Technical Report: Formal model for SECRET

Pascal Durr*, Lodewijk Bergmans, Mehmet Aksit
University of Twente, The Netherlands
{durr,bergmans,aksit}@ewi.utwente.nl

December 16, 2005

Abstract

This technical report provides a formal model for detecting semantic conflicts between aspects. The presented model abstracts from any AOP approach specifics. In [7] we make a preliminary instantiation of this formal model for the Composition Filter approach, we do plan to extend this work with an instantiation for AspectJ. The document starts with an example of a semantic conflict, next our approach is informally explained and finally the formal model is presented.

1 Problem statement

To illustrate the kinds of conflicts we consider, we present an example with two crosscutting concerns. One may of course discover numerous other examples of semantic conflicts between aspects. See for example[3].

Consider a base application which implements a simple protocol. Here, to handle inbound and outbound messages, the interface of class Protocol provides the methods sendData and receiveData. Now let us assume that we would like to add two new aspects: logging and encryption, which are applied to the same methods. The logging aspect, which is denoted as LoggingAspect is applied to the methods sendData(String) and receiveData(String). This aspect prints the arguments of both methods. The encryption aspect is denoted as EncryptionAspect [6] and provides encryption functionality for all outbound messages and the decryption for all inbound messages. The base system with both aspects is shown in figure 1.

In this example, both the logging advice and the encryption advice are applied to the same method sendData(String). Similarly, the logging advice and the decryption advice are applied to the same method receiveData(String). These two join points create semantic interference, which we would like to discuss in this section.

Consider, for example, the method sendData(String). Now assume that the logging aspect is used for debugging purposes and applied to plain massages only. In this

*This work has been partially carried out as part of the Ideals project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Senter program.

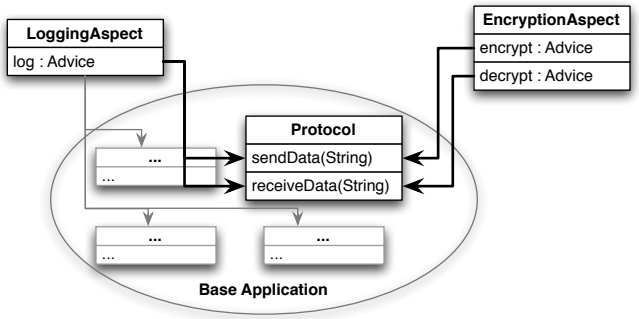


Figure 1: Encryption and Logging example

case, it seems to be a logical option to apply the logging advice before the encryption advice. However, one could also argue that the reverse order is preferable in a "hostile" context where the messages must be encrypted first before sending to the debugger. The exact order must be considered based on the requirements and therefore cannot be generalized to all cases. Now let us assume that it is required to apply logging before encryption. In this case, we consider applying the logging aspect after encryption is undesired, which is considered as a conflict situation between these two aspects.

Similarly, it is also possible to identify the following conflict for the method receiveData(String). At this method, the decryption aspect must be applied first before the logging advice. The opposite order is considered in this example as the second semantic conflict.

Now let us elaborate more on these two conflicting aspects. Individually, both aspects are consistent with their requirements and therefore they are considered sound. From the language compiler point of view, even the program with conflicting order is considered as a valid program without error. In a given requirements context, however, once these aspects are applied at the same join point, an emerging conflict situation appears. Such a semantic conflict may lead to undesired errors.

If one is aware of such (potentially) conflicting cases, he/she can enforce an ordering. For example, it is possible to enforce an ordering in AspectJ with the declare precedence construct. In practice, however, detecting emerging conflicts may not be that easy, especially when conflicting aspects crosscut the entire base application and share several join points. It is therefore necessary to develop techniques and tools to reason about the (potentially) semantic conflicts between aspects. By using the example case, the next section will informally explain our approach.

2 Approach

To reason about the behavior of advice and detect semantic conflicts among them, we need to introduce a formalization that allows expressing conflict detection rules. Clearly, a formalization of the complete behavior of advice in general would not only

be too complicated to manage, but also include too much detail and raise other issues. Therefore, an appropriate abstraction must be designed that can both represent the essential behavior of the advices, and be used to detect semantic conflicts among advice. Our approach is based on a resource-operation model to represent the relevant semantics of advice, and detect conflicts among them. We have chosen to adopt a resource-operation model, as this is an easy to use model that can represent both very concrete, low-level, semantics and very high-level, abstract behavior. For more detailed information about the model and its usage we refer to [7]. Our approach of conflict detection resembles the Bernstein[1] read-write sets for detecting possible deadlocks. A similar approach is also used for detecting and resolving conflicts in transaction systems, like databases[5]. A key idea is that some resource must be shared among advices for them to conflict. Hence semantic conflicts can be represented by modeling the operations that advice perform on some shared resource. In the following, based the example presented in section 1, we will explain the model intuitively. A more formal explanation is presented in section 3.

Figure 2 presents the flow diagram of our approach, starting from section **A** on the top ending at section **D** at the bottom. To avoid lengthy texts in the figure, we have decided to use graphical symbols. Aspects are represented as ellipses. Advices and crosscut specifications that belong to an aspect are represented respectively as rectangles and diamonds in the corresponding ellipse. Further, a standard UML class notation is used for base classes.

In part **A**, the aspects Logging and Encryption, and the base class Protocol are shown.

Part **B** in the figure shows the situation after the superimposition process, where the advices are attached at the shared methods. As shown in **C**, the advices that share a join point are subsequently composed into a sequence of advices. This could be compiler determined or explicitly defined by the programmer. Then, as shown in section **D** of the figure, for each shared join point, we transform sequences of advices to sequences of operations on resources and match conflict rules on the resulting sequences. This results in warnings or further actions if a conflict situation is detected.

2.1 Superimposition and Composition

In section 1, we have discussed the possible conflict situations in superimposing advices on the same join point. We have also assumed that for the method sendData(String), the correct order of application of advices should be first the execution of logging and then the encryption advice.

Figure 3 shows the shared join point and two possibilities of ordering the advices logging and encryption at this join point. According to the assumed requirements, the order in figure 3a is valid and the order in figure 3b is invalid. Form the perspective of our resource-operation model, a join point in fact determines the affected resources and a sequence of operations on these resources.

2.2 Model Transformation

In this section, we will describe how sequences of advices can be transformed to our conflict detection model. We start first by identifying the resources and subsequently

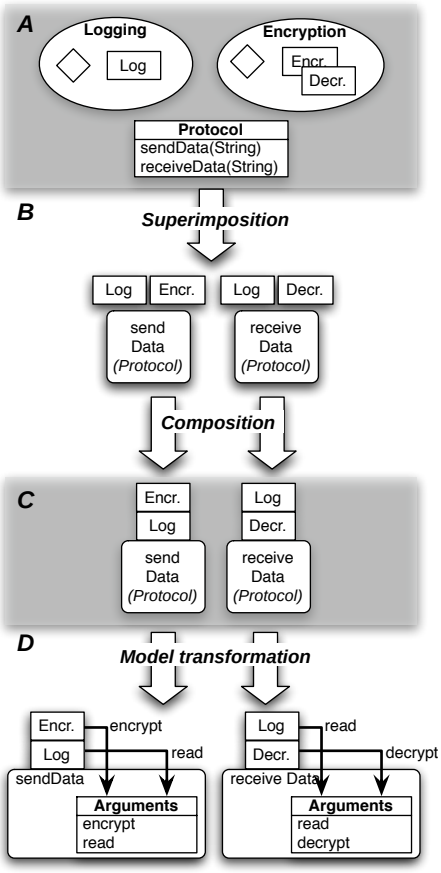
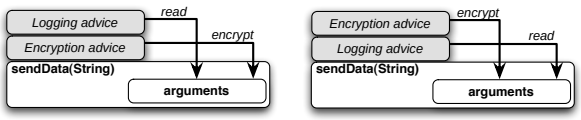


Figure 2: Overview approach

we determine the operations on these resources. Secondly, we describe the so called *conflicting patterns*, and finally we verify the model.

2.2.1 Resources

According to our resource-operation model, we have identified a set of generic resources that may be affected by the shared advices. One such resource is called arguments, which represent the arguments of received or sent messages. In fact, the logging, encryption and decryption advices all operate on a resource arguments. For example, the logging advice reads the argument of a message, whereas the encryption advice modifies the same argument. Similarly, the decryption advice also operates on the resource arguments.



(3a) Log **before** Encryption **(3b)** Log **after** Encryption

2.2.2 Operations

The logging advice accesses the arguments, this is a read operation on the arguments resource. The encryption advice encrypts the same arguments resource. Similarly, the decryption advice also operates, with a decrypt operation, on the arguments resource.

Although the very primitive actions on shared resources are basically read and write operations, if desired by the programmer, we think that such actions must be modeled at a higher level of abstraction. For example, in this paper, we will model both encryption and decryption advices as respectively encrypt and decrypt operations instead of read-write operations.

There is a subtle difference between changing the content of the arguments and transforming or encapsulating the data, as is the case with encryption. The intended meaning of the encryption and decryption advice is not to change the arguments. Also we would have lost our ability to distinguish between two, semantically, different actions: encryption and decryption. We chose to model the log action as a read operation as this adheres to the intended meaning of the advice. In short, the programmer must be able to choose his/her own higher-level operation definitions on the shared resources instead of primitive read-write operations only.

2.2.3 Conflict rules

A conflict rule is a requirement on a resource, which is specified as a regular expression on the sequences of operations per resource.

For example, in the example used in this paper, a conflict situation is specified as: "if a read operation occurs after an encrypt operation on the same resource, then it is considered as a conflict". Another conflict rule is specified as: "if a read operation occurs before a decrypt operation on the same resource, then it is considered as a conflict". These two requirements can be formulated using the following regular expression: $(((encrypt)(read)) \mid ((read)(decrypt)))$. In case of detecting error, several actions can be carried out, such as reporting the conflict to the programmer.

2.2.4 Conflict detection

For each shared join point, there is one sequence of operations on the resource arguments. In our example, we have thus two sequences, one for the method sendData(String) and one for method receiveData(String). Now let us assume that the regular expression shown in the previous section is tried to match against both operation sequences. Now assume that first an encrypt and then a read operation (caused by the logging concern) occur on the arguments resource at a shared join point. This would match the regular expression: $(((encrypt)(read)) \mid ((read)(decrypt)))$ and therefore raise a conflict.

The next section presents the conflict detection model in more detail.

3 Conflict Detection Model

Now we present the conflict detection model. This model is written in the formal specification language VDM[2]. To illustrate the formal model, figure 4 shows an overview similar to the overview of the informal approach as seen in figure 2. The same steps can be found in both figures. The labels for the transitions are replaced with names of the functions in our formal model.

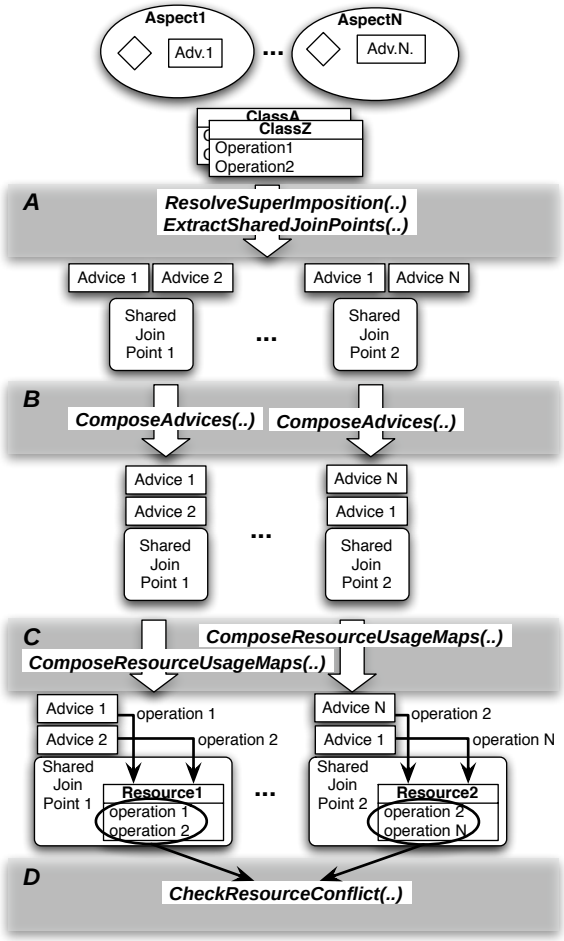


Figure 4: Formal approach

We first introduce the following primitives, starting with the type Name which is a non-empty, i.e. ordered, sequence($^+$) of characters.

$Name = char^+$

Next we define the types Operation and Operator, which is a name.

$Operation, Operator = Name$

Next we define the type ConflictRule, this is a set of all combinations of operations and operators among them. For example, in section 2.2.3, we have presented a regular expression for specifying conflicts. In the specification, the operations are read, encrypt and decrypt and the operators are denoted by the characters "*", "+" and "|".

$ConflictRule = Operation^* \times Operator^*$

We also introduce the types CrossCutLanguage and OrderingLanguage. We do not elaborate further on these types, except for stating that an instance of CrossCutLanguage selects language units and an instance of OrderingLanguage allows one to express ordering relations between aspects or advices.

$CrossCutLanguage, OrderingLanguage = \ldots$

The Resource composite type is defined as an Abstract Data Type[4], it is a 3-tuple with a name, a set of possible operations and a set of conflict rules. The conflict rules can be, in principle, in any suitable matching language, e.g. regular expressions, temporal logic or Prolog.

compose $Resource$ **of**
$name : Name,$
$alphabet : Operation$-**set**,
$conflictrules : ConflictRule$-**set**,
end

We define Resources as the set of all possible resources.

$Resources = Resource$-**set**

Once we have defined the resources and operations we can describe which operations are carried out on what resource. This is managed by a ResourceUsageMap, which is defined below.

$ResourceUsageMap = Resource \xrightarrow{m} Operation^*$
inv dom $ResourceUsageMap \subseteq Resources$

ResourceUsageMap keeps track of all operations on a specific resource. It maps a Resource to the corresponding, possibly empty, sequence of Operations. The invariant(**inv**) states that the domain(**dom**) of the mapping should be a subset of all resources.

3.1 Superimposition

Once we have defined the primitives of our conflict detection model we can define an advice, in terms of these primitives. In our model we abstract from specific AOP approaches, like AspectJ, Composition Filters, etc. To this end we abstract from implementation details and only consider the semantic implications of the advices. We do not propose a complete meta model here, we only focus on the semantic implications of an

AOP model, with respect to conflicts. An advice has a name and a ResourceUsageMap describing the operations on resources carried out by this advice, the VDM specification is shown below.

compose $Advice$ **of**
$name : Name,$
$rum : ResourceUsageMap$
end

Now we define an Aspect as a set of advices and a set of crosscut specifications, it also has a name.

compose $Aspect$ **of**
$name : Name,$
$advices : Advice$-**set**,
$crosscuts : CrossCutLanguage$-**set**
end

Subsequently, we define a JoinPoint[1] as a composite type with an id, a set of advices and a set of ordering specifications.

compose $JoinPoint$ **of**
$id : Name,$
$advices : Advice$-**set**,
$orderspecs : OrderingLanguage$-**set**
end

We can now define the function ResolveSuperImposition. See the gray region **A** in figure 4. The process of interpreting the CrossCutLanguage and resolving the superimposition depends very much on the chosen AOP approach. We therefore, only present the signature of such a function. ResolveSuperImposition is a function from the cross product of a set of advices and a set of crosscuts, to a set of join points.

$$ResolveSuperImposition : Advice\text{-}\mathbf{set} \times CrossCutLanguage\text{-}\mathbf{set} \rightarrow JoinPoint\text{-}\mathbf{set}$$

$$ResolveSuperImposition(advices, crosscuts) \triangleq \ldots$$

$$SharedJoinPoints = JoinPoint\text{-}\mathbf{set}$$

ExtractSharedJoinPoints is a function from a set of join points to a set of join points. See the gray region **A** in figure 4. The function iterates over all join points and concatenates(indicated by the rounded arrow) a new shared join point to the old (indicated with a reversed arrow on top) SharedJoinPoints. This is only done if the number of advices(**card**) in the set of advices for a join point is equal or greater than two.

[1]We assume that the same advice is only applied once to each join point.

$ExtractSharedJoinPoints : JoinPoint\text{-}\mathbf{set} \rightarrow JoinPoint\text{-}\mathbf{set}$

$ExtractSharedJoinPoints(joinpoints) \triangleq$
$SharedJoinPoints = \overleftarrow{SharedJoinPoints} \cup$
$\forall jp \in ResolveSuperImposition(advices, crosscutspec) \cdot jp$

pre card $jp.advices \geq 2$

3.2 Composition of Advices

Abstracting over all composition mechanisms at shared join points of all AOP languages is hard. For our semantic conflict detection, we do not make any assumptions about the composition model, but mainly we restrict the composition mechanisms to only produce a single sequence of advices. For example, we do not introduce concurrency in advice executions. We introduce the function ComposeAdvices which returns a sequence of Advices, given a set of advices and a possible ordering specification. This composition mechanism is language specific and is thus, not specified here. This function is shown the gray region **B** in figure 4.

$ComposeAdvices : Advice\text{-}\mathbf{set} \times OrderingLanguage\text{-}\mathbf{set} \rightarrow Advice^*$

$ComposeAdvices(advices, orderingspecs) \triangleq$
$\ldots$

The ordering specification is domain or application specific information. The implementation of the ComposeAdvices function itself incorporates, implicitly, the AOP approach specific language constraints, e.g. in AspectJ, *before* advice is always executed before *after* advice. The result of composition is thus a sequence of Advice called ComposedAdvicesSeq:

$ComposedAdvicesSeq =$
$ComposeAdvices(advices, orderingspecs)$

3.3 Detection of conflicts

Given the sequence of advices we can now transform this sequence to our model. First we define a ComposeResourceUsageMaps operation, which provides, given a sequence of advices and a resource, a composed sequence of operations for this resource. The preconditions(**pre**) state that resource r should be in the set of Resources and that advice sequence adviceseq may not be an empty sequence. The postcondition(**post**) states that for each advice in advieseq the corresponding operations for r are concatenated with the old operationseq. This operation is shown in the gray region **C** in figure 4.

$ComposeResourceUsageMaps\ (r\text{: }Resource, adviceseq\text{: }Advice^*)$
$operationseq\text{: }Operation^*$

pre $r \in Resources, adviceseq \neq [\,]$

post $operationseq = \overleftarrow{operationseq} \curvearrowright$
$\forall advice\text{: }adviceseq \cdot advice.rum[r]$

We define a helper function, MatchConflictRule which tries to match a conflict rule, rule, on a sequence of operations, operationseq. This function returns **true** if the rule matches the sequence, and **false** otherwise. The implementation of this function is beyond the scope of this paper, but in principle this can be any matching language.

$$MatchConflictRule : ConflictRule \times Operations^* \rightarrow \mathbb{B}$$
$$MatchConflictRule(rule, operationseq) \quad \triangleq$$
$$\ldots$$

With the ComposeResourceUsageMaps and MatchConflictRule operations we can now define the CheckResourceConflict function which determines, given a resource r, whether this resource is conflict free. The result is a boolean value, this is only true if there is a conflict in the operation sequence for this resource, and false otherwise. This operation is shown in the gray region **D** in figure 4.

$$CheckResourceConflict\ (r{:}\ Resource)\ result{:}\ \mathbb{B}$$
pre $r \in Resources$
post let $crum = ComposeResourceUsageMaps$
$(r, ComposedAdvicesSeq)$ **in**
$result = \forall rules \in r.conflictRules \cdot$
$MatchConflictRule(rule, crum)$

We define a shared join point to be conflict free **if and only if**:

$\forall resource \in Resources \cdot$
$CheckResourceConflict(resource) \neq$ **true**

References

[1] A. J. Bernstein. Program analysis for parallel processing. *IEEE Trans. on Electronic Computers*, EC-15:757–762, 1966.

[2] J. Dawes. *The VDM-SL reference guide*. Pitmann, 1991.

[3] P. Durr, T. Staijen, L. Bergmans, and M. Aksit. Reasoning about semantic conflicts between aspects. In *EIWAS '05: The 2nd European Interactive Workshop on Aspects in Software*, Brussel, Belgium, September, 1-2 2005.

[4] D. Kapur and S. Mandayam. Expressiveness of the operation set of a data abstraction. In *POPL '80: Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 139–153, New York, NY, USA, 1980. ACM Press.

[5] N. A. Lynch, M. Merritt, W. E. Weihl, and A. Fekete. *Atomic Transactions : In Concurrent and Distributed Systems*. Morgan Kaufmann, 1993.

[6] L. Z. Minwell Huang, Chunlei Wang. Toward a reusable and generic security aspect library. In *AOSD:AOSDSEC '04: AOSD Technology for Application-level Security*, Lancaster, UK, March, 23 2004.

[7] Pascal Durr. Detecting Semantic Conflicts Between Aspects. In *Detecting Semantic Conflicts Between Aspects*, pages 57–70, 2004. http://www.cs.utwente.nl/˜durr/papers/ Master_Thesis_Pascal_Durr.pdf.