

**MOTOROLA *WEAVR*: AN ADD-IN FOR
ASPECT-ORIENTED MODELING
IN TELELOGIC TAU G2**

by

THOMAS COTTENIER^{1,2}, ASWIN VAN DEN BERG¹, TZILLA ELRAD²

¹MOTOROLA LABS

²ILLINOIS INSTITUTE OF TECHNOLOGY

Prepared for the 2006 Telelogic Americas User Group Conference

Abstract

MOTOROLA **WEAVR**: AN ADD-IN FOR ASPECT-ORIENTED MODELING IN TELELOGIC TAU G2

The behavioral specifications of extra-functional concerns such as security, fault-tolerance or exception-handling tend to be hard to modularize in the UML. Their implementation is hard to encapsulate in separate state machine diagrams because their functionality tends to interact with the base behavior of the system at multiple locations. Aspect-Oriented Modeling (AOM) is a model transformation technique that focuses on the encapsulation of crosscutting concerns. AOM can be used to refine models by semi-automatically injecting more specific behavior into models that are defined at a lower-level of granular. This paper presents an add-in to Telelogic TAU G2 for Aspect-Oriented refinement of UML state machines. The add-in provides support for the modularization of crosscutting concerns throughout the system modeling, verification and code-generation phases.

A profile for the specification of aspects is proposed and our aspect weaving engine is presented. The approach is illustrated through a refinement of the “Ping Pong” example from the TAU G2 distribution.

Author Biography

THOMAS COTTENIER

Thomas Cottenier is a PhD student in the Computer Science department of the Illinois Institute of Technology and works as a researcher at the Software and System Engineering Lab at Motorola. Thomas holds an Electrical Engineering degree from the Université Libre de Bruxelles, Belgium and a MS in Computer Engineering from the Illinois Institute of Technology. Thomas' interests include Aspect-Oriented Software Development, Service-Oriented Architectures and Model-Driven Engineering. For the past couple years, Thomas has been working on platform for defining adaptive and decentralized service compositions based on the Globus Toolkit. At Motorola, Thomas has been developing an industrial strength Model-Driven Development framework that combines Aspect-Oriented Modeling and Automated Code Generation technologies.

Software Systems Engineering Research, Motorola Labs

1300 E Algonquin Road, 60196 Schaumburg, IL, USA

(847) 538 37 39

thomas.cottenier@motorola.com

INTRODUCTION

The primary decomposition mechanism for UML state machines is hierarchical. This decomposition does not allow the behavior of some concerns in a system to be encapsulated into separated model elements. Especially, when extra-functional requirements such as tracing, security or exception management are mapped to the system components, their functionality tends to affect the system at multiple places of the models. The implementation of these requirements is hard to modularize because their functionality tends to capture alternative execution paths of the systems, such as exceptional use cases. Hence, they tend to affect the control flow of multiple state machines within and across the components and classes of the system.

Their deployment is therefore tedious and requires coordination between geographically distributed development teams causing consistency problems and duplication of effort.

The Aspect-Oriented Software Development (AOSD) [1][2] community identifies those concerns as being crosscutting concerns. Their implementation cannot be well encapsulated within the modularity units of the language, because they follow different composition rules. During the initial design phases, these concerns cannot be mapped from requirement to design in isolation, and end up tangled with model elements that implement other requirements. In the context of UML state machines diagrams, these concerns tend to introduce new states and inject multiple decision actions into state machines, causing a multiplication of possible transitions paths.

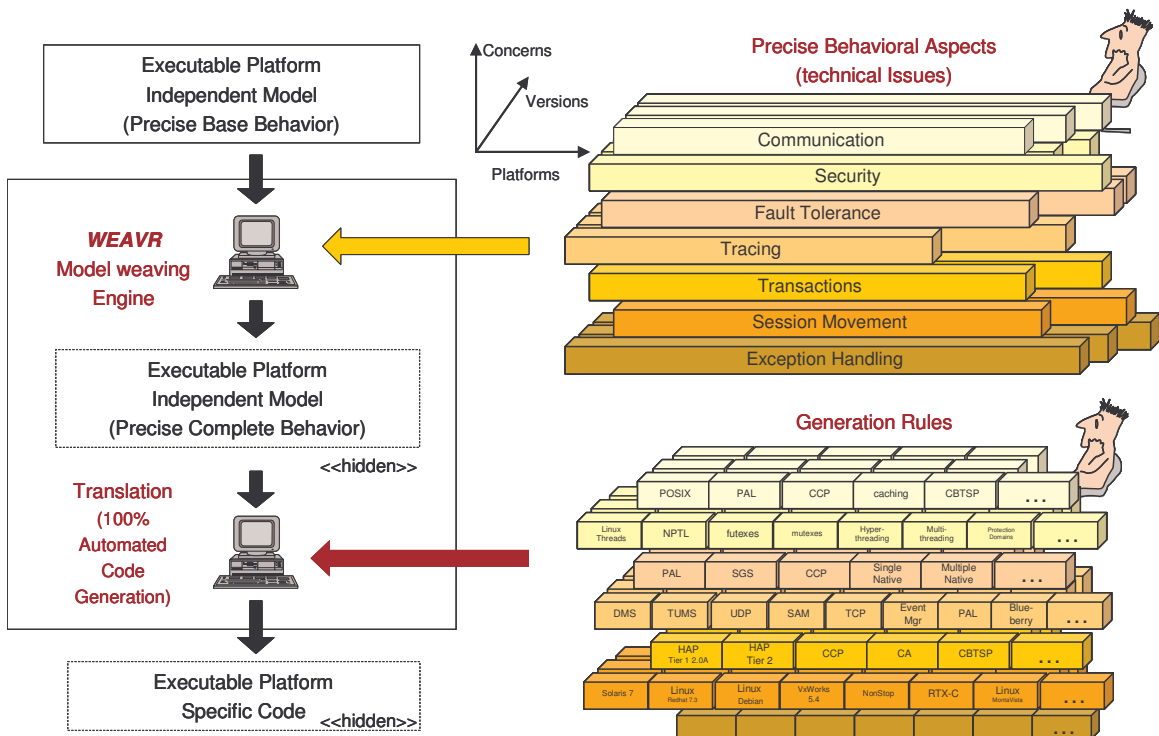


Figure 1. Motorola Model-Driven Architecture stack

AOSD introduces a new unit of modularity, called an Aspect, which encapsulates the implementation of a crosscutting concern. Aspect-Oriented languages require a special type of compiler, called a weaver, whose role is to coordinate Aspects with the core system implementation.

Aspect-Oriented Modeling [3] (AOM) is a branch of Aspect-Oriented Software Development that focuses on the encapsulation of concerns at the system design and modeling level. AOM techniques provide notations for specifying crosscutting concerns at the modeling level.

We view Aspect-Oriented Modeling as a particular form of Model Transformation, in the sense of the OMG Model-Driven Architecture (MDA) [4] initiative.

Aspect-Oriented Modeling techniques allow us to separate the implementation of more specific behavior from models that capture the domain problem at a lower level of granularity. A weaving engine working at the modeling level can then automatically transform these models as to coordinate the aspect models with the domain problem models.

Model weaving therefore complements code generation. Advanced code generation techniques enable Platform Independent Models to be automatically transformed into executable code, given an extensive set of platform-specific generation rules. Yet, this model must include the complete behavior of the application. Inevitably, technical considerations such as authentication, caching or logging must also be modeled at this level in order to produce a complete application.

Model weaving provides a more flexible solution to the integration of these concerns than general model transformation:

1. Aspects provide a modular way to express a model transformation with respect to a particular concern.
2. Aspects enable model transformations to be expressed in terms of the model entities to be transformed, as opposed to Query-View-Transformation [6] queries which are expressed in terms of metamodel entities.

Figure 1 sketches the Model-Driven Engineering stack deployed in production at Motorola. The base functionality and behavior of the application is captured in Platform Independent Models (PIM). The PIM models are executable in the simulation environment. This enables the base functionality to be verified and tested in isolation. The PIM is thereafter iteratively refined by weaving aspect models encapsulating specific concerns into the PIM. The resulting PIM is hidden from the developer because weaving introduces generated elements into the model, which do not have a user friendly representation. The woven PIM is also fully executable, which enables the integration of the model with specific aspects to be tested separately. The woven PIM is then automatically mapped to a PSM from which code is generated.

The weaving mechanism needs to address the coordination of crosscutting behavior.

This work therefore focuses on the encapsulation of crosscutting behavior in UML state machines. A profile for the specification of aspects is proposed, and our weaving engine is discussed. Our add-in aims at enforcing the separation of concerns between domain problem and extra-functional requirements throughout the system lifecycle, including model verification, system trace visualization and code generation. Therefore, we maintain one important invariant: system developers should never have to see a woven model. The

system is defined by its domain problem models and its aspect models. Developers should never see a model artifact that is generated by our weaving engine. Yet, all activities involving code generation, such as model verification, need to operate on generated models. Our add-in therefore performs a series of transformations that allow the verification to proceed on woven models, while the models that are visualized during verification are the original and the aspect models. Our add-in also filters out the sequence diagrams that are generated by the model verifier in order to separate the action occurrences of the original model from the action occurrences introduced by the Aspects.

This paper is organized as follow. Section 2 presents a profile for the specification of crosscutting behavior in state machines. Its use is illustrated through a tracing aspect which customizes the tracing functionality of the model verifier. Section 3 discusses visualization of crosscutting concerns, and illustrates the AOM views that are provided by the add-in. Section 4 discusses the simulation of Aspect-Oriented models and the decomposition of the generated sequence diagrams. Section 5 presents how the aspect models are coordinated with the original models before code generation. Finally, Section 6 concludes this paper.

ASPECT-ORIENTED MODELING PROFILE

Basic Constructs

Two essential language constructs are required for Aspect-Oriented Modeling. First, we need to specify the “what”; the behavior of the crosscutting concern. As our aspects are intended to be woven into UML state machines, this behavior specified using statechart diagrams. The “what” of the aspect is encapsulated into a special kind of operation, which is extended by a “Connector” stereotype. Connectors correspond to the “Advice” construct in Aspect-Oriented Programming (AOP) languages such as AspectJ [5].

Second, we need to specify the “where”; the locations in the domain model where the crosscutting behavior needs to be injected. These locations are expressed as a query over the domain model state machines. Two types of queries are supported: queries on state machine actions and queries over state machine transitions. We provide a graphical notation for these queries using state machine diagrams. Queries are encapsulated in a special kind of operation, denoted by the “Pointcut” stereotype.

Pointcuts and Connectors are encapsulated in a special type of class, extended by the “Aspect” stereotype.

Aspects can contain multiple Pointcuts and multiple Connectors. Aspects therefore contain a Binding diagram which defines which connectors are bound to which Pointcuts. Those bindings are defined using dependency relationships from Connectors to Pointcuts which are stereotyped by the “Bind” stereotype.

Figure 2 illustrates the binding diagram for a tracing aspect. This tracing aspect captures 4 kinds of events:

- Calls to a method
- Executions of an operation
- Output actions (the sending of a Signal)
- Triggered Transitions from one state to another

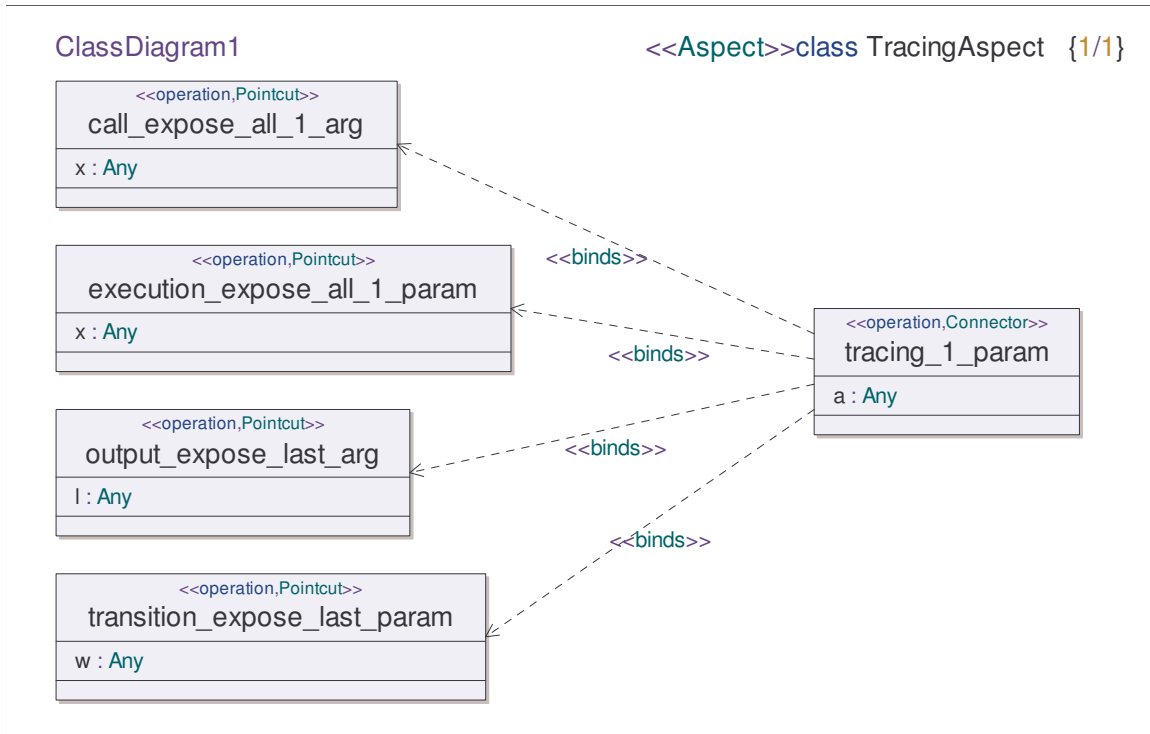


Figure 2. Binding diagram for a simple tracing aspect

Pointcut Diagram

A Pointcut is a query over the domain model elements. Two different types of pointcuts are supported: Action Pointcuts and Transition Pointcuts.

Pointcuts have three different features: an expression, a state machine diagram and an interface.

A Pointcut expression is a regular expression over the signature and the type of the parameters of the actions or the transitions that are captured by the Pointcut. The Pointcut expression is defined as an operation that is defined in the scope of the Pointcut and is stereotyped by the “Expression” stereotype. The semantics of the predefined ‘Any’ type have been overloaded to serve as a wildcard that can represent any type.

The Pointcut state machine diagram specifies what type of actions or transitions are matched by the Pointcut. Pointcut state machines need to adhere to a restricted version of the UML state machine metamodel. The metamodels for action and transition Pointcut state machine diagrams are out of the scope of this paper. Figure 3 and Figure 4 show examples of Pointcut state machine diagrams for action and transition Pointcuts.

Action Pointcuts can match method calls (CallExpr ExpressionAction), constructor calls (CreateExpr ExpressionAction), timer set or reset actions (TimerSetAction) or signal send actions (OutputAction).

Transition Pointcuts can match state machine initializations and method executions (StartTransition) or state machine transitions and terminations (TriggeredTransition).

The complete semantics of transition queries and transition selection are out of the scope of this paper.

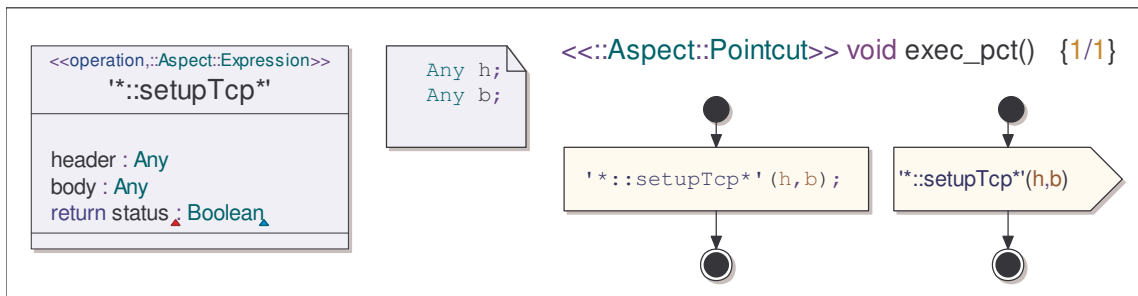


Figure 3. Action Pointcut Diagrams

Pointcuts have an interface that specifies which of the arguments of the matched action or which of the parameters of the matched transition are exposed by the Pointcuts. Exposed arguments or parameters are visible by the Connectors bound to the Pointcut, and their values can therefore be modified by actions occurring in the Connector.

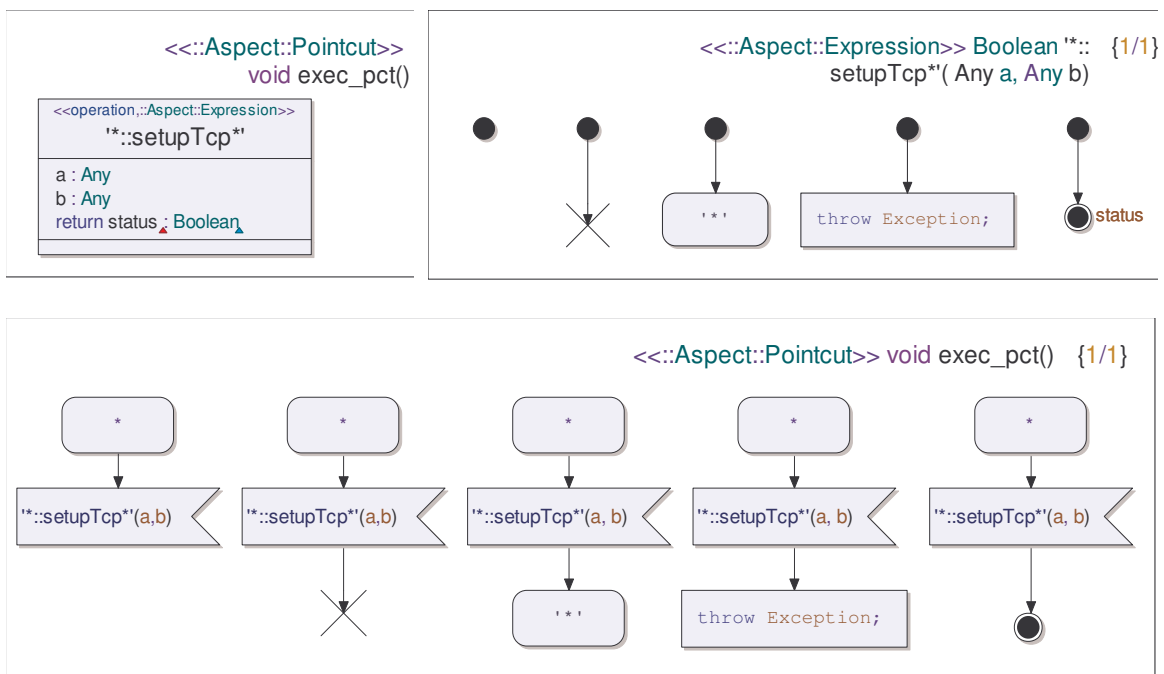


Figure 4. Transition Pointcut Diagrams

For example, the Pointcut of Figure 5 matches all calls for which the called operation takes one parameter of any type, and returns a parameter of any type. The Pointcut only exposes the call argument to Connectors.

The Pointcut of Figure 6 matches all triggered transitions, from some state to any other state, for which the trigger matches the signature of the expression. The Pointcut only exposes the last parameter of the trigger.

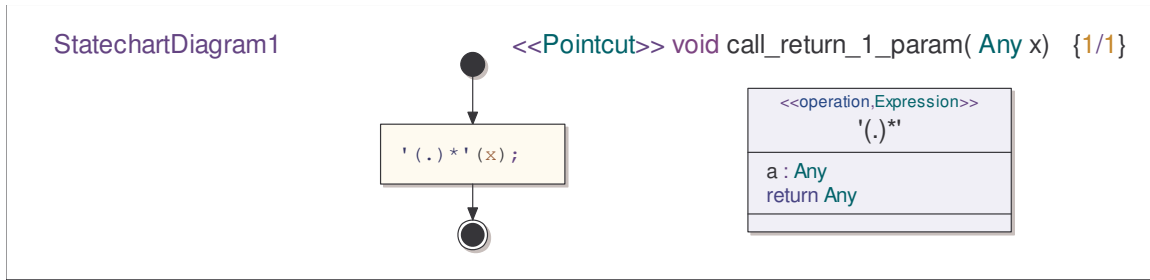


Figure 5. Tracing Aspect Call Expression Action Pointcut

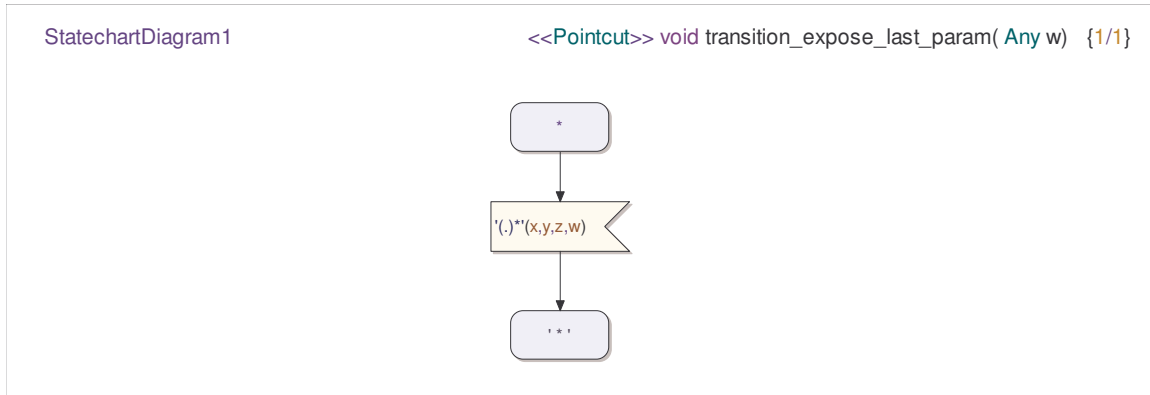


Figure 6. Tracing Aspect Triggered Transition Pointcut

Connector Diagram

A Connector specifies the behavior that is to be injected in the base model at all locations that match the Pointcuts to which the Connector is bound to.

A Connector has two features: a state machine diagram and an interface.

The interface of the Connector (Required Interface) must be compatible with the interface of all Pointcuts (Provided Interfaces) to which it is bound to. The Connector parameters correspond to the action arguments or the transition parameters of the actions or transitions that match the Pointcut.

The Connector state machine diagrams specifies how the Aspect behavior interacts with the actions or transitions matched by the Pointcuts. The actions or transitions that match the Pointcuts are represented in the Connector by a call to a special method called “proceed”. If “proceed” is not called in the Connector, then the matched actions or transitions are replaced by the Connector actions.

A Connector can potentially be instantiated in many contexts in the domain model. There is therefore a need to parameterize the actions of the connector according to the contexts in which it is instantiated.

The Aspect-Oriented Modeling profile therefore provides a special class called “thisJoinPoint”. thisJoinPoint represents the action or transition that is matched by the Pointcuts in a specific context. It provides a set of reflective methods that allow the Connector to retrieve information about the context in which it is instantiated.

Some of the thisJoinPoint interface methods are document as follow:

getName()	returns the signature of the event matched by the pointcut as a Charstring such as the signature of the called operation, the signal sent or the signal received
getJoinPointType()	returns a Charstring corresponding to the type of Joinpoint to which the Connector is bound to, such as 'CallExprAction', 'OutputAction' or 'TriggeredTransition'
getThisClassName()	returns a Charstring corresponding to the name of the class in which the Connector is instantiated
getTargetClassName()	returns a Charstring corresponding to the name of the class target by the Action in case of an Action Joinpoint. Otherwise, getTargetClassName() is equivalent to getThisClassName()
getSelf()	returns the Pid of the Active Class in which the Connector is instantiated
getTarget()	returns the Pid of the Active Class to which the signal is sent to in case of an OutputAction Joinpoint. Otherwise, getTarget () is equivalent to getSelf ()

The thisJoinPoint API also contains some methods that have been added for some more pragmatic purposes:

print()	This method builds up a print method based on the static structure of type of the Object passed as a parameter.
toString()	This method builds up a toString() method depending on the static structure of the type of the Object passed as a parameter.

::thisJoinPoint
<pre> + static getParameterTypeNames () : String + static getDeclaringTypeName () : String + static getName () : String + static getParameterNames () : String + static getThisClassName () : String + static inline ('in' : String) + static inlineObject (a : String) + static getThisClass () : TTDMetamodel::Class + static getTargetClass () : TTDMetamodel::Class + static getTargetClassName () : String + static getSelf () : Pid + static getTarget () : Pid + static print ('in' : T) + static toString ('in' : T) : Charstring + static getJoinPointType () : String </pre>

Figure 7. thisJoinPoint reflective API

These reflective features are useful in the case of the tracing Aspect. The Connector of Figure 8 uses the `thisJoinPoint` reflective API to print out a customized tracing message before and after the actions matched by the Pointcuts defined in Figure 2.

Note that the precise type of the Parameter passed to the Connector is not known by the Connector implementation, as denoted by the ‘Any’ parameter.

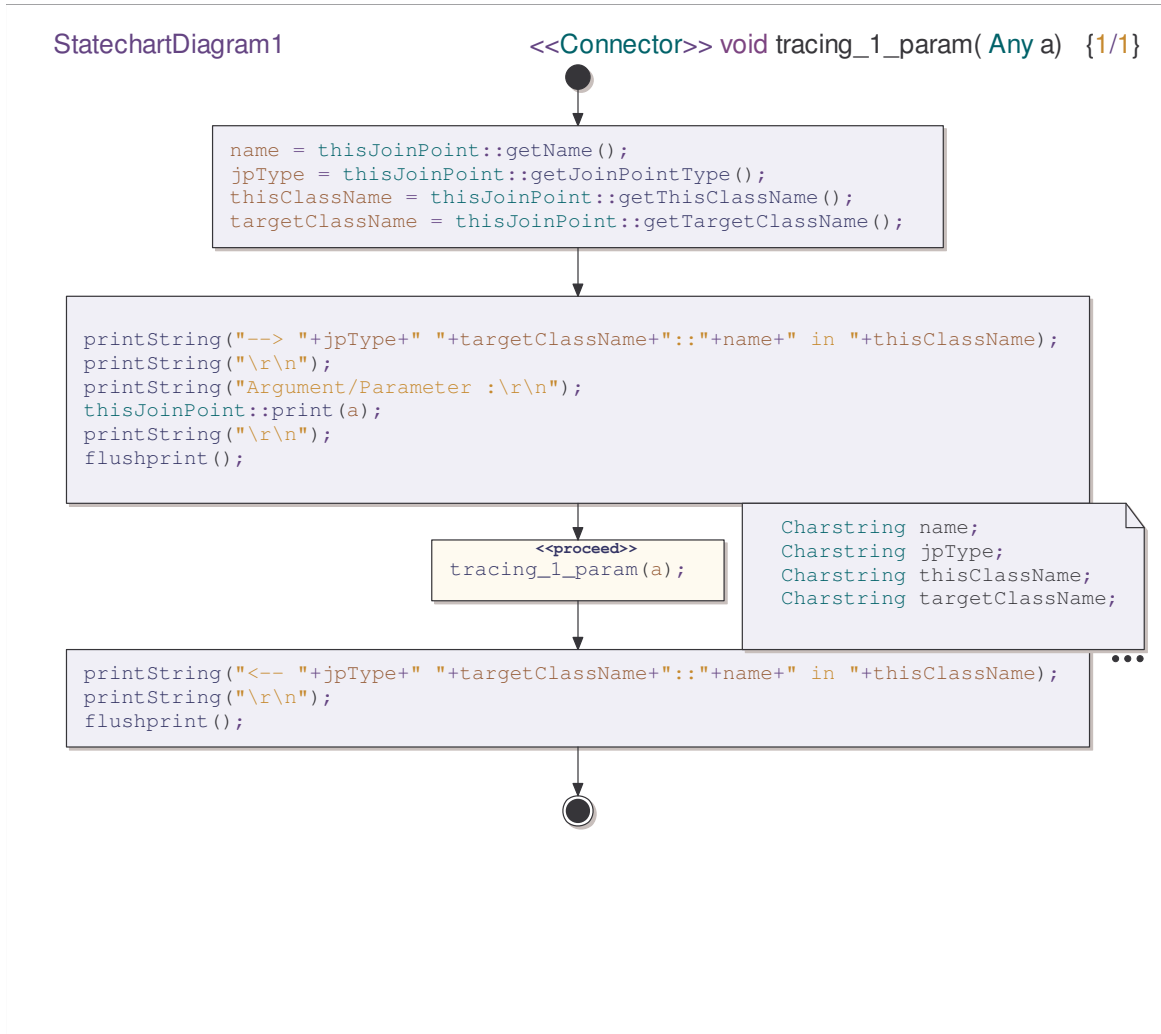


Figure 8. Connector of the Tracing Aspect

Aspect Deployment

The scope of application of an Aspect is defined by a “crosscuts” dependency. Figure 9 illustrates the deployment of the Tracing Aspect applied to two active classes, Player1 and Player2.

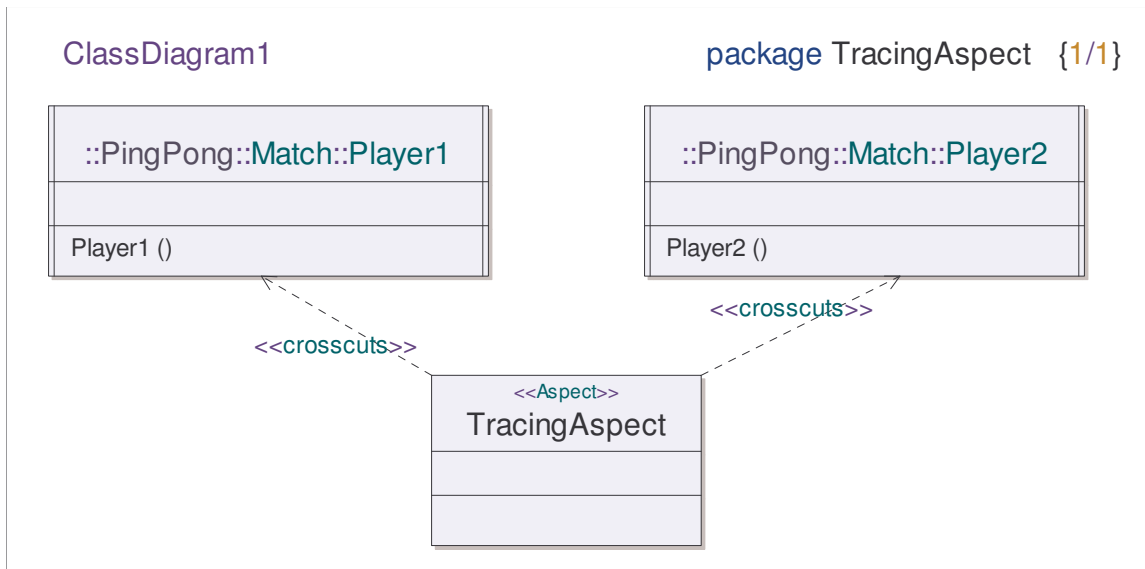


Figure 9. Deployment of the Tracing Aspect

ASPECT-ORIENTED VIEWS

When deploying an aspect in the project workspace, the add-in interprets the Aspects and performs two operations. First, it computes all the locations in the base model that match the Pointcut queries. Second, it instantiates the Connectors bound to those Pointcuts in the context of those locations. We never want developers to see woven models. Yet, we want to provide them with a way to inspect the base model and visualize the locations in the model that match Pointcuts and the corresponding Connector instances.

Joinpoint Annotations

The add-in provides a View of the base model where the locations at which the Aspects interact with the model are annotated. Such locations are called “Joinpoints”. A Joinpoint is either an action or a transition that matches a Pointcut.

The AOM profile provides a Joinpoint stereotype which is used to annotate the actions and transitions that match Pointcuts.

Figure 10 shows the state machine diagrams of 2 active classes that engage in an exchange of signals. These state machines are slightly modified examples of the Ping Pong example from the TAU G2 distribution.

9 joinpoints match the Tracing Aspect:

- 2 Call Expression Actions to the Pong2Ping and Ping2Pong methods of the Conversion class match the call_expose_all_1_arg pointcut
- 2 Start Transitions corresponding to the execution of the Pong2Ping and Ping2Pong operations of the Conversion class match the execution_expose_all_1_arg pointcut
- 3 Output Actions corresponding to the sending of the Ping and Pong signals match the output_expose_last_arg pointcut
- 2 Triggered Transitions triggered by the reception of the Ping and Pong signals match the transition_expose_last_param pointcut.

The add-in annotates those joinpoints with the “Joinpoint” stereotype and marks the corresponding symbols in the diagrams.

Symbols corresponding to Action Joinpoints are colored in yellow whereas Transition Joinpoints are delimited by small green task symbols. The interface of the Pointcut that correspond to these Joinpoints is also displayed in the property bar.

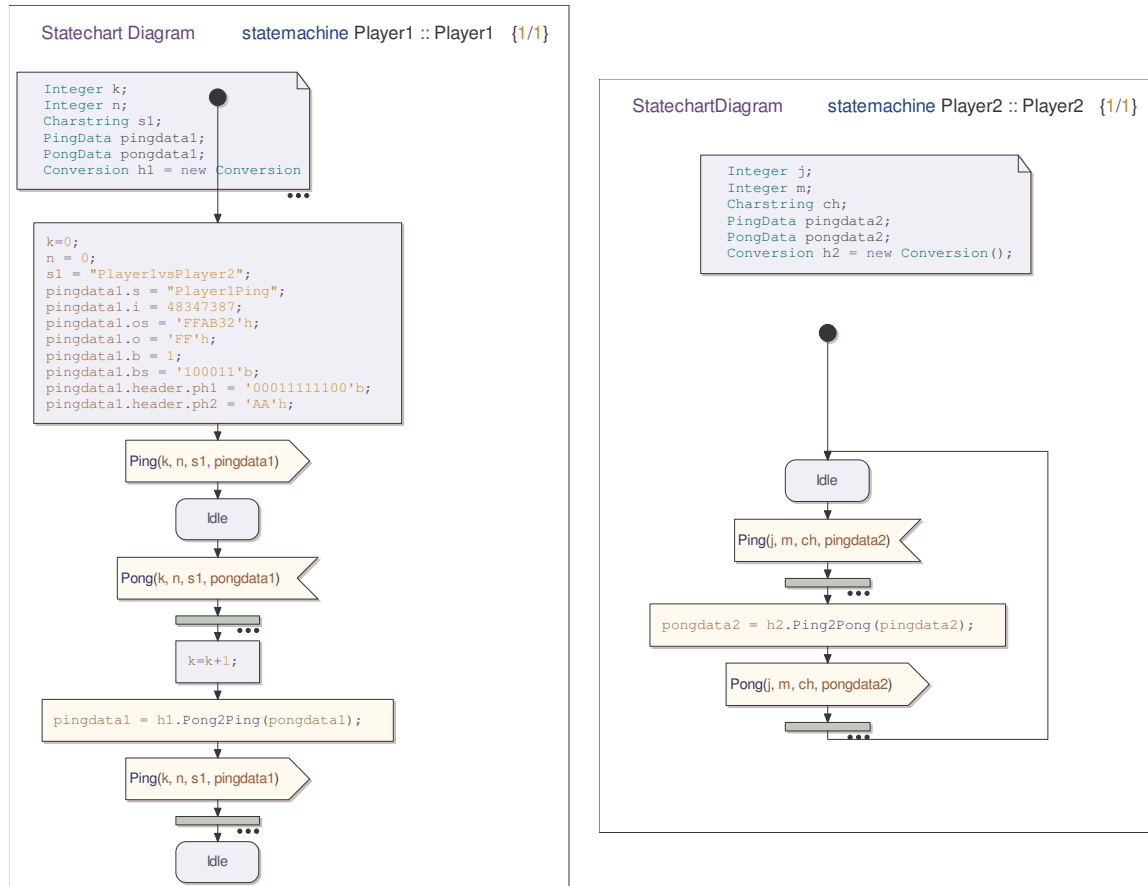


Figure 10. Ping Pong example with Joinpoint Annotations

Connector Instances

The Joinpoint annotations indicate to the developer that some additional behavior interacts with the base model at these points. In order to visualize this behavior hyperlinks are created between Joinpoints and the corresponding instantiations of the Connectors bound to the Pointcuts matching the Joinpoint.

The state machine diagrams of the Connector instances can therefore be visualized by clicking on Joinpoint symbols or the transition delimitation marks. However, Connector instances are not visible in the model view; they remain invisible to the developer unless he explicitly clicks on the Joinpoint symbols.

Figure 11 shows the instantiation of the Connector of Figure 8 in the context of the state machine of Player 2 for the Ping2Pong Call Expression Action joinpoint.

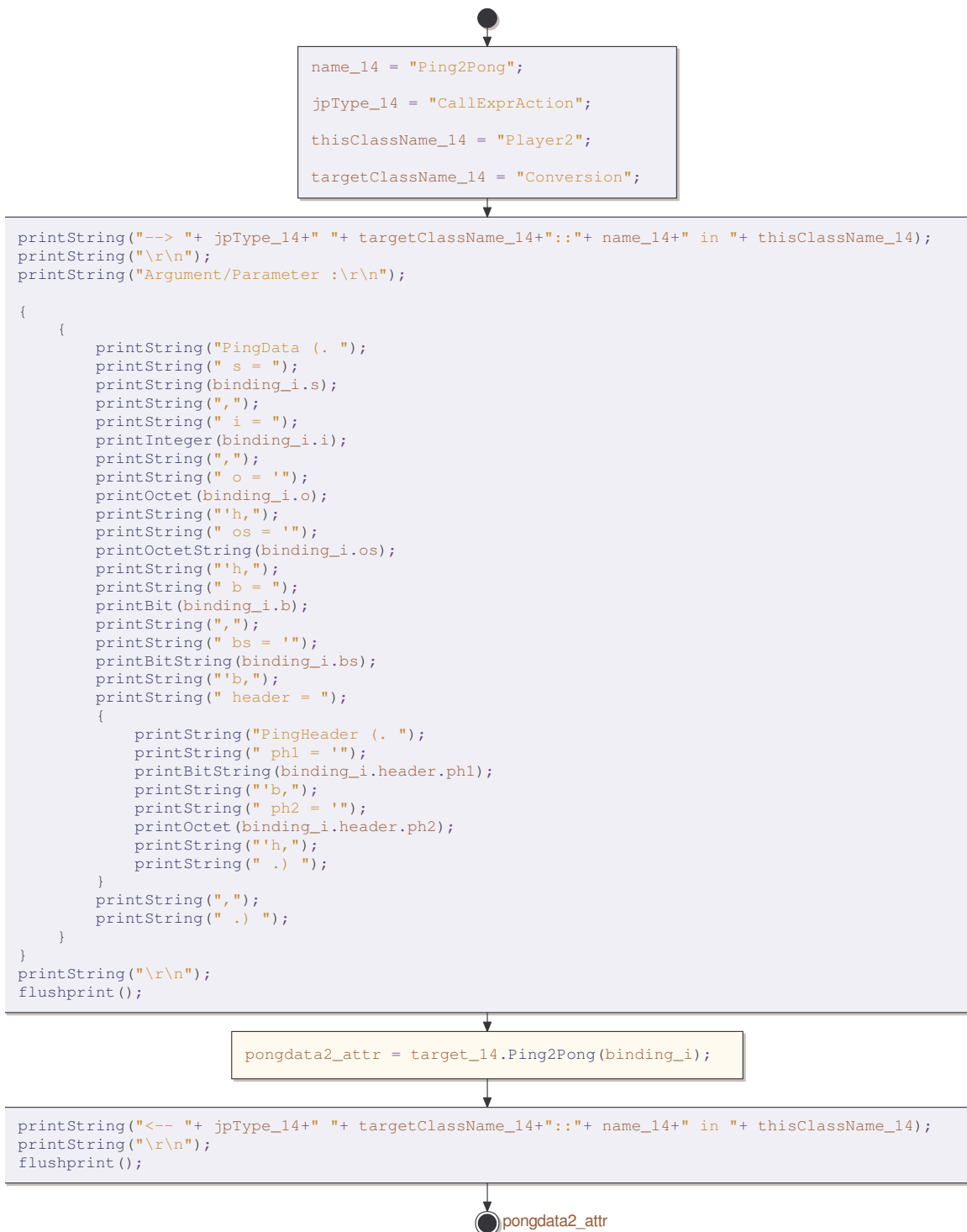


Figure 11. Connector Instance in the context of class Player2, as displayed when clicking on the Ping2Ping call expression action Joinpoint in Player2



Figure 12. Connector Instance in the context of class Player1, as displayed when clicking on the input symbol corresponding to the Pong triggered transition

During instantiation, all calls to the thisJoinPoint methods are resolved:

- getClass() resolved to "Player2"
- getTargetClassName() resolved to "Conversion"
- getName() resolved to the name of the method called, "Ping2Pong"
- getType() resolved to type of Joinpoint matched, "CallExprAction"
- thisJoinPoint::print() was resolved to print out the data structure passed as a parameter to the Connector.

Finally, the 'proceed' call has been resolved to the Joinpoint the Connector instance is bound to, in this case, the call to Ping2Pong.

Figure 12 displays a Connector Instance for the Pong triggered Transition. In the case of Transition Joinpoints, the Connector instance resolves the 'proceed' call as a call to a generate method that represents the transition selected by the Pointcut. The Transition Joinpoint is represented as a Start Transition. Figure 13 represents the Start Transition of the generated method that represents the Pong triggered Transition.

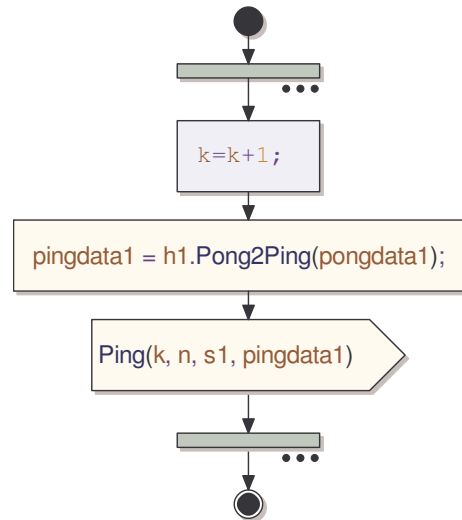


Figure 13. The Start Transition of the generated method representing the Pong Triggered Transition

SIMULATION OF ASPECT-ORIENTED MODELS

Except for the Joinpoint annotations, the AOM add-in has not modified the base model. Yet, the base model will behave differently in the Model Verifier. The add-in intercepts the TAU executable that generates the code that drives the model simulation (ABWSmSim) and performs the integration of the Connector instances and the base model right before code generation. In the case of Action joinpoints, it replaces the original call by a call to the Connector instance to which the Joinpoint is bound to.

After the code generation, the add-in performs the reverse operation, and restores the base model to its original state. A similar operation is performed for transition Joinpoints using Join Actions.

The model appearing in the Model Verifier is therefore the original base model, with annotated Joinpoints, while the code that drives the simulation corresponds to a woven

model. These operations allow us to simulate a woven model without the developer having to visualize it.

When the model verifier prompt is about to enter the call to the “Ping2Pong” operation of class “Conversion”, it does not jump to the state machine diagram that represents the body of the “Ping2Pong” operation. Instead, it jumps to the Connector Instance that has been instantiated in the context of the call. The model verifier prompt only proceeds to the implementation of the “Ping2Pong” method when the call corresponding to the “proceed” is executed in the Connector Instance.

In order to enforce the separation of concerns across the lifecycle, we need to decompose the trace that is generated by the Model Verifier into action occurrences from the base model and action occurrences that are introduced by the Aspects. The add-in therefore monitors the sequence diagrams that are generated by the model verifier, and restructures the action occurrences according to the Aspects defined in the model.

The action occurrences of the TracingAspect lifeline represented in Figure 15 actually occur in the context of the instance of the Player2 process (p2). The add-in filters out those action occurrences and makes them appear as if they occurred in the context of the Aspect. Different views of the generated sequence diagrams can be provided. Developers that are not interested in the concern modularized by the aspect can choose not to see the Aspect action occurrences, so they can focus on the behavior of the base model.

The effect of the Aspect on the Ping Pong example is to customize tracing in the Model Verifier tab. Figure 14 shows a sample of the trace produced by the TracingAspect in the model verifier message tab, when the TAU model verifier tracing is disabled (set-trace 0).

```
--> OutputAction Player1::Ping in Player1
Argument/Parameter :
PingData (. s = Player1Ping,
           i = 48347387,
           o = 'ff'h,
           os = 'ffab32'h,
           b = 1,
           bs = '100011'b,
           header = PingHeader (. ph1 = '00011111100'b,
                                  ph2 = 'aa'h
                                .)
           .)
<-- OutputAction Player1::Ping in Player1
--> TriggeredTransition Player2::Ping in Player2
Argument/Parameter :
PingData (. s = Player1Ping, i = 48347387,... .)
--> CallExprAction Conversion::Ping2Pong in Player2
Argument/Parameter :
PingData (. s = Player1Ping, i = 48347387,... .)
--> StartTransition Conversion::Ping2Pong in Conversion
Argument/Parameter :
PingData (. s = Player1Ping, i = 48347387,... .)
<-- StartTransition Conversion::Ping2Pong in Conversion
<-- CallExprAction Conversion::Ping2Pong in Player2
```

Figure 14. Sample of the trace produced by the TracingAspect in the model verifier message tab

CODE GENERATION FROM ASPECT-ORIENTED MODELS

When performing model weaving for the purpose of model verification we are constrained by the fact that we want to maintain the modular structure of the Connector instances.

For code generation, we want to perform model weaving as to minimize the performance overhead of the Aspects. In many cases, the aspects need to be inlined in the base model. For the purposes of code generation, we are not concerned with the presentation of the model. We can therefore throw the presentation of the models away, which facilitates the weaving operations.

CONCLUSIONS

This paper proposes a profile for the modularization of crosscutting concerns in state machine diagrams, using Aspect-Oriented Modeling. An add-in for Aspect-Oriented model transformation is discussed. The add-in enables developers to enforce separation of concerns between domain problem and extra-functional requirements throughout the system lifecycle, including system design, model verification, trace visualization and code generation.

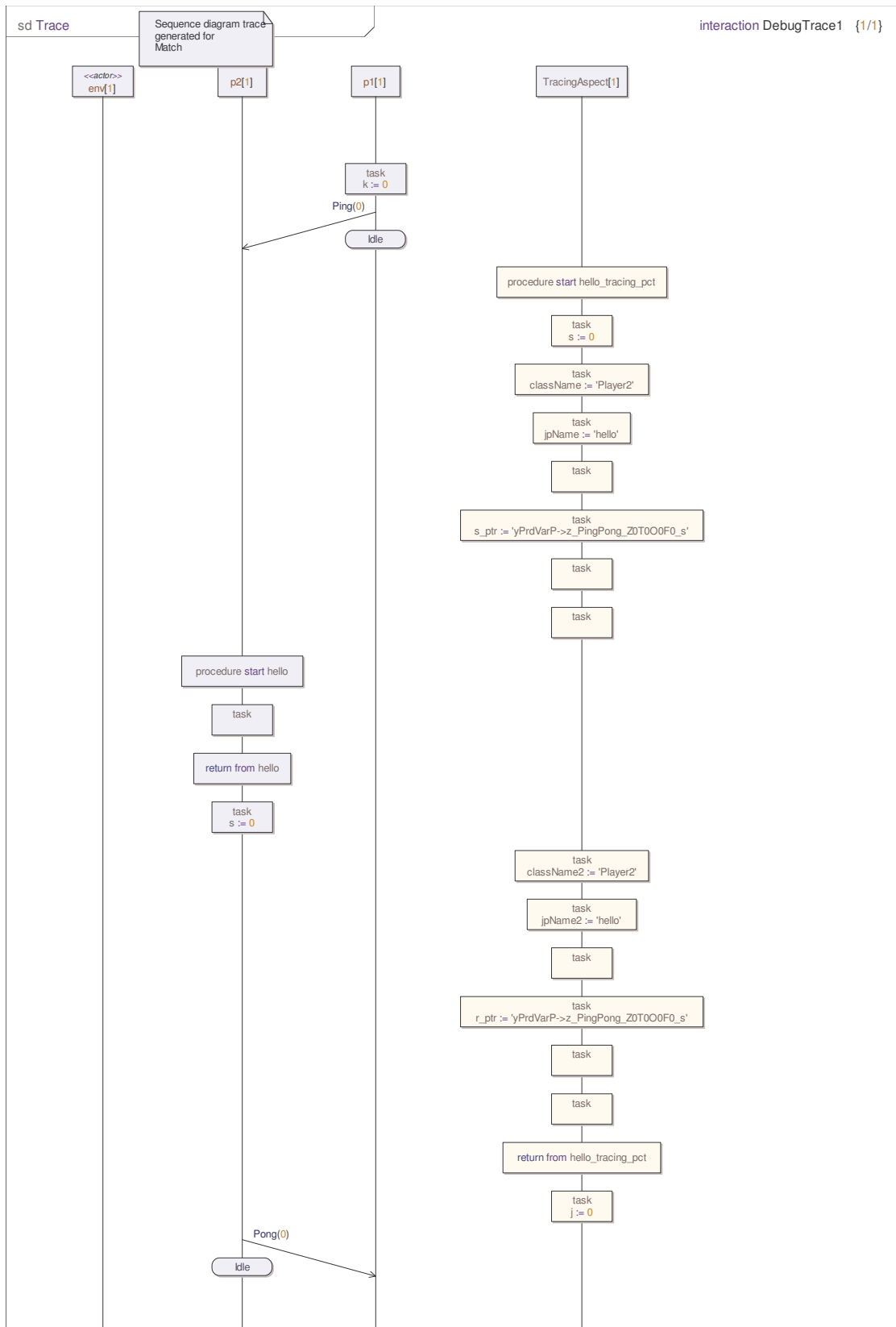


Figure 15. Aspect-Oriented decomposition of the sequence diagram generated by the model verifier

REFERENCES

1. Kiczales, G., et Al.: Aspect-Oriented Programming. Proceedings of the European Conference on Object-Oriented Programming, Springer-Verlag (1997)
2. Elrad, T., Filman, B., Aksit, M., Clarke, S.: Aspect Oriented Software Development, Addison Wesley (2004)
3. Aspect-Oriented Modeling homepage, <http://www.aspect-modeling.org/> (2001)
4. OMG: Model-Driven Architecture homepage, <http://www.omg.org/mda/> (2000)
5. AspectJ homepage, <http://www.eclipse.org/aspectj/> (2000)
6. OMG. MOF QVT Final Adopted Specification, Specification ptc/05-11-01, Object Management Group (2005)