

Joinpoint Inference from Behavioral Specification to Implementation

Thomas Cottenier^{1,2}, Aswin van den Berg¹, Tzilla Elrad²

¹ Software and System Engineering Research Lab, Motorola Labs,
1303 E. Algonquin Rd, 60196 Schaumburg, IL, USA

² Concurrent Programming Research Group, Illinois Institute of Technology,
3100 S. Federal Street, 60696 Chicago, IL, USA

{thomas.cottenier, aswin.vandenberg}@motorola.com
{cotttho, elrad}@iit.edu

Abstract. Aspect-Oriented Programming languages allow pointcut descriptors to directly quantify over the implementation points of a system. Such pointcuts introduce strong mutual coupling between base modules and aspects and are problematic with respect to independent development. This paper introduces a new joinpoint selection mechanism based on state machine specifications. The interfaces of a system include a specification of the effects of method invocations on the state of the module instance. This specification is not defined with respect to potential aspects, but unambiguously describes the observable behavior of the module. We show how a smart joinpoint selection mechanism is able to infer points that might be located deep inside the implementation of a module, given pointcuts that are expressed entirely in terms of its specification elements. We present a tool, the Motorola *WEAVR*, which implements this technique in a Model-Driven Engineering environment.

Keywords: Aspect-Oriented Software Development, Modules and Interfaces, Model-Driven Engineering

1 Introduction

Since the inception of Aspect-Oriented Software Development in 1997, it has been known that Aspect-Oriented Programming (AOP) languages introduce strong coupling between base modules and Aspects. AOP languages allow pointcut descriptors to refer directly to the implementations of modules to capture joinpoints, points where Aspects inject behavior through advices. This ability is essential to unleash the real expressiveness of Aspects. Joinpoints that are exposed in the interface of a module can indeed be intercepted through other techniques than AOP. This practice is problematic with respect to modularity and independent development. Modules that are advised by aspects become hard to evolve independently. Small refactorings are susceptible to modify the way aspects interact with a module, breaking the semantics of Aspects.

There are two main research directions in addressing this problem. The first direction of research advocates restricting the expressiveness of Aspects by forfeiting the obliviousness of modules. Approaches such as Open Modules [1] or Crosscutting Interfaces [2] propose to move the pointcut descriptors from the aspect definition to the interfaces of modules. Aspects are only allowed to advice joinpoints that are explicitly published in the interface of a module. Also in this category are approaches that annotate the implementation of the base or organize the structure of modules in such a way that they are easily captured by specific aspects [3].

A second approach favors investigating alternative ways to reason about modularity in the presence of Aspects, while maintaining their full expressiveness [4].

This paper aims at spanning a new direction of research that addresses this problem in a different way. We propose to develop techniques that allow joinpoints located deep inside the implementation of a module to be *inferred* from pointcut descriptors that are entirely defined in terms of behavioral specifications. Traditional interfaces do not provide sufficient information about the runtime behavior of their components. Behavioral specifications such as state machines can describe the observable behavior of a module and the effects of method invocation on the state of the module instances in a way that is both intuitive and semantically well defined.

We show through some non-trivial examples that it is possible to define expressive Aspects without compromising the modularity of base modules. It requires the runtime behavior of modules to be specified using well defined semantics. Behavioral specifications do not need to be defined with respect to potential Aspects. They should appear naturally in the early stages of the software development lifecycle. This approach works particularly well in the context of Model-Driven Engineering, especially in domains where the use of statecharts [5] is well established, such as in the telecom industry. Yet, the approach could be generalized to programming languages using interface specifications such as Typestates [6] or predicates.

The paper is organized as follows. First, we introduce some of the Model-Driven Engineering practices in industry to develop large telecom infrastructure software. We distinguish between state machines specifications, used for system validation and state machines used for system verification and code generation. We present an Aspect-Oriented Modeling tool for the UML 2.0, the Motorola **WEAVR**. The tool performs weaving of Aspects at the modeling level and is currently being deployed in production, in the network infrastructure business unit.

Section 3 illustrates the approach through some examples that take on some of the concerns presented in the AO Challenge [7]. The AO Challenge consists of a series of fine grained concerns that are required to implement fault tolerance through transactional mechanisms. These concerns depend on each other and interact in subtle ways which makes their Aspect-Oriented implementation problematic. The solution presented is fully implemented in terms of the behavioral specification of the system but addresses the problem in a way that would be difficult, if not impossible to match using in an AOP language such as AspectJ.

Section 4 details the joinpoint selection mechanism that enables the **WEAVR** to infer implementation points of the system in terms of its specification. Some of the issues associated with the selection mechanism are discussed as well as its limitations.

Section 5 discusses related work and further research directions. Finally, Section 6 concludes this paper.

2 Aspect-Oriented and Model-Driven Software Engineering

2.1 Model-Driven Software Engineering

In this paper, Model-Driven Software Engineering is discussed in the context of fully automated code generation from precise behavioral models, a.k.a *translation* of models into executable artifacts [8]. The style of UML modeling used is also highly influenced by the ITU Specification and Description Languages (SDL) [9].

The telecom industry has a long tradition of Model-Driven Software Engineering. It pioneered the field starting in the 70's, with the SDL. 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 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 models as input and deliver highly optimized platform specific code, mostly in C and C++. Optimizing code generation has had an important effect of the system development process. The structure of the generated code is 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.

The UML 2.0 has adopted many of the language features of SDL, including the composite-structure architecture diagrams, support for transition-oriented state machines, and parts of the SDL action language semantics. This makes it possible to interpret UML 2.0 models as SDL-like specifications using a lightweight profile, and fully automatically generate executables. This is exactly what is performed by tools such as Telelogic TAU [10] and the Motorola Mousetrap code generator [11].

The next section presents the different phases of development lifecycle used in this context, and discusses the use of Aspect-Oriented Software Development.

2.2 System Design and Validation

The requirements of the system are captured using use cases expressed as Message Sequence Charts (MSC) or UML sequence diagrams. These use cases are translated into test cases definitions expressed in a notation such as the as TTCN (Testing and Test Control Notation) [13]. The test case definitions drive the design and implementation process at different levels of granularity. It is absolutely essential to be able to validate the system design and architecture as early as possible in the lifecycle. For large systems, validation is essentially performed through simulation and testing. There is therefore an important emphasis on the executability of system specifications. We consider executability in conformance to the test cases as an important property that needs to be maintained at all phases of the development process.

The architecture of the system is essentially defined using Composite-Structure Diagrams, specification State Machine Diagrams along with Class Diagrams.

2.2.1 Composite Structure Diagrams

Composite-Structure Diagrams define a hierarchical decomposition of the system. They are used early in the development process to attribute implementation responsibilities to teams of developers with respect to the modules that implement specific requirements of a system.

A Composite-Structure Diagram defines the internal run-time structure of an active class (a.k.a, a process definition), in terms of other active classes. These building blocks are referred to as parts. Composite structure diagrams express the communication within an 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 access server. An instance of a server is composed of two subcomponents, one dispatcher and request handlers. The number of request handlers is unbounded, and initially 0. The dispatcher is responsible for forwarding external requests to a request handler. Request handlers maintain sessions through a context id, (CID_t) and access resources delivered by services, identified by their resource id (RID_t)

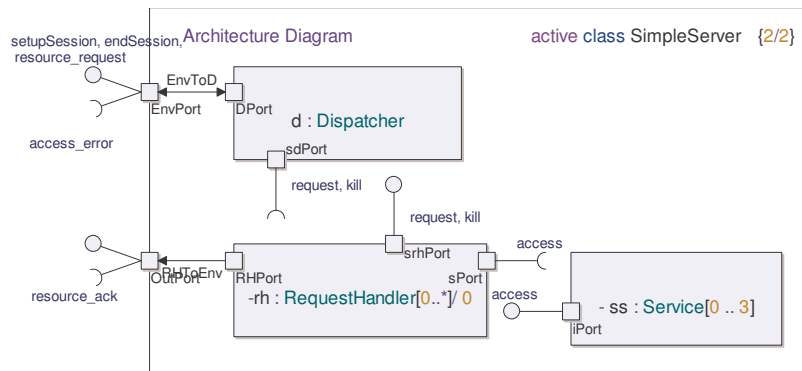


Fig. 1. A Composite-Structure Architecture diagram for a simple resource access server

2.2.2 State Machines Specification

The observable behavior of each part is specified using *state-oriented* state machine specifications. These state machines are not fully executable. They specify what the state transitions of the system are, their triggers and what output they produce. They do not define how this output is produced or what actions are executed along state transitions. Yet, such models can be simulated. Whenever the simulator is not fed with enough information about the system, the developer is prompted to enter the data that is required to complete the execution.

State machines specifications are defined directly in the interface of modules as a “provided” behavior. Figure 2 shows the interfaces for the request handler and the services, along with their provided state machine specifications. Note that the state machine of the Service interface is not directly executable. It does not specify how the decision is made whether a resource access occurred successfully or not. This decision needs to be entered by the user during the simulation. Figure 3 shows a trace that has been generated by the simulator during validation, in the case of a successful resource access. The traces generated by the model simulator can be systematically tested with respect the requirement use case, expressed as TTCN verdicts.

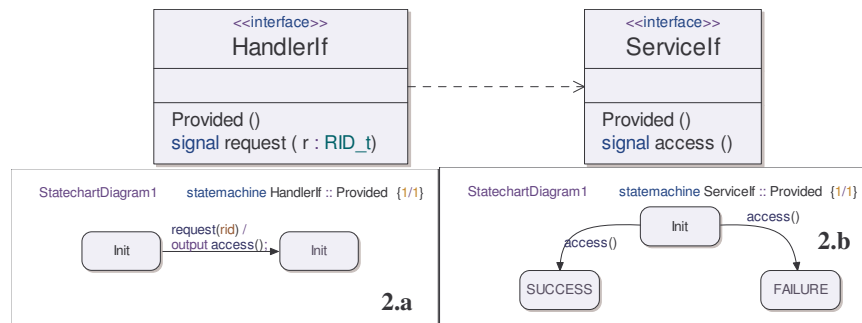


Fig. 2. Interfaces of request handlers and services and provided specification state machines

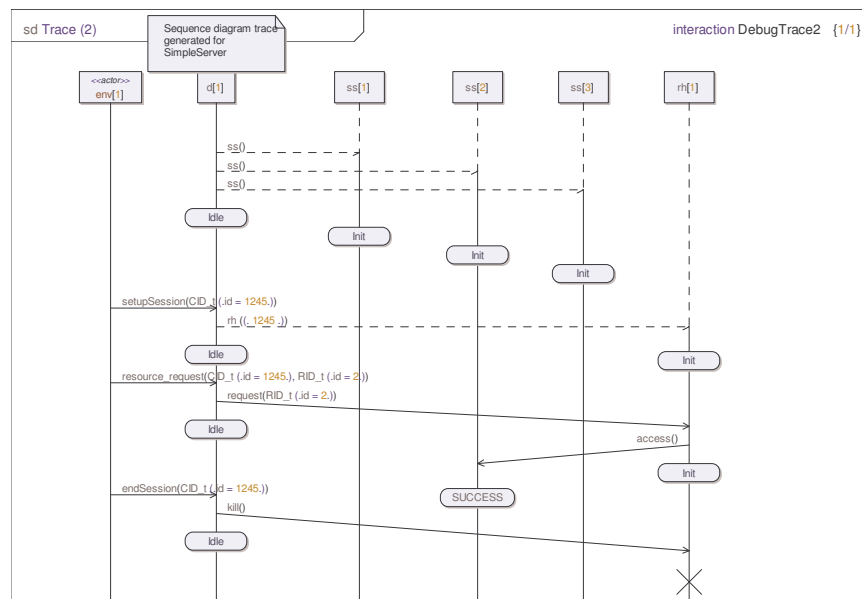


Fig. 3. Trace generated by the model simulator for the validation of the system specification. The development process is driven by artifact execution, testing and validation of the generated traces

2.2.3 State Machines Inheritance and Realization Mapping

State machines can inherit from other state machines directly, without resorting to hierarchical state machines. A specialized state machine may add features or change features of the original state machine. Features that may be added include states, transitions, variables and other entities that can be declared in a state machine.

We also support a particular type of realization relationship between state machine specifications. A state machine can be the realization of a more abstract state machine according to a *realization mapping*. Figure 4 shows two state machine specifications for two different types of resources, *Signals* 4.a and *Channels* 4.b. Both state machine specifications can be mapped to the *Service* state machine specification by mapping the states and the triggers of *Signal* and *Channel* to the states and signals of *Service*. This mapping defines a function from transitions of one state machine to transitions of another.

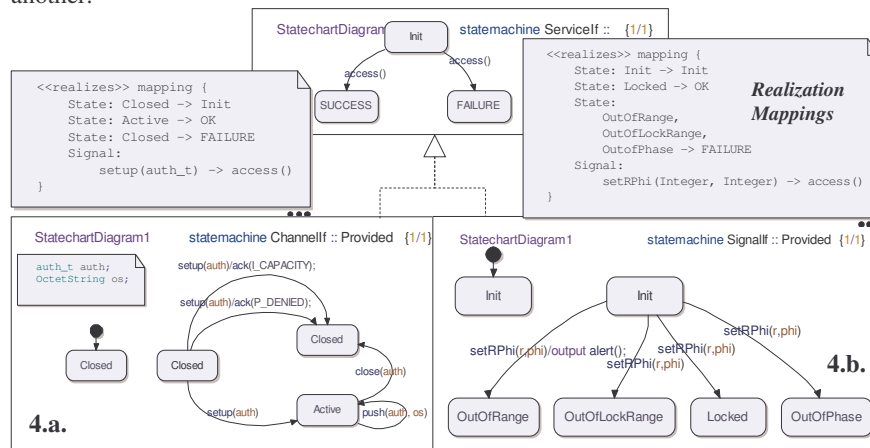


Fig. 4. Realization relationship between the state machine specifications of *Signal* and *Channel* and the state machine specification of *Service*

State machine realization relationships offer a powerful abstraction mechanism. A state machine specification provides a particular perspective of other state machine specifications. This perspective focuses on the properties that are relevant to the concern captured by the specification. Transitions that are not relevant can be omitted from the mapping. State machine specifications can realize multiple other specifications, each of those focusing on a different property.

State machine realization also favors reuse. The dependency of a module on the behavior of other modules can be expressed in terms of a state machine that captures the nature of this dependency. Third party components can then be seamlessly integrated by mapping their state machine specification to the specification of the interface. In the case of the example, the request handler can be specified in terms of a service it interacts with. The interaction of the request handler with the *Signal* and *Channel* resources can then be expressed in terms of the *Service* specification given a realization mapping, as illustrated in Figure 4.

2.3 System Implementation and Verification

The Class Diagram of Figure 5 depicts the implementation classes of the request handlers and the Channel and Signal resources. The inheritance relationship between the Service interface and the Channel and Signal interfaces is a realization relationship between their provided state machine specifications. The mappings are defined as stereotype tagged values associated with the generalization.

The state machines that define the behavior of the Channel and Signal resources inherit from the specification state machines of the ChannelIf and SignalIf interfaces. This inheritance relationship enforces that the transitions defined in the specifications are maintained at the level of the implementation. This enforcement is critical to the verification of the system, ensuring that the implementation of the system conforms to its specification.

The implementation of the Channel and Signal resources are defined using *transition-oriented* state machines, as illustrated in Figure 6. Transition-oriented state machines provide a better view of the control flow and the communication aspects of state transitions. 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 make the control flow explicit using decision actions, represented as diamonds.

The behavior of the request handler is defined in Figure 6.a. It starts up by instantiating some resources (i.0). Upon receiving a request, it accesses a resource that is indexed by a resource id (i.1), instantiates it if necessary (i.1.i), and accesses some of its own resources (i.2, i.3).

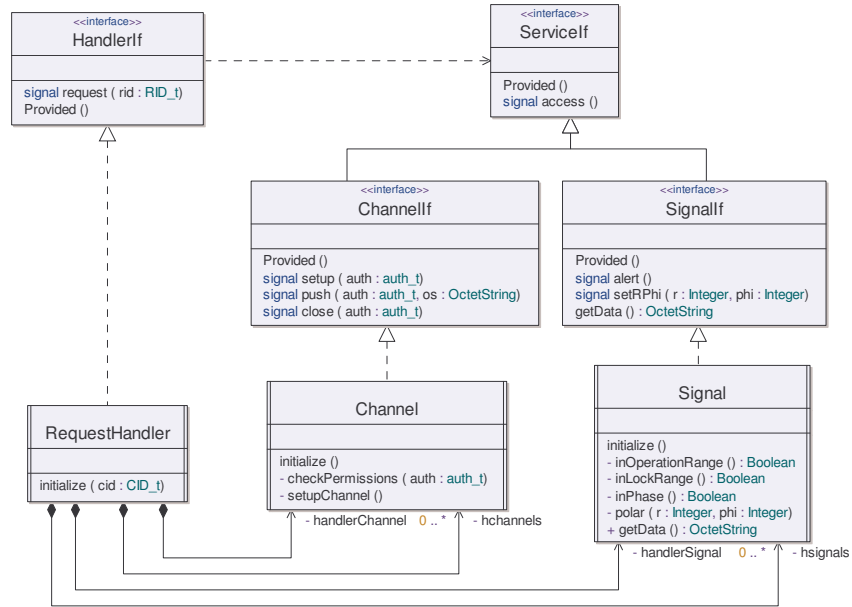


Fig. 5. System specification interfaces and implementation classes.

The behavior of the Signal and Channel resources is partially defined in Figure 6.b and 6.c. We only illustrate the behavior that is relevant to this discussion. Access to the resources triggers internal state transitions, whose outcome is controlled by a decision tree. These decisions are internal to the component behavior and are not exposed in its behavioral interface. Decision Actions have important semantics associated to them because they determine the outcome of a transition and directly affect the future behavior of the component.

The verification process is driven by the conformance to the system specifications (state machine specifications) and by the test case definitions. Each iteration step brings the model closer to full executability in the simulator. Some model checking techniques can also be used to verify properties on the implementation models.

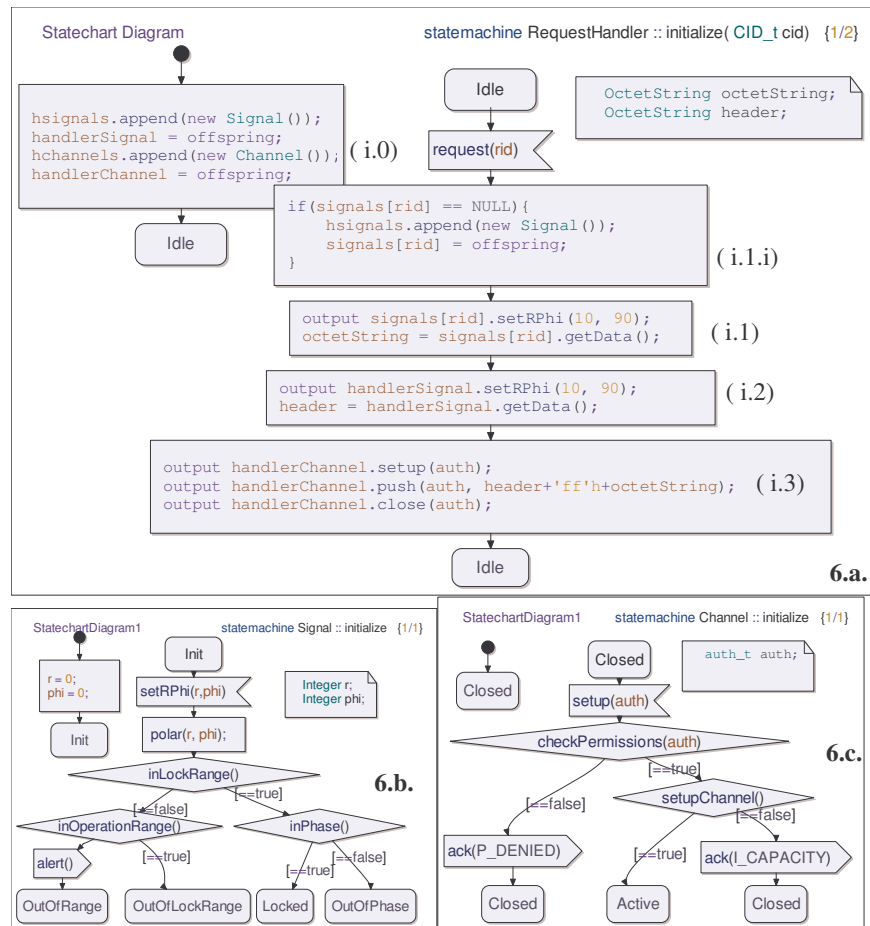


Fig. 6. Partial implementations of a Request Handler, a Signal and a Channel as transition oriented state machines

2.4 Model Translation and Code Generation

Once the system has been validated and thoroughly tested in the simulator, the models are translated to platform specific executables and tested in the field. The code generators used can be highly optimized for different target platforms. They handle the specificities and the configurations of the platforms and integrate additional concerns in the models that are not explicitly handled in the models. In particular, the specific libraries that are used do not have to be referred to in the models. These are handled by code generation. Examples are threading libraries, operating system API's or the use of a particular transport protocol. The code generation also performs different source code level optimizations that involve code motion and dead code elimination.

The structure of the generated code is typically pretty different from the structure of the models. Code generators do not map modeling concepts such as state machines into structures that are friendly to manual inspection. The code optimization process further destroys the structural and syntactic correspondence between the system models and the generated code and may modify the overall decomposition of the system to improve performance. Manual inspection and refinement of the generated code is not a viable practice.

2.5 Weaving Aspects into Models

The code generation does integrate crosscutting concerns such as tracing, logging or recurring platform specificities automatically with the base system. These concerns are activated and deployed through the configuration of the code generator. Yet, there are concerns, such as fault tolerance, security or timing constraints that are highly dependent on the application logic and cannot be handled in a systematic way by the code generator. The implementation of these concerns depends on the application.

Concerns that interact with the control flow of multiple state machines are hard to modularize using the abstractions of the UML. Alternative use cases or fault tolerance concerns tend to introduce new states and new decision actions in multiple states machines, across different development teams. State machine inheritance mechanisms do not support these kinds of refinements. Each development team needs therefore to re-implement this logic within multiple states machines. In practice, different teams would implement the same concern slightly differently, which leads to inconsistencies and an important replication of effort. It has been estimated that model size could be reduced by an average 40% if the replication of behavior within and across state machines could be avoided.

There is therefore a strong motivation to provide means to modularize application level concerns that interact with the control flow of state machines at multiple locations. For the reasons mentioned in the previous section, the use of an AOP language at the level of the generated code is not an option. Generated code lacks the necessary structure to apply aspects. Crosscutting concerns need to be fully coordinated with the base system at the level of the models. The Motorola **WEAVR** is a model transformation engine that provides enables Aspects to be defined at the modeling level and fully woven with the base models before code generation.

3 Motorola WEAVR: Weaving Aspects into Models

The Motorola *WEAVR* provides language constructs to capture Aspects in UML 2.0 and perform weaving of state machines before code generation. In a large industrial setting, the most important benefits from Aspect-Oriented programming are obtained when Aspects can be deployed across components of a system that are developed by independent development teams. These components communicate through well defined interfaces, to which state machine specifications are associated. The architecture of those components is also exposed to the extent it is unlikely to change during the implementation phases. The interfaces of subcomponents are also publicized, including their state machine specifications.

On the other hand, state machines that define the implementation of those components evolve quickly from iteration to another. In this context, it is critical that Aspects be defined exclusively in terms of the specification elements of the system. The verification process ensures that the implementation always conforms to these specifications - including that the realization mappings from component specifications to perspectives of those specifications are maintained.

This Section illustrates by example how joinpoints that might be located deep inside the implementation of a module can be inferred from state machine specifications. This capability is essential to apply the full expressiveness of Aspects in a context where information hiding is strictly enforced. We also illustrate the language constructs used in *WEAVR* and some of its more advanced features. The joinpoint selection mechanism itself is discussed in detail in Section 4.

The examples presented are simplified interpretations of some the AO Challenge [] concerns applied in a distributed setting. The concerns handled relate to exception handling and recovery, applied to the example presented in Section 2. The Aspects presented are Exception Handling, Recoverability, Atomicity and Two-Phase Commit. The concurrency control Aspects are not illustrated due to space limitations.

3.1 Exception Handling Aspect

The state machine specification of Figure 2.b indicates that a resource access can either succeed or fail. This specification corresponds to a particular mapping, or interpretation of what is considered a failure is in the context of the application. The specification does not constrain what the response of the resource is in case of failure. The Exception Handling Aspect enforces this response across all the components of the system. In case of exception handling, it is important to abort the execution at the early as possible, from the point at which the exception is detected. This example illustrates how the joinpoint selection mechanism is able to introduce advices within the implementation body of the module, at locations that are inferred from a pointcut definition that is completely defined in terms of specification elements.

Figure 7 defines a *WEAVR* Aspect. It simply states that whenever the system is going to end up in a state considered as a failure in the specification of Figure 2.b, an acknowledgement containing an error code should be sent back to sender of the access signal, without proceeding with the execution of the transition.

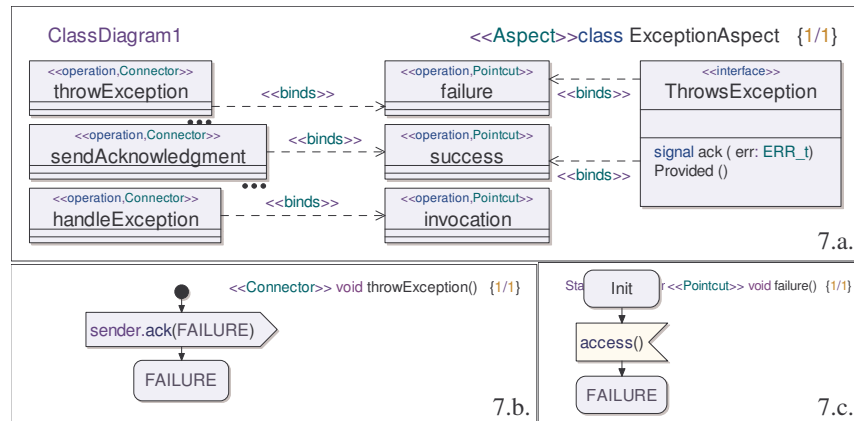


Fig. 7. The Exception Aspect binding diagram, the *failure* pointcut and the *throwException* Connector

The Aspect introduces a new interface, *ThrowsException*, which is bound to the object instance that contain joinpoints for the *success* and *failure* pointcuts. This interface declares the acknowledgement signal and the corresponding error codes. The Aspect also includes pointcuts and connectors. Connectors correspond to AOP language advices. Connectors are bound to advices through a dependency that is annotated with the *binds* stereotype. Connectors are instantiated for each joinpoint that matches a pointcut descriptor. The binding dependency specifies how arguments and parameters are passed from joinpoints to Connector instances.

A pointcut can either match a state machine transition, as illustrate in Figure 7.c, the invocation of a method, the setting of Timer or the output of a signal. All these events are considered as EventClass entities in UML 2.0.

The *failure* pointcut of Figure 7.c matches a selection of execution paths in the state machines. These selections are shown in Figure 8. The green marks delimit portions of the execution paths that match the pointcut of Figure 7.c. The marks that occur first in the execution path correspond to *before* joinpoints whereas the second marks correspond to the *after* joinpoints.

The joinpoint selection mechanism places the *before* mark at the first location in the execution paths for which the control flow determines that the state machine will transition to a state that matches the target state of the pointcut definition, from a state that matches its initial state, when triggered by a signal that matches the trigger of the pointcut definition. The matching mechanism can include multi-states, wildcards, and is resolved with respect of the realization mappings.

Figure 7.b illustrates a Connector that is connected to the execution statements that precede the selection (as shown by the start symbol), aborts the current execution (it does not call *proceed* keyword) and forces the state machine in a state that is already defined in the state machine implementation where it is instantiated, the failure state (*Failure* is declared in the pointcut it is bound to). The paths selections are deleted by the Connector and replaced by a direct transition to the failure state. In particular, the *inOperationalRange()* decision action is not executed anymore.

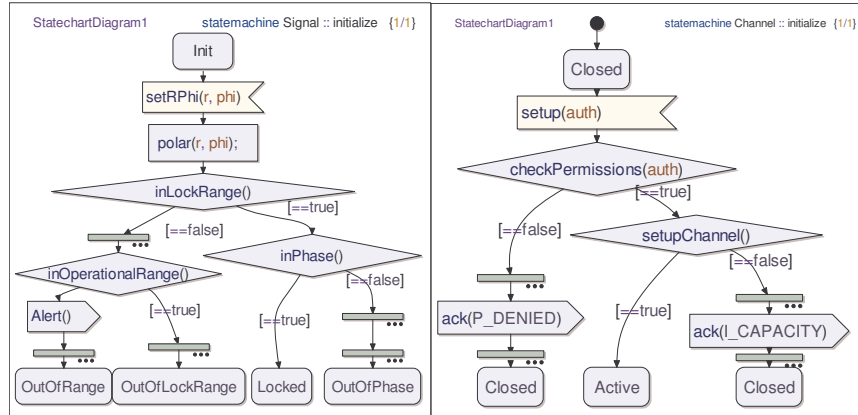


Fig. 8. The green marks delimit selections of executions paths in the state machine that match the pointcut of Figure 7.c.

Property. Transition pointcuts can capture joinpoints that might be located deep inside the implementation of state machines. These pointcuts are entirely defined in terms of the state machine specification entities. The joinpoint selection mechanism performs static control flow analysis to determine the earliest points in the implementation that match the pointcut definition, according to the mappings that are defined in the system.

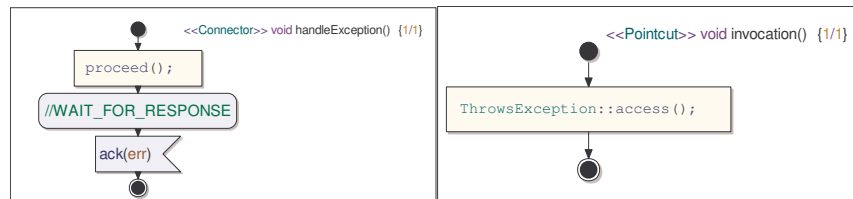


Fig. 9. The *invocation()* pointcut and the *handleException()* Connector.

Figure 9 illustrates the use of call/send signal pointcuts. The *invocation* pointcut selects send signal actions, remote procedure calls (calls to active classes) or simple method invocations (calls to passive classes) that match the *access* methods of the *Service* interface. The *handleException* Connector waits for the response provided by the resource and recovers the returned value in the *err* variable.

Property. Caller side pointcuts abstract away from the particular invocation mechanism. This is achieved through the use of implicit states followed by a trigger (as shown by the commented out state name). In the case where the pointcut matches a send signal action, the joinpoint is replaced by a remote procedure call.

3.2 Recovery Aspect

The Recovery Aspect builds on top of the ThrowsException Aspect. It illustrates the ability to introduce new transitions in state machines.

The recovery Aspect backs up the state of resource instances before an *access* invocation is processed. The backup advice (see the right side of Figure 10.d) is applied before any transition proceeds, from the Init state to any other state, as indicated by the wildcard used in the name of the target state of pointcut 10.b.

The aspect introduces a new transition in the state machines that contain joinpoints. The Init state referred to in the Connector is the same Init states as the one matched in the pointcut. The restore transition (on the left side of Figure 10.d) is also publicized in the provided interface of the Aspect.

Property. The effects of the connector on the behavior of the base models are specified in the interfaces of Aspects. This makes it possible to reason about Aspect/Base composition in terms of specification elements: the interface of the model, the interfaces of Aspects and their pointcuts. Access to the implementation of the state machines is not required.

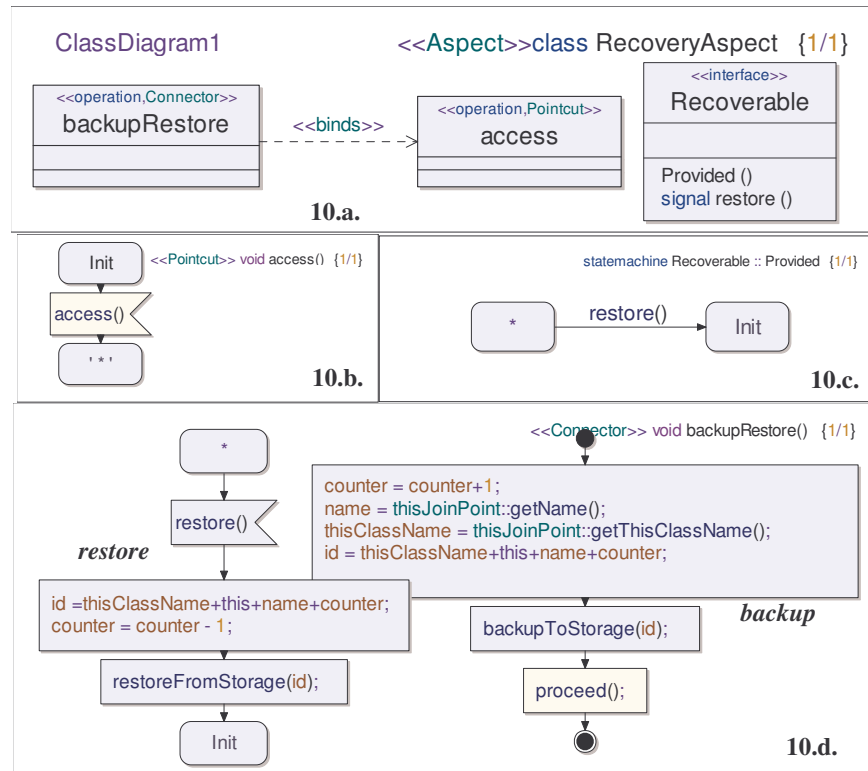


Fig. 10. The Recovery Aspect

3.3 Atomicity Aspect

The Atomicity Aspect ensures that all the transitions that access resources proceed in an atomic way. When a resource access fails, the aspect ensures that the resources previously accessed along that transition are restored to the state in which they were before the transition was triggered.

This Aspect depends on the deployment of the Exception Handling Aspect and the Recovery Aspect. A resource must notify the caller that an exception occurred and needs to provide the recovery capability. The Atomicity aspect illustrates how Aspects can introduce new states and new labels into state machines. Labels are represented as ellipses. They correspond to *goto* labels and are used to organize transitions into transition sections. State and label introduction is a powerful mechanism that allows aspects to build up complex static control flow structures.

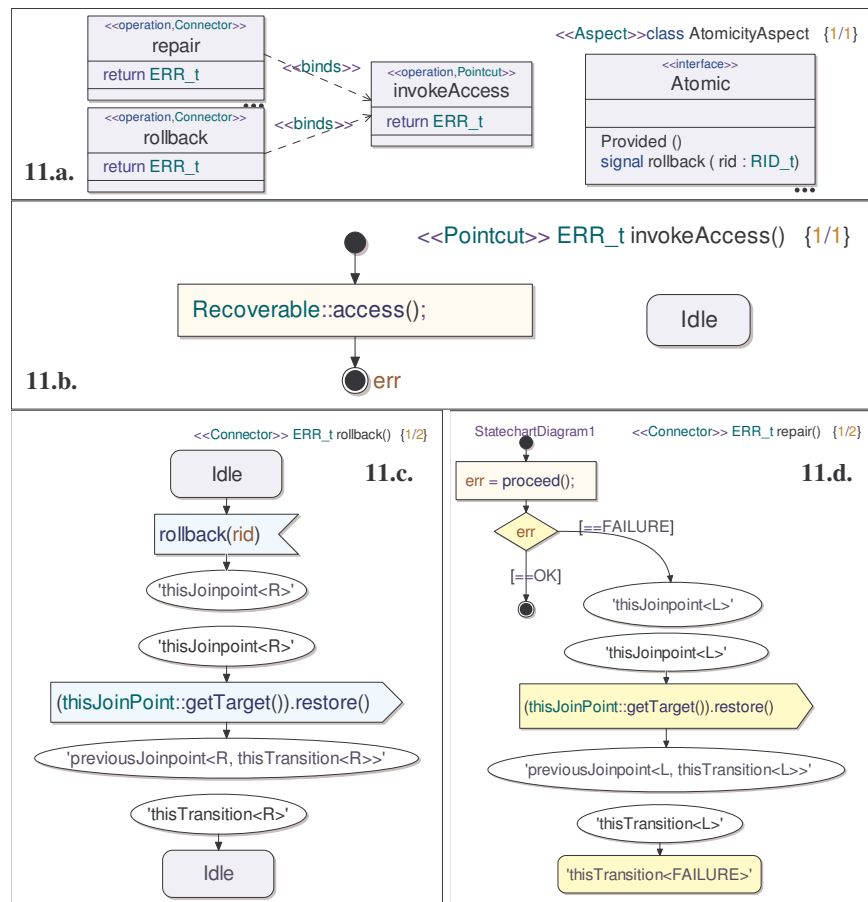


Fig. 11. The Atomicity Aspect

Figure 11 depicts the Atomicity Aspect. The pointcut of Figure 11.b intercepts the access to recoverable resources. The Connector of Figure 11.d builds up a static control flow structure that handles the failure of a resource access. A resource access failure is handled by recovering all the resources previously accessed along the current state transition execution. The Connector of Figure 11.c builds up the *rollback* transition. Figure 12 illustrates the result of weaving the Exception Handling, Recovery and Atomicity Aspects in the request handler model. In practice, woven models are not supposed to be manually inspected. The weaving really occurs on the model, and is not applied to its presentation entities.

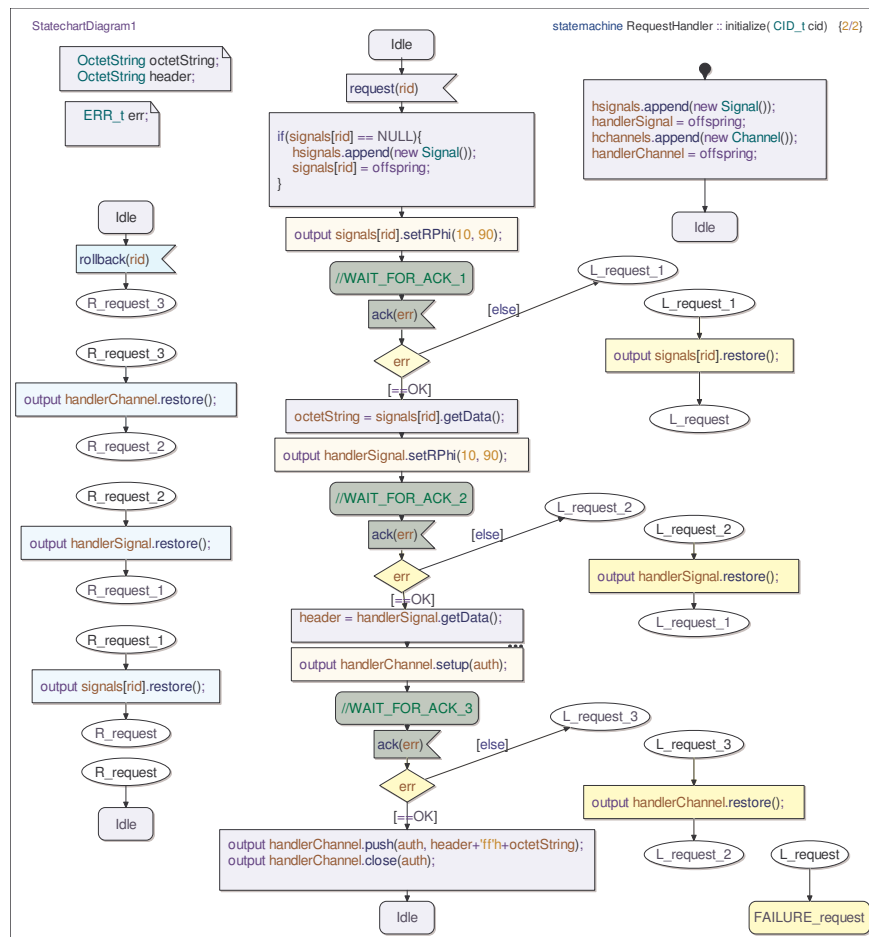


Fig. 12. Representation of the result of weaving the Exception Handling, Recovery and Atomicity Aspects in the request handler

A state or label introduction occurs when a Connector refers to a state or a label that is not declared in the pointcut descriptors it is bound to. When introducing states and labels into state machine, the scope of the introduction needs to be specified. States and labels can be introduced per State Machine, per Transition or per Joinpoint.

A state that is introduced per state machine, `thisStateMachine<state_name>`, is shared among all connector instances in the scope of a same state machine. A state that is introduced per transition, `thisTransition<state_name>` is shared by all connector instances bound to joinpoints of the same transition. A state that is introduced per Joinpoint, `thisJoinPoint<state_name>`, is unique to the connector instance bound to a specific joinpoint.

The joinpoints of a transition are partially ordered. In the case of per Joinpoint states, it is possible to refer to states that have been introduced by the connector bound to the previous joinpoint. This is accomplished using the `previousJoinPoint<state_name, default_state>` notation, which resolves to a default state in the case the previous joinpoint is not defined (the joinpoint is the first joinpoint occurring in a transition).

When referring to states or labels introduced by previous joinpoints the **WEAVR** can construct control structure statically, as shown in Figure 12 or dynamically. In the general case, this can only be achieved dynamically as discussed in Section 4.6.

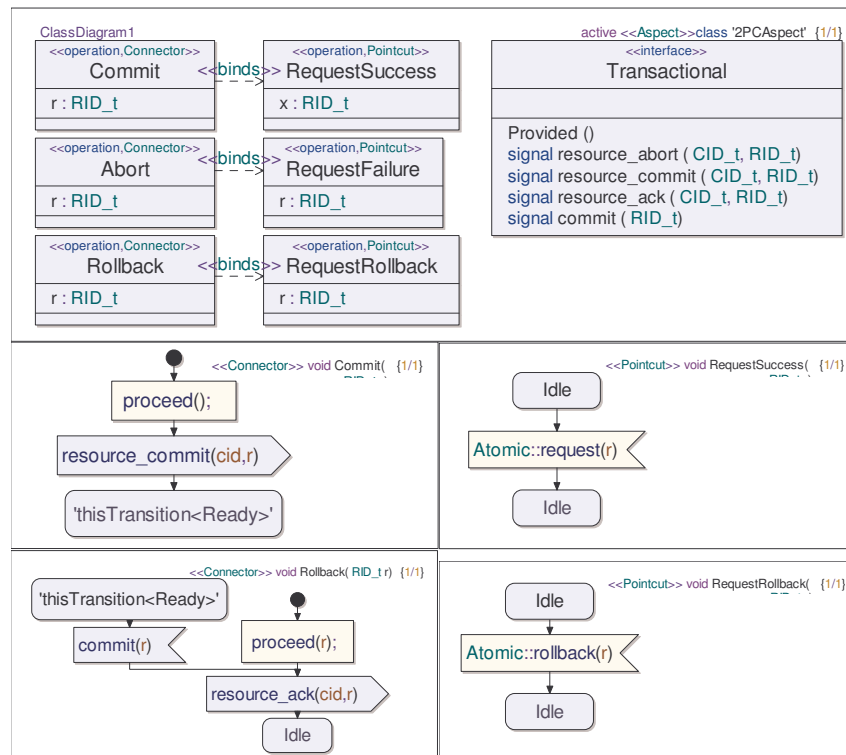


Fig. 13. Implementation of the Two-Phase Commit Aspect

Property. The control flow structures provided by state machines (states, decision actions and labels) make it possible to declare flow dependencies that span multiple advice instantiations. This allows aspects to introduce complex execution paths in the system. In the case of Atomicity, the Aspect is able to introduce the repair and rollback functionality directly, while AOP implementations would need to define runtime structures explicitly in the advice implementation and perform application monitoring.

3.4 Two-Phase Commit Aspect

The deployment of the Atomicity Aspect provides the necessary structure that allows a distributed transactional protocol such as Two-Phase Commit to be deployed transparently on the resource handlers. The Aspect for Two-Phase Commit is presented in Figure 13. The Aspect only needs to introduce a per Transition Ready state, which delimits the first phase of the protocol and declare the signals that drive the phases of the protocol and send acknowledgements to the transaction coordinator.

3.5 Discussion

The Two-Phase Commit problem presented in the previous section is a simplified representation of a real problem encountered in production models. 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.

Transparent deployment of fault tolerance behavior is not considered as a practice that is desirable through Aspect-Oriented Programming techniques [16]. We believe that Aspect-Oriented techniques can achieve a better separation of concerns at the level of state machines compared with code level techniques, especially in the domains of fault tolerance and concurrency. State machine specifications provide more information about the behavior of the modules, which allows aspects to get a better semantic grip on the module. The awareness of the base model with respect to the fault tolerance is implicitly captured by its state machine specification. This specification should not be declared with respect to potential aspects, but should appear naturally in the early design phases. Aspects can also introduce new state machine specifications and realization mappings in the system. Aspects can therefore explicitly declare the perspective of the system that is relevant to the behavior injected by the Aspect. Section 4 details some of the mechanisms of joinpoint inference used in the Motorola *WEAVR*.

4 Inference of Joinpoints from Specification to Implementation

The Motorola *WEAVR* performs weaving of state machine implementation in terms of state machine specifications. This operation requires the weaver to *infer* implementation joinpoints from pointcut designators expressed in terms of specification elements. This section discusses the joinpoint inference mechanism and outlines some of its properties.

4.1 Joinpoint Model and Selection

The weaver needs to perform a mapping between specifications that describe *what* state transitions are triggered and the logic that implements *how* these transitions are executed. A transition from a state S to a state T , triggered by an event i , ($S \xrightarrow{i} T$) corresponds to a tree of possible runtime *traces*, whose roots are *triggers* from state S and whose leaves are *next state actions* to state T . The nodes of this tree are points in the execution where the control flow affects the reachability of states. These are either decision actions or jumps statements to labels. This tree of possible traces maps to *execution paths* in the state machine implementation. The unit of weaving is a *selection* of those execution paths. A selection corresponds to a subtree of possible traces, whose leaves are next state actions.

The correspondence between the tree of traces and the graph of execution paths in the state machine can be obtained by performing a depth-first search on the state machines, starting from triggers that match the pointcut designator.

Definition. The pointcut designator $_{pct}(S \xrightarrow{i} *)$ refers to transitions from state S to any state, triggered by an event i . It matches a tree of traces whose root is a trigger i from state S . The corresponding execution paths are obtained by performing a DFS, starting from a root trigger i from state S . The leaves of the DFS tree are *all* reachable next state actions. This pointcut therefore always resolves to a single selection of execution paths within a state machine.

$$_{pct}(S \xrightarrow{i} *) \Rightarrow \{\bigcup path(S \xrightarrow{i} *)\} \quad (1)$$

Before advices are located right after the trigger. After advices are located right before the next state actions.

Definition. The pointcut designator $_{pct}(S \xrightarrow{i} T)$ refers to transitions from state S to state T , triggered by an event i . It defines the set of the largest subtrees of $_{pct}(S \xrightarrow{i} *)$, for which all leaves are next state actions to state T . This constraint can be expressed as follows.

$$_{pct}(S \xrightarrow{i} T) \Rightarrow \{\bigcup path(S \xrightarrow{i} *)\} \setminus \{\bigcup path(S \xrightarrow{i} NOT(T))\} \quad (2)$$

The pointcut defines a set of selections. In the case $\{\bigcup path(S \xrightarrow{i} NOT(T))\}$ is an empty set, it defines only one selection whose root is the trigger i on state S . Otherwise, it defines a set of trees whose roots are *decision actions* and whose leaves are next state

actions to T . Before advices are located right after the decision action. After advices are located before the next state actions.

Property. The root of a selection is never contained within a cycle in the state machine graph. A cycle is formed when a jump statement (a goto label statement) refers to one of its ancestors. The jump statement can only be part of the selection if all its reachable states match the pointcut end state. The ancestor is therefore also part of the selection.

4.2 Discussion

This matching method is very expressive because it can localize the important decision points in the execution of a state machine.

Decision points represent conditional statements that have a significant outcome on the state of the system, and on its future behavior. Conditional statements, in general, are hard to match directly, because they can be implemented in different ways, are prone to refactorings, and do not have an identifier. Signature-based matching of conditional statements is therefore not a good idea, and is rarely implemented in AOP languages.

Yet, decision points tend to be important crosscutting points. They are points where different use cases interact. As a result, aspects based on the procedural decomposition need to write complicated pointcuts that essentially attempt to detect those decisions points indirectly, which leads to brittle aspects. We consider the work on Stateful Aspects and Trace-Based Pointcuts [24][25] as being proposals that attempt to address this problem at the code level. The transition selection mechanism allows semantically significant decision points to be identified in terms of state machine states and triggers, which are stable elements in the system, and have an intuitive semantic meaning.

The technique has proven particularly useful in practice to perform logging and exception handling.

4.3 Quantification over States and Triggers

The definition of a pointcut selection (2) naturally supports quantification. The start state S and the end state T can refer to multistates, or sets of states. Multi states are supported by the UML 2.0 and heavily used in the SDL. Pointcuts can therefore quantify over states, which is easily extended to quantification over triggers and states.

Definition. A pointcut $(S \xrightarrow{i} *)$ whose start state S is a multistate defines multiple selections. These selections correspond to all the trace trees that start with a trigger that matches multistate the pair S, i . A selection is uniquely defined by the root of the corresponding trace tree.

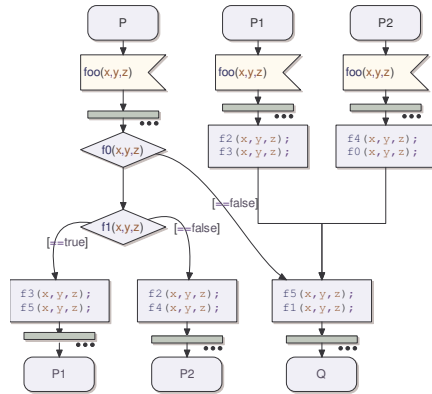


Figure 14.a Selections for
 $pct((P \text{ OR } P1 \text{ OR } P2) \xrightarrow{foo} *)$

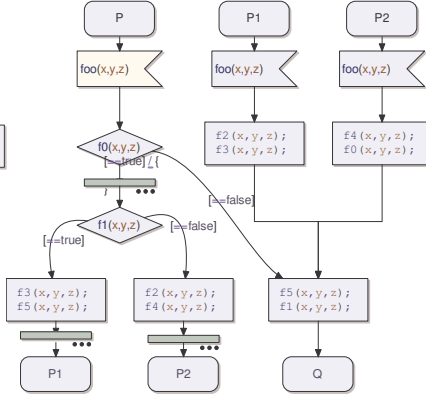


Figure 14.b. Selection for
 $pct(P \xrightarrow{foo} (P1 \text{ OR } P2))$

Figure 14.a represents 3 distinct selections: a selection from (P, foo) to $*$, a selection from $(P1, foo)$ to $*$ and a selection from $(P2, foo)$ to $*$.

Definition. A pointcut $(S \xrightarrow{i} T)$ whose start state S is a state and whose end state T is a multistate defines multiple selections as defined in (2).

Figure 14.b represents one selection: the selection from (P, foo) to $(P1 \text{ OR } P2)$.

4.4 Pointcut Composition

The definition of a pointcut selection (2) also supports pointcut composition through the logical AND and OR operators. The weaver simply performs the intersection or union of selections.

4.5 Isolation of Selections

The unit on which weaving is performed is a selection of execution paths. It is important to isolate the effects of the weaving between selections because different selections can refer to the same portions of execution paths within a state machine. Connectors introduce behavior that is instantiated in the context of each selection matched. A connector might introduce an after advice that contains reflective calls proper to the next state actions of the selection.

The weaver therefore performs state machine *expansion*. The state machine of Figure 14 contains behavior that is common to all transitions. This behavior is executed whenever a transition ends up in Q . An advice that affects this common behavior should be isolated from transitions that do not match the pointcuts it is bound to.

Figure 15 shows how the common behavior is replicated in order to introduce the after advice correctly. The weaving is performed on the selection, and the woven selection replaces the old one. Paths that did not match the pointcut are not affected.

The *WEAVR* performs some optimizations to avoid expanding the size of the model too much. In certain cases it might be better to insert a dynamic check around the advice execution rather than expanding the decision tree. This depends on the configuration of the application and the constraints on the model size.

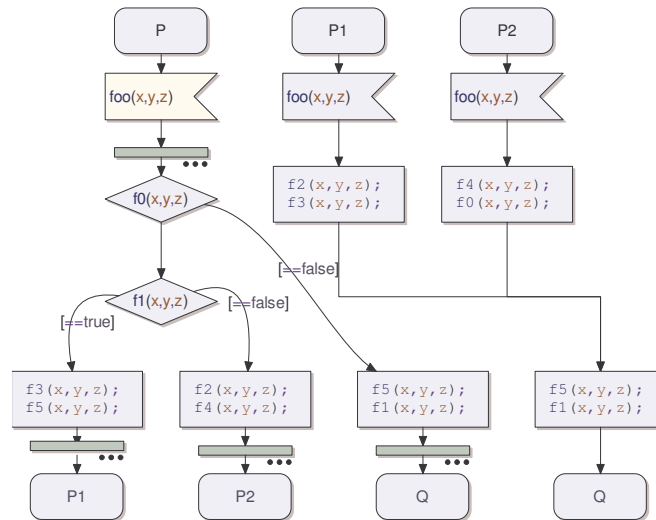


Figure 15. Isolation of the selection matching $pct(P \xrightarrow{foo}(P1 \text{ OR } P2 \text{ OR } Q))$. The behavior that is common to paths that do not match the pointcut has been replicated to apply the after advice correctly.

4.6 Ordering of Joinpoints and Control Structures

Connectors can introduce new states and new labels into state machines. States and labels can be introduced per State Machine, per Transition or per Joinpoint. In the case of per Joinpoint states or labels, it is possible to refer to states that have been introduced by the connector bound to the previous joinpoint in the transition. This technique enables Aspects to construct complex control flow structures. This is illustrated by the Atomicity Aspect of Figure 11. Figure 12 shows the effects of the Aspect on the implementation model of Figure 6.a.

The representation of the woven model shows the case where the control structure introduced by the Aspect can be determined statically. This is not possible in the general case where the transition decision tree includes forward edges, back edges or cross edges. For example, the implementation of Figure 6.a could contain a loop which invokes resource access multiple times. In these cases the *WEAVR* maintains a runtime jump table that is ordered according to joinpoint ancestor relationship in the execution trace of the transition.

5 Related Work

We discuss two categories of related work. First, we relate to other approaches to Aspect-Oriented Modeling (AOM). Second, we discuss how this work relates to Stateful aspects and pointcut composition mechanism that capture sequences of events in the trace of the system.

Most approaches to AOM focus on system architecture, design and validation rather than implementation, code generation and verification. In general, Aspect-Oriented Modeling approaches can be classified in two main categories [14].

Approaches that emphasize model weaving see AOM as a model transformation technique. Aspects enable crosscutting concerns appearing in models to be modularized and abstracted out. Platform-specific models and code are fully or semi automatically generated. It is therefore beneficial to weave aspects directly at the model level rather than the code level. Typically, these approaches are not interested in generating code-level aspects from models; weaving is fully supported at the modeling level. Examples of model weavers are C-SAW [15] and MDA Query-View-Transformation (QVT) [16] based approaches such as the ATL ModelWeaver [19].

Other approaches propose modeling notations to represent code-level crosscutting concerns at the modeling level. Tool support focuses on analysis tools, code skeleton generators that target AOP languages and round-trip engineering tools to keep model-level aspects and code-level aspects synchronized. The generated code needs to be inspected and manually refined. Behavioral model weaving is therefore not an option; aspects have to maintain their modular structure throughout the development process. Examples of AOM approaches that fall in this category are Theme UML [20] or Jacobson's use case based approach [21]. We think these approaches are more suitable for requirement analysis and early design, rather than Model-Driven Engineering.

Related work concerning the application of AOSD to Harel Statecharts includes the Aspect-Oriented Statechart Framework (AOSF) [22][23]. The AOSF targets early design models and validation using state-oriented state machines, as opposed to system implementation and verification. 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. The joinpoint model proposed is not based on the states of the system, but on individual transition, that are identified by their trigger. Rules specify which transitions trigger crosscutting transitions, in an orthogonal region.

The work on stateful aspects [24][25] 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. The history of the system is recognized as an important property that should be captured by pointcut designators. Stateful aspects allow important state transitions to be identified through the recognition of a pattern of successive events. We consider the need for such pointcut designators as a symptom that the system implements reactive behavior. As such, the system would be better decomposed using the natural decomposition for reactive systems, statecharts. Stateful aspect pointcuts require explicit knowledge of the characteristic actions of a particular transition. Using state machines directly, the specific sequences of actions

that characterize a transition are abstracted out. This knowledge is captured by the states of the system.

Finally, the path selection mechanism bears resemblances to the predictive pcf flow [26][27]. Again, pcf flow is based on a method-based joinpoint model and is expressed in terms of the implementation of the path, rather than in terms of the properties of the path, as it is the case with state transitions.

6 Conclusions

We demonstrate a technique that allow joinpoints located deep inside the implementation of a module to be *inferred* from pointcut descriptors that are entirely defined in terms of behavioral specifications. Traditional interfaces do not provide sufficient information about the runtime behavior of their components. This forces Aspect-Oriented Programming language to refer directly to the implementation of modules rather than their specification.

We show through some examples that it is possible to define expressive Aspects without compromising the modularity of base modules, by taking advantage of the abstraction provided by state machines specifications. We believe that Aspect-Oriented techniques can achieve a better separation of concerns at the level of state machines compared with code level techniques, especially in the domains of fault tolerance and concurrency. State machine specifications provide more information about the behavior of the modules, which allows aspects to get a better semantic grip on the module. The awareness of the base model with respect to the fault tolerance is implicitly captured by its state machine specification.

Behavioral specifications do not need to be defined with respect to potential Aspects. They should appear naturally in the early stages of the software development lifecycle. This approach works particularly well in the context of Model-Driven Engineering because there is a direct mapping from the system specification to its implementation. The particular technique discussed is by no mean the only way implementation joinpoints can be inferred from behavioral specifications. Also, the approach could be generalized to programming languages using interface specifications such as Typestates or predicates.

The inference method discussed in the paper is very expressive because it can localize important decision points in the implementation of a state machine. Decision points represent conditional statements that have a significant outcome on the state of the system, and on its future behavior. Conditional statements, in general, are hard to match directly, because they can be implemented in different ways, are prone to refactorings, and do not have an identifier. Signature-based matching of conditional statements is therefore not a good practice, and is rarely implemented in AOP languages.

Yet, decision points tend to be important crosscutting points. They are points where different use cases interact. As a result, aspects based on the procedural decomposition need to write complicated pointcuts that essentially attempt to detect those decisions points indirectly, which leads to brittle aspects. We consider the work on Stateful Aspects as being proposals that attempt to address this problem at the code

level. The transition selection mechanism allows semantically significant decision points to be identified in terms of state machine states and triggers, which are stable elements in the system, and have an intuitive semantic meaning.

We also introduce an Aspect-Oriented Modeling tool that implements the joinpoint selection mechanism in UML 2.0, the Motorola **WEAVR**. The tool performs weaving of Aspects at the modeling level and is currently being deployed at in production at Motorola, in the network infrastructure business unit.

References

1. Aldrich, J. Open Modules: Modular Reasoning about Advice. In Proceedings of the European Conference on Object-Oriented Programming, pp 144-268, Glasgow, Scotland, 2005.
2. Griswold, W.G., Shonle, M., Sullivan, K., Song, Tewari, N., Cai, Y., Rajan, H.: Modular Software Design with Crosscutting Interfaces. *IEEE Software* 23:1 pp. 51–60, 2006.
3. Gybels, K., Brichau, J.: Arranging Language Features for More Robust Pattern-Based Crosscuts. In proceedings of the International Conference on Aspect-Oriented Software Development, pp 60–69, Boston, Massachusetts, USA, 2003.
4. Kiczales, G., Mezini, M.: Aspect-Oriented Programming and Modular Reasoning. In proceedings of the International Conference on Software Engineering, pp 49–58. 2005.
5. Harel, David. 1987. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8 231-274.
6. DeLine R., Fähndrich, M., Typestates for Objects, In Proceedings of the European Conference on Object-Oriented Programming, pp 144-268, Oslo, Norway, 2004.
7. Kienzle, J., Gelineau, S, AO challenge - implementing the ACID properties for transactional objects, In proceedings of the International Conference on Aspect-Oriented Software Development, Bohn, Germany, 2006.
8. Mellor, S.J., Balcer, M.J. Executable UML: A Foundation for Model Driven Architecture, Addison-Wesley, 2002.
9. ITU, Z. 100: Specification and Description Language (SDL), International Telecommunication Union , 2000.
10. Telelogic. TAU G2 homepage. <http://www.telelogic.com/products/tau/index.cfm>, 2005.
11. 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
12. ETSI: Test and Test Conformance Notation, version 3, TTCN-3 Homepage, <http://www.ttcn-3.org>, 2005
13. Cottenier, T., van den Berg, A., Elrad, T. Modeling Aspect-Oriented Compositions. Proceedings of the Satellite Events at the 8th International Conference on Model Driven Engineering Languages and Systems, Montego Bay, Jamaica, LNCS 3844, pp. 100-109, Springer-Verlag, 2005.
14. 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.
15. 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.
16. Bast, W., Kleppe A., Warmer, J. MDA Explained: The Model Driven Architecture: Practice and Promise, Addison-Wesley, 2003

17. Frankel, D.S. Model Driven Architecture: Applying MDA to Enterprise Computing, John Wiley & Sons, 2003.
18. OMG. MOF QVT Final Adopted Specification, Specification ptc/05-11-01, Object Management Group, 2005.
19. 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.
20. Clarke, S., Baniassad, E. Aspect-Oriented Analysis and Design. The Theme Approach Addison-Wesley, 2005.
21. Jacobson, I. Ng, P-W., Aspect-Oriented Software Development with Use Cases: Addison-Wesley, 2004.
22. Elrad T., Aldawud O., Bader A.. Aspect-oriented Modeling - Bridging the Gap Between Design and Implementation. Proceedings of the First ACM SIGPLAN/SIGSOFT International Conference on Generative Programming and Component Engineering, Pittsburgh, PA., , pp. 189-202. 2002
23. Mahoney, M., Bader, A., Aldawud, O., Elrad, T., Using Aspects to Abstract and Modularize Statecharts. The 5th International Workshop on Aspect-Oriented Modeling, in Conjunction with UML 2004, 2004.
24. Vanderperren, W., Suvee, D., Cibrán, M. A., De Fraine, B. Stateful Aspects in JAsCo, Software Composition Workshop (LNCS), ETAPS 2005, Edinburgh, Scotland, 2005
25. R. Douence, P. Fradet, M. Sudholt. Composition, Reuse and Interaction Analysis of Stateful Aspects, In proceedings of the 3rd International Conference on Aspect-Oriented Software Development, Lancaster, UK, 2004
26. Gregor Kiczales. Keynote talk at the 2d International Conference on Aspect-Oriented Software Development, 2003
27. Ostermann, K., Mezini, M., Bockisch, C.: Expressive Pointcuts for Increased Modularity. In Proceedings of the 19th European Conference on Object-Oriented Programming, Glasgow, UK, 2005