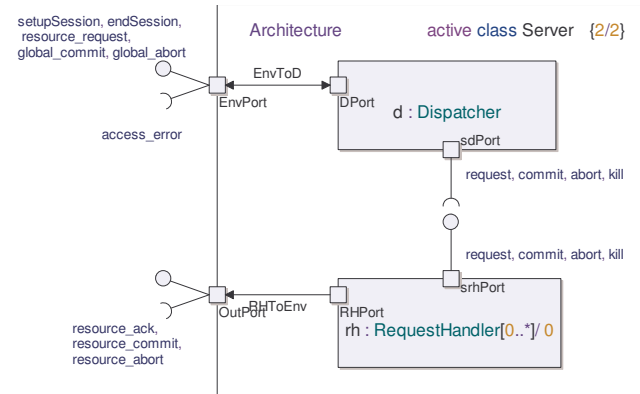

Figure 1. The Composite-Structure Architecture diagram for a simple server

3.1.2 Transition-Centric State Machines

In order to obtain an executable model, the detailed behavior of operations and active classes must be specified. This is done during behavior modeling.

A behavior specification may contain states (a state machine implementation), or it may be stateless (an operation body). Whether a state machine implementation or an operation body is used depends on the particular behavior to be modeled. State machine implementations are preferable when the behavior has a reactive nature, that is, its execution heavily depends on the

history of the system. The natural decomposition of reactive behavior uses hierarchical states and behavioral inheritance, as opposed to a classic inheritance hierarchy. Behavioral inheritance is out of the scope of this paper. A typical system would therefore contain both state machine implementations and operation bodies.
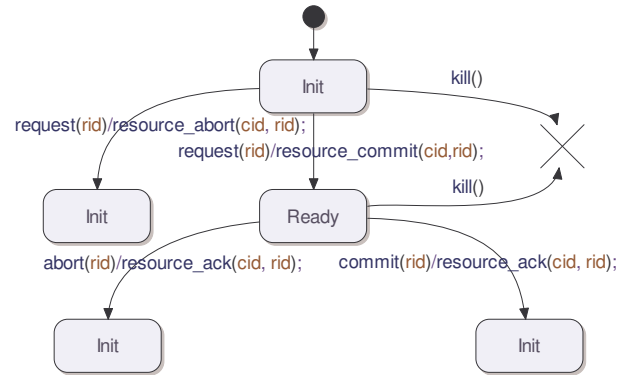

Figure 2. Specification of the observable behavior of a request handler as a state-centric state machine

The traditional representation of a state machine is a state-centric view of a state machine.

Figure 2 is a state-centric *specification* of the external behavior of a request handler. It defines the input signals that trigger transitions, and the output signals that are fired on the way. State-centric state machines give a good overview of the external behavior of the system but are not practical for defining the implementation of a system.

Transition-centric state machines provide a better view of the control flow and the communication aspects of a specific set of transitions. They are primarily used for defining the detailed internal behavior of a reactive component. Transition-centric state machines use explicit symbols for different actions that can be performed during the transition. They make the control flow explicit using decision actions, represented as diamonds.
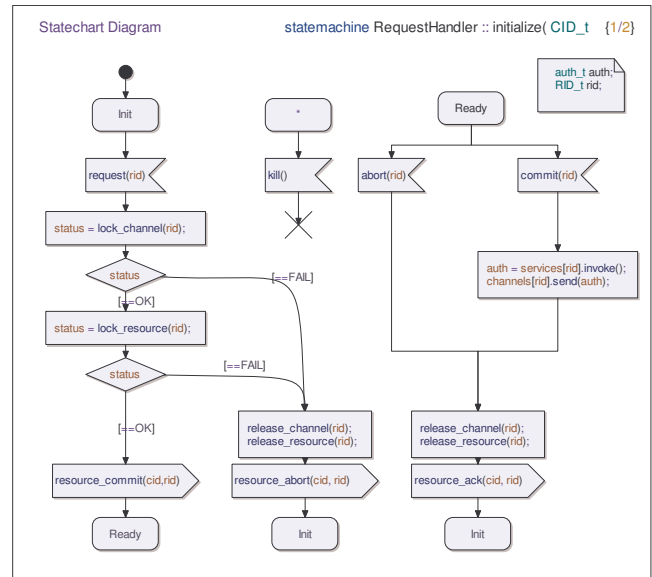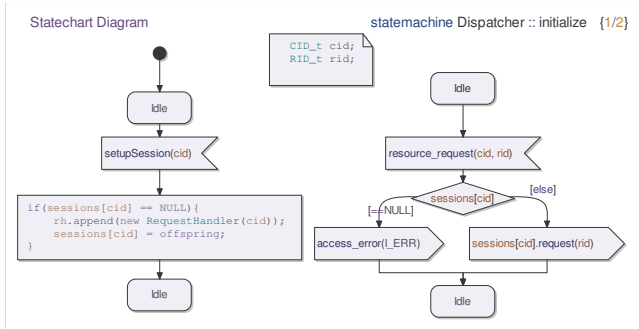

Figure 3. Implementation of a request handler as a transition-centric state machine

Figure 4. Implementation of the dispatcher as a transition-centric state machine

Figure 3 depicts the complete behavior implementation of the request handler, as a transition-centric state machine.

Similarly, Figure 4 shows a portion of the implementation of a the dispatcher as a transition-centric state machine.

3.1.3 Action Language

The use of an action language [11] makes UML models executable, i.e. it allows designers to test and simulate models and to fully automatically generate executable code.

An action Language is intended to be programming language independent. It includes operations that support the synchronous manipulation of objects, the generation and handling of signals, and the logical constructs that support the specification of algorithms.

Transition-centric state machine embed actions in their transitions, as shown in Figures 3 and 4. Examples of supported actions are variable definition, assignment, new, output, set timer, expression statement such as calls, if statement, while statement and delete statement.

3.2 Model Simulation, Execution and Testing

Telecom infrastructure software is typically developed in parallel with the hardware platforms the software is designed to run on. In many cases, the platforms are not finalized yet when key features of the software need to be validated, tested and verified. The ability to simulate and test the models, independently of the target platform, early in the lifecycle, is essential.

In general, the ability to simulate and test models early in the development lifecycle is key to the success of Model-Driven Engineering technologies, and is the main advantage of translationist approaches, a.k.a approaches that emphasize fully automatic code generation through model translation.

The models of Figures 1, 3 and 4, along with the corresponding class diagram, fully implement the base functionality of the resource server example. Although the access methods of the resources and the channels themselves have not yet been implemented (they are platform specific), the base functionality of the server can be executed in a simulation environment, tested, and validated for conformance to its specification (Figure 2). Figure 5 displays a trace generated by the model verifier, for a successful resource access. The generated sequence diagram can be displayed at a much finer grained level of detail. Figure 5 only displays state transitions and the signals exchanged between active class instances and the environment.
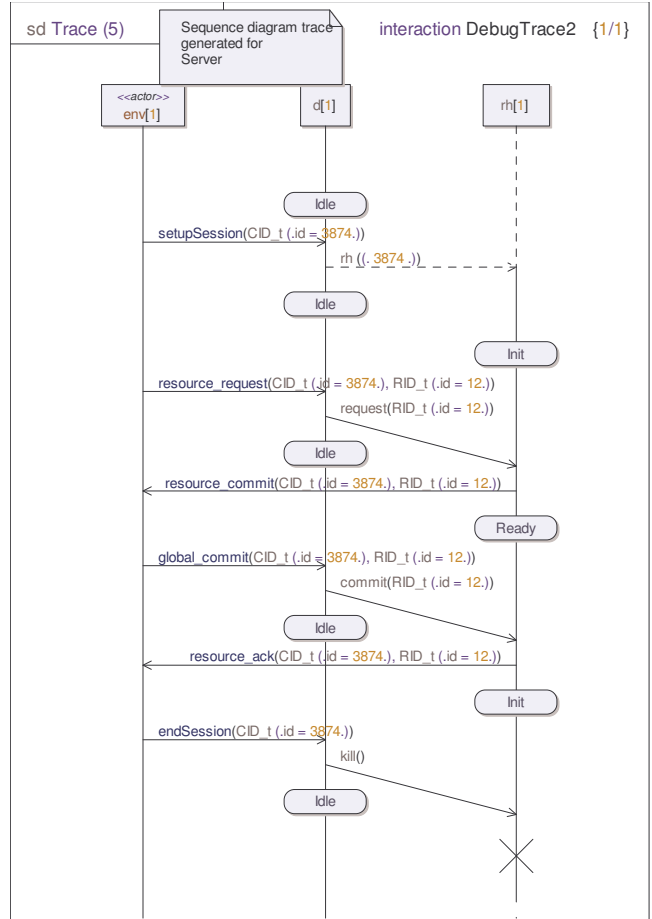
Figure 5. Sequence Diagram trace generated by the model simulator for a successful resource access, for the models of Figure 1, 2 and 4.

The simulation environment allows test cases to be fed to the system. Test cases are either derived from the system requirements manually, or are generated from the system specification models, such as state-centric state machines. The test cases themselves are represented as sequence diagrams to which verdicts are associated, or in textual form, as TTCN (Testing and Test Control Notation) [11] test case definitions.

Once the system has been validated and thoroughly tested, the models can be translated to platform specific executables and tested in the field.

3.3 Crosscutting in Behavioral Models

The two-phase commit problem presented in the previous section is a simplified representation of a real problem we have in production. One of the systems under development is composed of a large number of distributed subcomponents. For an interaction to occur successfully, all those components need to operate in a synchronized fashion. If one resource or communication channel in the system cannot be accessed safely, the interaction needs to be aborted or delayed. As a result, each component needs to implement a variant of 2PC, for each component it communicates with, which amounts to a number of 2PC request handlers that is quadratic to the number of components. Each development team needs therefore to re-

implement 2PC in the context of the specific resources that are managed. In practice, different teams would implement the same concern slightly differently, which leads to inconsistencies and important replication of effort.

There is therefore a strong motivation to separate the implementation of 2PC from the implementation of the specific resource access methods, so that the common behavior can be implemented once and instantiate in the context of all components. The concern cannot be encapsulated using traditional OO or statechart decomposition techniques because the behavior that is specific to each subcomponent interacts with the control flow of 2PC.

The next section introduces the Motorola WEAVR, an Aspect-Oriented Modeling engine for UML 2.0 state machines.

4. THE MOTOROLA *WEAVR*

The Motorola WEAVR is an add-in to Telelogic TAU [15] that performs 4 distinct functions. First, it includes a profile that allows developers to define Aspects in UML 2.0. Second, it provides a join point visualization engine that allows the effects of an aspect on a model to be visualized and validated. Third, it performs full aspect weaving at the modeling level. Finally, it includes a simulation engine that allows aspect models to be simulated, without breaking the modular structure of Aspects.

4.1 Aspect-Oriented Modeling

4.1.1 Aspects

In the *WEAVR*, an aspect is a class that is extended by the <<aspect>> stereotype. It can contain owned members such as attributes, operations, signal definitions or ports, which are treated as inter-type declarations. An aspect can also contain pointcuts and connectors. A connector is the equivalent of an AspectJ advice.

Pointcuts and connectors are operations that are extended by the <<pointcut>> and <<connector>> stereotypes, respectively. The implementation of pointcuts and connectors are state machine implementations. Connectors are bound to specific pointcuts using the <<binds>> dependency. The order of precedence of connectors applying to the same join points within an aspect is defined using the <<follows>> dependency.

The scope of an aspect can be specified explicitly by declaring <<crosscuts>> dependencies from the aspect class to the packages or classes to which the aspect applies. If no such dependency is specified, the aspect applies to the complete system. Other stereotypes are used to define the order of precedence and constraints between Aspects. For a discussion on pointcut composition and aspect composition in the Motorola WEAVR, see [13].

Figure 6 illustrates a simple tracing aspect that is deployed in the scope of the server of Figure 1. The aspect contains one tracing connector that is bound to six pointcuts. The pointcuts match all occurrences of 6 distinct type of events: call expression actions (method calls), output actions (sending a signal), method executions, the initialization transition of a state machine, state transitions and the transitions that terminate state machine instances.

The <<binds>> stereotype binds the parameter or arguments exposed by the pointcuts to the connector they are bound to.
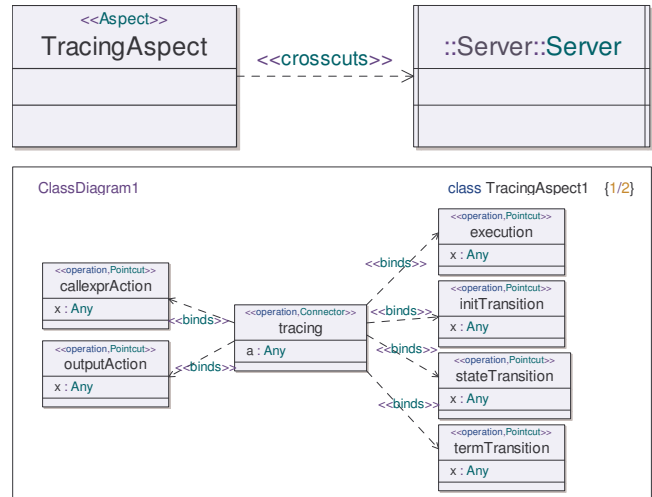
Figure 6. Deployment of an aspect that traces call expression actions, output actions and transitions in the scope of the server active class

The type 'Any' indicates a wildcard on the type of the arguments/parameters. Figure 6 only displays a connector and pointcuts that have one parameter. The complete tracing aspect includes pointcuts and connectors for the different combinations.

4.1.2 Pointcuts

The modeling environment provides two primary behavior decomposition dimensions that are complementary. While the reactive behavior of a system can be decomposed in state machines and sub-state machines (see behavioral inheritance, [10]), transformational behavior can be decomposed according to an Object-Oriented decomposition. Yet, both decompositions are primarily hierarchical. It is therefore not surprising that some concerns do not align with either of these decompositions. Crosscutting concerns need therefore to be handled according to both paradigms.

The *WEAVR* recognized two main categories of join points: Action join points, that capture call expressions, timer set actions or constructor calls and Transition join points, which capture sets of execution paths within a state machine.

A pointcut designator is expressed as a state machine implementation of a pointcut operation. The parameters of the pointcuts specify which arguments or parameters of the pointcut designator are exposed to connectors.

4.1.2.1 Action Pointcut Designators

An action pointcut designator is a state machine implementation that features one non-terminating action.

The *WEAVR* limits action pointcut designtors to call expression actions, output actions, create expression actions (constructor calls) and timer set and reset actions.

Figure 7 shows the pointcut designators for the *callExprAction* and *outputAction* pointcuts referred to in Figure 6. The *callExprAction* pointcut designator matches a call to any method with one parameter and one return parameter, and exposes the argument of the call to connectors.
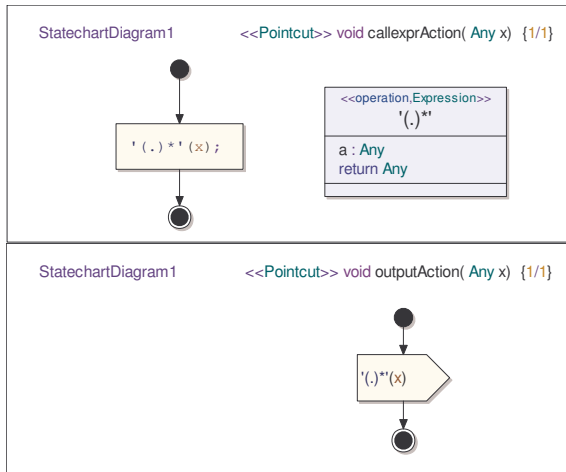
Figure 7. Action pointcut designators for a call and an output action, with one argument

4.1.2.2 Transition Pointcuts

A transition pointcut designator is a state machine implementation that features one transition. A transition pointcut designator is characterized by one starting state, an event occurrence or a method signature, and a terminating action, such as a next state action, a return action or a stop action. A transition pointcut designator can quantify both on the names of its states and on the signature of its event. Quantification over states is possible because state name are explicitly defined in the specification of a component (state-centric state machine).

Figure 8 shows the pointcut designators for the *execution* and *initTransition* pointcuts referred to in Figure 6. For both these pointcuts designators, the starting state is the start state. The pointcut designator matches transitions within the execution of a method, matched by the signature of the pointcut designator expression, which is annotated with the <<expression>> stereotype. The *execution* pointcut is equivalent to execution pointcut in AspectJ. It matches all the execution paths of a method, from the start state to the return termination action.

The *initTransition* pointcut designator captures the initialization transitions of all the state machine implementations that match the pointcut expression. An initialization transition is executed from the start state to some other state. Typically, the initialization transition performs tasks that are similar those performed by a constructor.
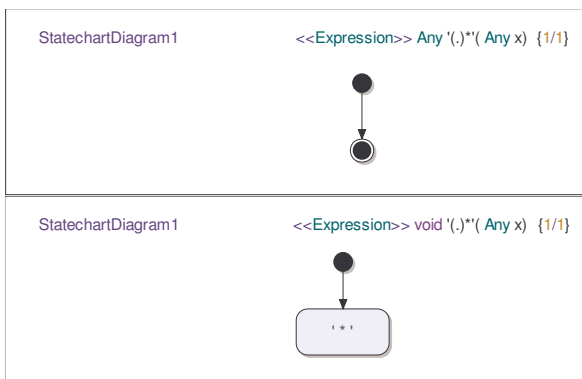

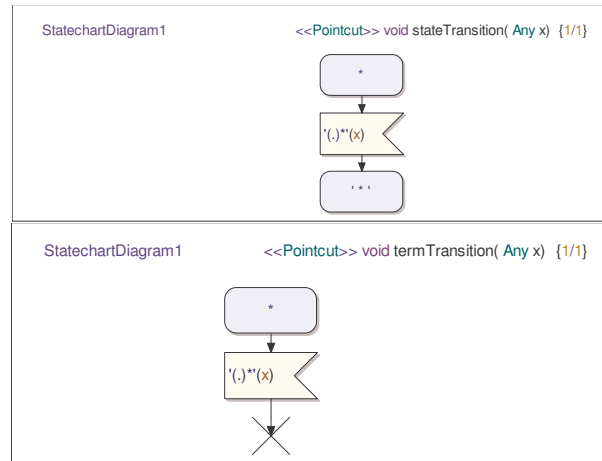**Figure 8. Expressions for the execution and initTransition pointcuts**

Figure 9. A transition pointcut designator as a triggered transition

Figure 9 shows the pointcut designators for the *stateTransition* and *termTransition* pointcuts referred to in Figure 6. These pointcut designators are implemented as triggered transitions and are interpreted in a particular way. A transition pointcut designator from state S to state T triggered by i matches the complement of the execution paths from S to NOT T, triggered by i, from all the execution paths from S to T, triggered by i.

$$sel(S \xrightarrow{I} T) = \{ \bigcup path(S \xrightarrow{I} T) \} \setminus \{ \bigcup path(S \xrightarrow{I} NOT(T)) \}$$

This matching method is very powerful because it can localize the important decision points in the execution of a state machine. We will illustrate this method in the examples of section 5. For a complete development of the matching mechanism refer to [14].

4.1.3 Aspect Connectors

In the *WEAVR*, connectors are always represented as the equivalent of around advices. A connector is a state machine implementation. It always contains a start state and a return state. A Connector can invoke the selection (action or transition) matched by the pointcut to which it is bound to, through the *proceed* keyword. A connector takes as parameters the arguments or parameters passed by the pointcut to which it is bound to. Furthermore, it can retrieve information on its instantiation context through the *thisJoin point* reflective API. Figure 10 illustrates a connector for a tracing aspect referred to in Figure 6.
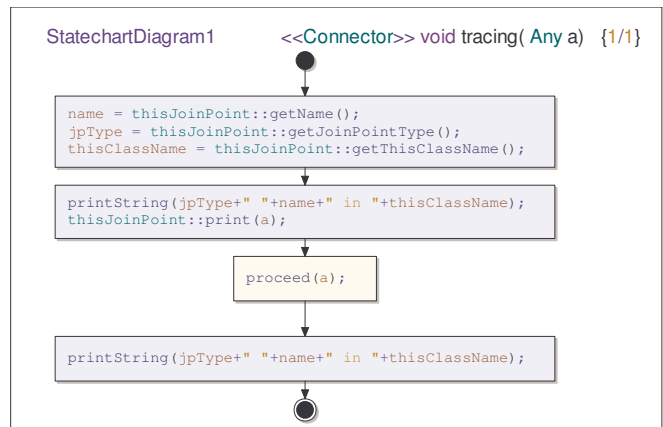

Figure 10. The tracing connector

4.2 Aspect Effect Visualization Engine

The transition matching mechanism has non-trivial semantics. It is therefore important to provide a visualization environment so that developers can validate the join points matched by the pointcuts of an aspect, and visualize the effects of the aspect at those locations. This is also very important in the context of increasing the trust developers have in the tool, and with respect to model simulation. The WEAVR therefore includes a visualization engine that annotates join points, delimits transitions matched by the transition pointcut designators and shows how connectors are *instantiated* in a specific context.

4.2.1.1 Join point Annotations

When the tracing aspect of Figure 6 is applied to the server of Figure 1, the visualization engine colors the symbols that correspond to join points in a distinctive color and annotates them with information about the aspect, such as the pointcut that captured the join point and the arguments or parameters that are exposed to connectors.

Figure 11 shows how the request handler is annotated by the visualization engine. The symbols containing action join points have been colored in pink, while transition join points are localized by green marks along the matching transitions. The triggers to those transitions have also been colored in pink.
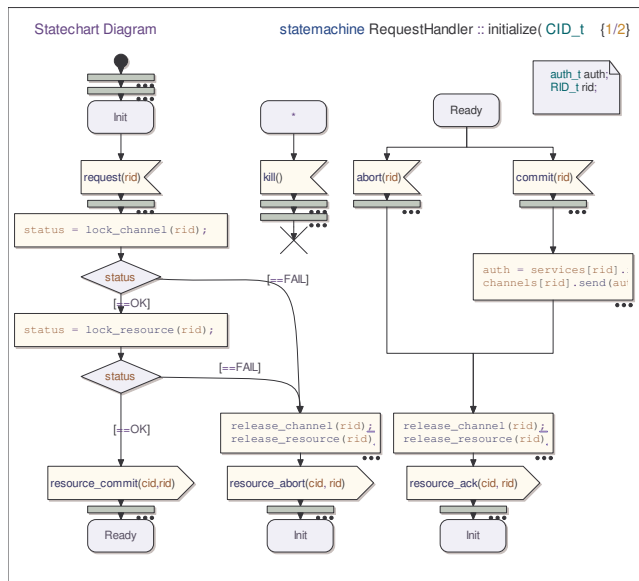
Figure 11. **The request handler as annotated by the visualization engine after the tracing aspect has been applied**

4.2.1.2 Connector Instances

When the user clicks one of the colored symbols, a state machine implementation diagram pops up and displays the instantiation of connectors applied to those locations.

Figure 12 shows an instance of the connector defined in figure 10, in the context of the triggered transition from state Ready to state Init, triggered by the commit signal. This transition matches the pointcut of Figure 9. Note that the reflective calls to *thisJoin point* have been resolved, the parameter *a* has been bound to the transition parameter *rid*, and that the *print(Any)* method has been resolved print statements that correspond to the *RID_t* datatype.
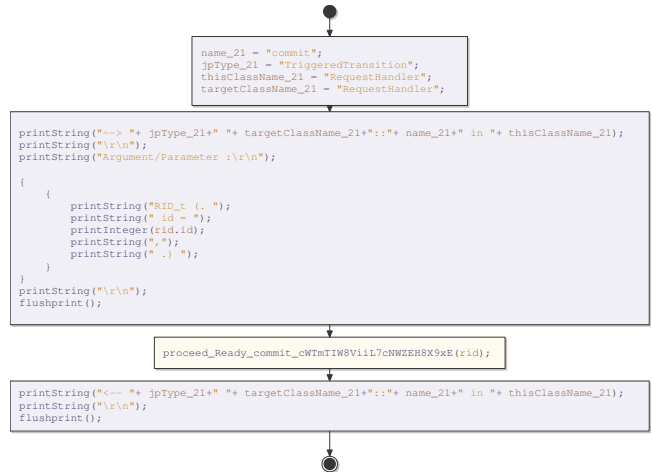
Figure 12. **An instance of the connector of Figure 10, as instantiated in the context of the transition from Ready to Init, triggered by the commit signal**

The *proceed* statement has been replaced by a generated method that *represents* the transition join point. The implementation of this method is represented in Figure 13.b. Figure 13.a represents the transition from Ready to Init, triggered by the abort signal.
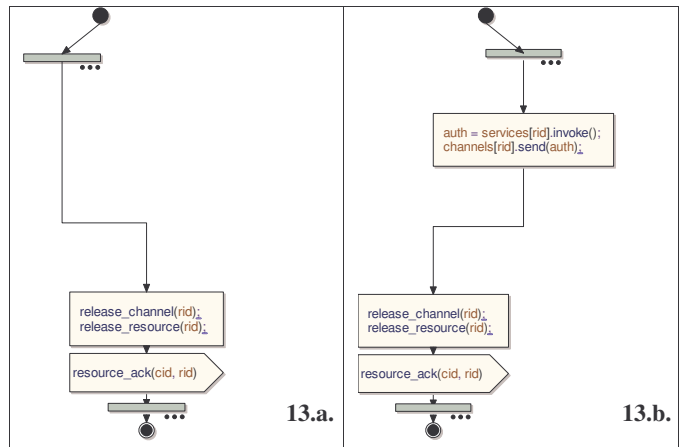
Figure 13. **Representations of the matched transitions from state Ready to state Init, triggered by the abort and commit signals, respectively**

4.3 Simulation, Weaving and Execution

The *WEAVR* tool includes an Aspect-Oriented simulation engine. The simulator executes Aspect-Oriented models in such a way that:

1. The modular structure of Connectors is maintained
2. The models execute in a semantically equivalent way to the woven model

When encountering a join point, the simulation environment "jumps" to the connector instances bound to the join points, and returns control to the original model after the execution of the connector instances. This allows developers to familiarize themselves with the semantics of Aspects and test the woven models before full weaving is performed. The final weaving is performed right before code generation. Developers are never supposed to manually inspect a woven model. Various optimizations can therefore be performed while the presentation elements of the model can be disregarded.

5. EXAMPLES

This section illustrates the use of the Motorola *WEAVR* through two examples of crosscutting concerns pertaining to the server example of Figure 1.

5.1 Transaction Timeout Aspect

The request handler of Figure 3 has one major weakness. It the instance enters the Ready state, but never receives a commit or abort signal, it will never terminate and will not be able to handle new requests. It is therefore safer to terminate the instance if neither the commit or abort signals occur after a given delay. The aspect of Figure 14 implements a timeout concern for the 2PC protocol. First it introduces a new transition in the request handler, a transition from Ready to termination, triggered by the *toTimer* timeout timer. Second it resets the *toTimer* timer before every transition from *Init* to *Ready*, triggered by request.
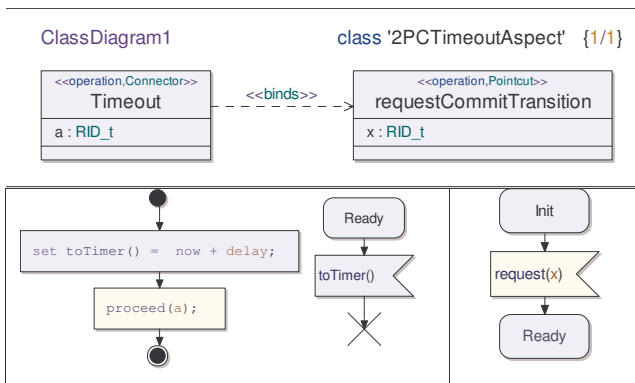

Figure 14. A timeout aspect for the 2PC protocol of Figure 3

This aspect illustrates the semantics of the transition join point matching mechanism discussed in Section 4.1.2.2. The transition matched by the pointcut is the *Decision Answer Transition* from the last decision point on the value of *status*, to the state *Ready*. Figure 15 shows the green delimitations marks for this transition in the visualization engine (15.a), the connector instance for this transition (15.b), and a representation of the matched transition (15.c), as an implementation of the generated method that replaced the *proceed* statement in the connector instance.
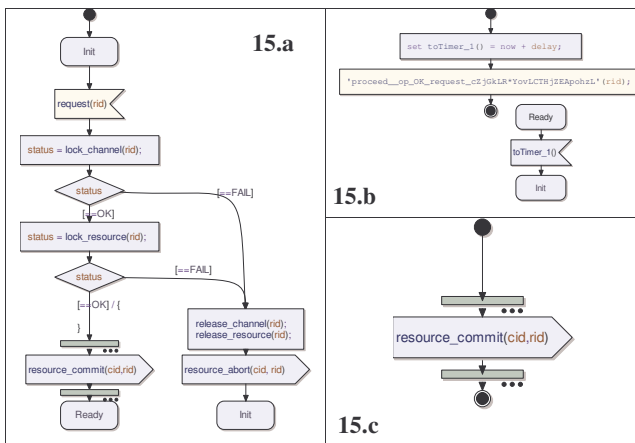

Figure 15. Delimitation of the transition matching the pointcut of Figure 14 in the visualization engine, the corresponding connector instance and matched transition

5.2 Two-Phase Commit Aspect

As discussed in Section 3.3, there is a strong motivation to separate the implementation of the 2PC protocol from the implementation elements that are specific to the request handler of Figure 3. Figure 16 shows how this separation can be achieved using the *WEAVR*. The *2PCAspect* package encapsulates the signal definitions and state transitions required to implement the specification of Figure 2. The ResourceAspect encapsulates all the methods that are specific to the particular resource managed by the request handler.
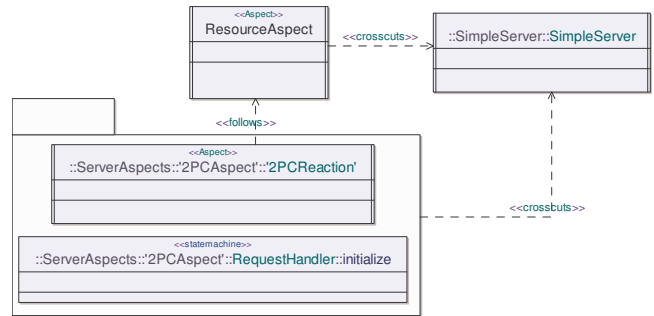

Figure 16. Aspect-Oriented implementation of the request handler of Figure 3. The aspects separate the implementation of the 2PC protocol from the elements that are specific to a particular resource request handler

5.2.1 Enforcement of the 2PC Specification

Aspect-Orientation enables the implementation of the 2PC protocol to be enforced, independently of the implementation of the specific resource access implementation. This is realized by separating the implementation of the transitions that handle the protocol input messages from the reactive output actions that control the signals that are required to implement the specification of Figure 2.

The implementation of the Aspect that enforces 2PC contains two main sub components.

First, the transitions that handle the 2PC input signals, and the corresponding state transitions they trigger are defined separately in a state machine implementation, represented in Figure 17. This state machine implementation is merged with the state machine implementation of the request handler, as indicated in Figure 16. The corresponding ports and signal definitions are defined in the 2PCAspect.

Second, the mandatory reaction of the system is defined be the aspect by the pointcuts and connectors illustrated in Figure 18.

This separation allows the reaction of 2PC to be enforced independently of the specific resource access methods.
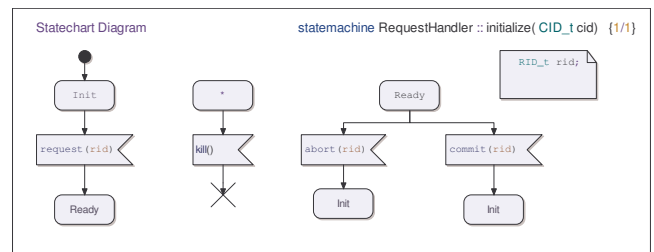

Figure 17. 2PC input message state transitions as a state machine introduction, to be merged with the request handler
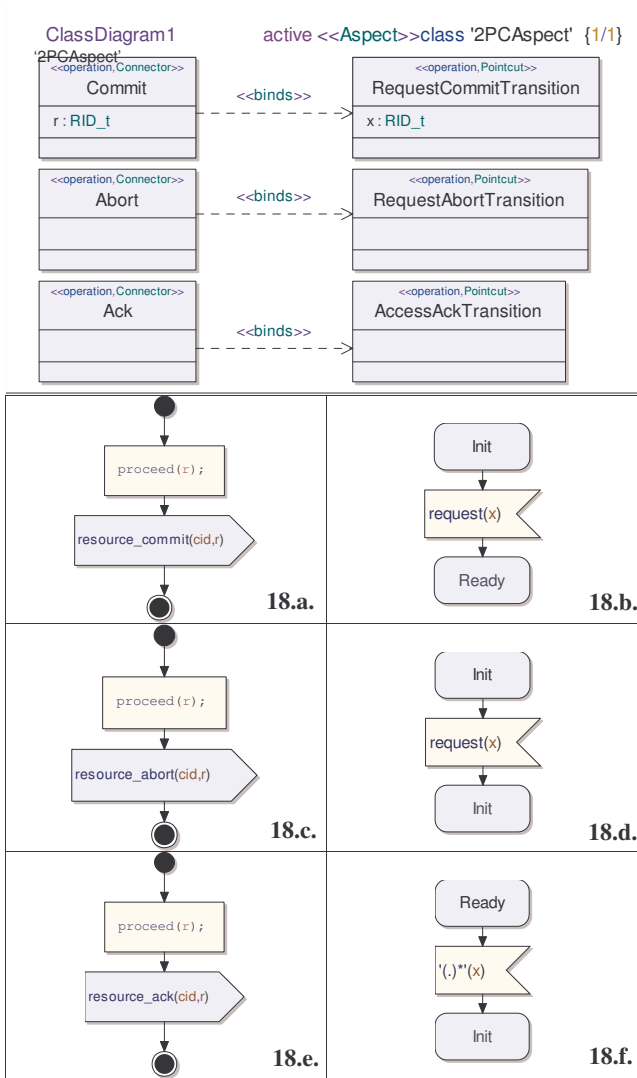
Figure 18. Pointcuts and connectors of the 2PC aspect, which enforce the reaction of the system, with respect to the transition defined in the state machine of figure 17
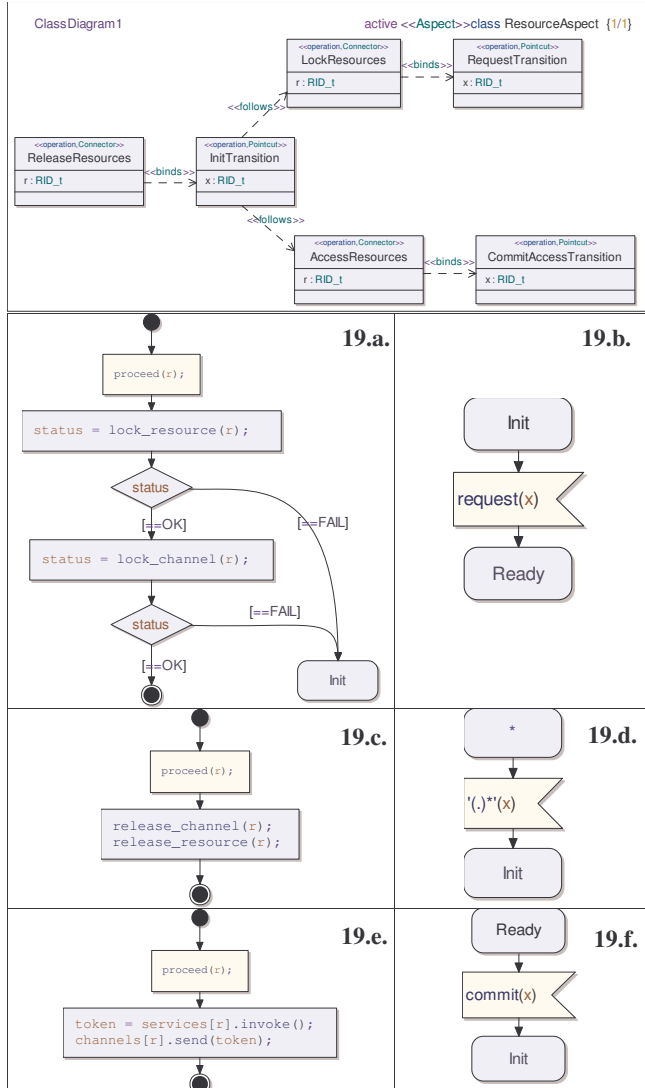

Figure 19. Binding diagram, pointcuts and connectors of the resource management aspect which encapsulates the specific resource access methods for the request handler

5.2.2 Resource Specific Behavior

The implementation of the resource access methods that are specific to each request handlers can now be defined independently of the 2PC protocol implementation. Figure 19 illustrates the binding diagram, the pointcut designators and the connector implementations for the resource management aspect.

Figure 19.a. shows how the transition from *Init* to *Init*, triggered by *request* is implemented. This also illustrates how states can be referred by connector implementation. This transition is not defined in the 2PC aspect because it is specific to the resource management methods. It is important to note that the transition that is introduced by connector 19.a is matched by pointcuts 18.d and pointcuts 19.d. The evaluation of these pointcuts needs therefore to occur after the connector 19.a has been applied to the system. This constraint is expressed by the <<follows>> dependencies in the diagrams of Figure 16 and Figure 19. The latter expresses that pointcut *initTransition* (19.d) is evaluated after connectors 19.a, *LockResources* and 19.f have been bound.

6. ADOPTATION AND DEPLOYMENT

The *WEAVR* is currently being used in production by development teams within the Network and Enterprise Business Unit. For the moment, adoption is limited to very simple aspects, such as the tracing aspect and the timeout aspect.

In particular, the tracing aspect proved very useful for initial deployment of the system on the platforms. These platforms are pretty primitive, and do not include a standard debugging environment. The model simulator includes advanced tracing capabilities, as shown in Figure 5. The tracing aspect allows us to provide the same tracing functionality for platform runs, without having to clutter the models with tracing statements.

The deployment of such aspects does not affect the development process at all, which facilitates adoption across development teams. Furthermore, as these aspects are pretty generic, they can be deployed by developers transparently, without requiring a complete understanding of the technology from their part.

Yet, we think that the most benefits from the use of Aspect-Oriented Software Development technologies could come from the deployment of aspects that *enforce* the conformance to specifications, such as the 2PC specification of Figure 2. The aspect-oriented implementation of the request handler discussed in the previous section could potentially reduce a lot of the development costs in one of the systems under development, while increasing consistency and the overall quality. A first estimate indicates that the overall size of the models could be reduced by 25 to 40%, if concerns such as fault-tolerance and security would be implemented as aspects.

Yet, the implementation and deployment of such aspects would be radical change in the development process. Today, the responsibilities concerning fault-tolerance and security requirements are specified at the level of the individual components. This has the advantage that the development teams know exactly how to implement these concerns for the components they are responsible for. Yet, it leads to an important replication of effort. The implementation of these concerns as aspects would require that these requirements would be specified at the system level, in a form that could easily be mapped to aspect implementations.

The degree of maturity of the Motorola *WEAVR* is still in an initial stage. It is critical to improve the degree of confidence in the tool. The following activities are therefore critical:

1. Further develop the visualization engine and the simulation and testing capabilities for aspect-oriented models. This is essential to increase the degree of understanding and confidence in aspect-oriented technologies among the development teams
2. Accelerate and generalize the deployment of simple aspects such as the tracing and timeout aspects across the business unit (bottom-up approach)
3. Conduct a quantitative study on the potential benefits of aspect-orientation for systems under current development, concerning overall quality and model size reduction. We are working on factoring fault-tolerance and security concerns out of production models and gathering data that could convince upper management to adopt the technology early in the development life cycle in the future (top-down approach)
4. Integrate the research body on aspect-oriented architecture and requirement engineering (early aspects) in order to be ready to describe aspect-oriented solutions at the specification level

7. CONCLUSIONS

We presented an industrial Model-Driven Engineering development environment and discussed Aspect-Oriented technologies in this context. We presented the Motorola *WEAVR*, its join point model for transition-centric state machines, and discussed its current degree of adoption. The particular join point model adopted is of essential importance. All the transition pointcut designators presented in the paper are expressed in terms of stable specification elements rather than implementation elements. This enables aspects to be defined in terms of a system specification, without requiring a complete knowledge of its implementation, giving such aspects additional robustness. This might reveal a differentiating advantage of Aspect-Oriented techniques applied at the modeling level over AOP technologies.

REFERENCES
[1] ITU, Z. 100: Specification and Description Language (SDL), International Telecommunication Union, 2000

[2] Baker, P., Weil, F., Liou, S., Model-Driven Engineering in a Large Industrial Context, In Proceedings 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2005), (Montego Bay, Jamaica, October 2005), LNCS 3844, pp. 100-109, Springer-Verlag, 2005

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

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

[5] Cottenier, T., van den Berg, A., Elrad, T. Model Weaving: Bridging the Divide between Translationists and Elaborationists. Workshop on Aspect-Oriented Modeling at the 9th International Conference on Model Driven Engineering Languages and Systems, Milan, Italy, 2006

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

[7] 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 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, Vancouver, Canada, 2004.

[8] Elrad, T., Aldawud, O., Bader, A.: Aspect Oriented Modeling - Bridging the Gap Between Design and Implementation. In Proceedings of the First International Conference on Generative Programming and Component Engineering (GPCE 2002 6-8, 2002), ACM Press, Pittsburgh, PA, USA, 2002

[9] Klein, J., Helouet, L, Jézéquel, J.M.: Semantic-based Weaving of Scenarios. In Proceedins of the 5th International Conference on Aspect-Oriented Software Development (AOSD' 06), ACM Press, Bonn,Germany, 2006

[10] Harel, David. Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming 8, 1987

[11] OMG. Semantics for a foundational subset for executable UML models - request for proposal. Request for Proposal ad/2005-04-02, Object Management Group, 2005.

[12] ETSI: Test and Test Conformance Notation, version 3, TTCN-3 Homepage, http://www.ttcn-3.org, 2005

[13] Zhang, J., Cottenier, T., van den Berg, A., Gray, J.: Aspect Composition and Interference in the Motorola Aspect-Oriented Model Weaver, Workshop on Aspect-Oriented Modeling at the 9th International Conference on Model Driven Engineering Languages and Systems, Italy, 2006

[14] Cottenier, T., van den Berg, A., Elrad, T.: Joinpoint Inference from Behavioral Specification to Implementation, submitted to ECOOP'07, available at www.iit.edu/~concur/weavr/papers/, 2006

[15] Telelogic. TAU G2 homepage, http://www.telelogic.com/products/tau/index.cfm, 2005