Semantic Aspect Interactions and Possibly Shared Join Points

Emilia Katz Shmuel Katz
Computer Science Department
Technion – Israel Institute of Technology
{emika, katz}@cs.technion.ac.il

ABSTRACT

When multiple aspects can share a join-point, they may, but do not have to, semantically interfere. We present an in depth analysis of aspect semantics and mutual influence of aspects at a shared join-point, in order to enable programmers to distinguish between potential and actual interference among aspects at shared join-points. An interactive semi-automatic procedure for specification refinement is described, that will help users define the intended aspect behavior more precisely. Such a refined specification enables modular verification and interference detection among aspects even in the presence of shared join-points.

Keywords

Aspect interference, semantics, specification, shared join-points

1. INTRODUCTION

Multiple aspects, when woven into the same base system, might happen to have common join-points. This possibility gives rise to many important questions and problems, from understanding how the potentially applicable advice pieces should be woven at such a common join-point (as they cannot be applied all at the same time), to conflict detection and resolution, since application of one advice might interfere with the computation or even applicability of another.

In this paper we consider this question in depth, for aspects and systems modeled as state transition diagrams, with specifications given as linear temporal logic (LTL) assumptions and guarantees. As a solution to aid in understanding the implications of shared join-points, we describe an easily automatizable interactive procedure that will help users specify their intentions for aspect behavior in a specific system more precisely, and check whether there is actual interference with respect to this specification. Based on the answers to a series of questions to the user, the LTL specifications of the aspects are automatically augmented. The state transition model is also modified to handle shared join-points. Then the verification and automatic interference checks from [10] can be used to detect subtle cases of interference at shared join-points, or establish that there is no such interference, using the augmented specifications.

Ways to detect shared join-points are described in [14, 15]. Several works study shared join-points as a source of possible conflicts, some (e.g., [14]) even see common join-points as the main source of interference among aspects. A language independent technique [8] makes it possible to check whether an undesired order of aspect application at a shared join-point is possible, where the list of undesired orders has to be explicitly provided by the user. It is implemented in the "Secret" tool for Compose* [2, 13]. However, presenting the undesired orders list requires a thorough analysis of the system by the user, and also might not be able to reflect all the intended behaviors, as at different states different orders of application might be possible. In [1] another tool for checking potential interference at common join-points is described, applicable for the Compose* language. It checks all the possible orders of aspect applications at a common join-point, and declares a conflict if different orders result in different resulting states. This method is fully automatic, but may lead to many false positives (some of which are described later in this paper). An additional tool for aspect interference detection, performing dataflow checks to find out whether one aspect affects variables used in another, is presented in [17]. It can also be used to check interactions at shared join-points, though its scope is broader. Interactions found by this tool are also only potentially harmful.

Weaving techniques for conflict resolution at shared join-points appear in [14, 6, 7]. In [14], a first analysis of types of mutual influence of aspects applied at a shared join-point appears. This analysis is extended in our paper, though used for a different purpose.

The intended semantics of weaving several aspects at a common join-point in a pre-defined order is addressed in papers on the semantics of aspects, such as [16, 4, 5].

However, as described in [12], not all the conflicts at shared join-points can be resolved by a clever weaving. Thus it is important for the user to be able to detect the conflicts and differentiate between real problems and false alarms. Below we show the possible influences of multiple aspects on each other at a join-point, and classify them according to the way they affect the specification of the aspect.

The paper is organized as follows: Our analysis of aspect semantics and types of mutual influence at shared join-points appears in Section 2. In Section 3 we show the questions the answers to which can be automatically processed to add appropriate predicates to a temporal logic specification. Section 4 gives examples of applying the questions. We conclude in Section 5.

2. SEMANTICS OF ASPECT BEHAVIOR AT A COMMON JOIN POINT

In AspectJ, the aspects at a common join-point are applied one after another, and each time before performing an advice the pointcut condition is re-checked. As a result of such a semantics, when a base system arrives at a join-point matched by an aspect A, it is not necessarily the case that the advice of A is immediately executed. It might be the case that other aspects are present in the system that also match this join-point, and it might thus happen that some other advices are executed before the advice of A, changing the state of the system in which A will be applied. Moreover, A's advice might not be executed at all, in case one of the previously executed aspects left the system in a state which is not a join-point of A any more.

Thus the execution of the woven system from the moment it arrives at a join-point matched by some of its aspects is determined not only by the set of matching aspects, but also by the order of their application at this point. So if this order of application is not explicitly prescribed by the user, the non-determinism of aspect application may result in different states.

However, the fact that different orders of advice application lead to different resulting states does not necessarily mean that the aspects semantically interfere. Let us consider the following example, presented in [1]: Several aspects are defined for systems in which messages of type String are sent between objects. Two of these aspects are Logging and Encryption. Both aspects are applied at the same join-points - when a message is sent in the system - and different orders of their application will result in different states of the system. If Logging is executed before Encryption, the logged message will be the original one, otherwise it will be the encrypted message produced by the Encryption aspect. In [1] such a situation is considered interference between the two aspects, but in fact the decision on whether it is interference or not should depend on the aspects' specifications. In our example, the goal of the Encryption aspect is to encrypt every message before it is sent to the server. Consider the following possible specifications of the Logging aspect, described more formally in Section 4:

1. The log should record all the messages as they were originally attempted to be sent by the user, so that the user will be able to view the list of messages (s)he sent.
2. The log should record all the messages as they were actually sent to the server in order to compare the sent messages to the received ones (as received) and verify that no messages got lost or garbled.
3. The goal of the Logging aspect is to measure the network activity of the system. Thus, though the contents of the messages are written to the log, they are of no importance to the user, and what matters is only the number of messages sent and their frequency, e.g. the times of the messages sent and the number of lines in the log.
4. The logging records all the attempts to send a message, even if they are aborted for whatever reason. It logs each message as it was attempted to be sent by the user.

All the cases above can happen in our example system, and different order of application of the two aspects at their common join-point will lead to different resulting states, but not in all the cases above do the aspects interfere. The requirements from the Encrypting aspect are never violated by Logging, no matter in what order they are executed, but in variants (1), (2) and (4) the aspects may interfere: in variants (1) and (4), the goal of Logging will not be reached if the Encrypting aspect is applied first, and in variant (2), applying Encrypting after Logging will cause a problem. However, in variant (3) applying the aspects in any order will not violate the requirements from Logging or from Encrypting, thus there will be no interference. As will be shown later, an Authorization aspect can also be applied, further complicating the situation.

The above example shows the need to analyze possible semantic effects of sharing a join-point more deeply. We consider the AspectJ operational semantics, where first all the places in the code of the base program that are matched by the static part of some aspect's pointcut, are identified. Such places are called *shadow join-points.* Note that a shadow join-point is usually defined by a place in the code of the program, but sometimes can contain additional information. After shadow join-point identification, at each such join-point the weaving order of the potentially applicable aspects is defined (an aspect is considered potentially applicable if the static part of its pointcut matches the current shadow join-point). The weaving order does not have to be defined statically, it can be determined upon arrival of the computation at the join-point. At last, when a computation arrives at a join-point, each of the potentially applicable aspects, one by one and in the previously defined order, is checked for full applicability and immediately executed if indeed applicable (i.e., if both static and dynamic parts of the pointcut are matched by the current state). All the rest of the paper refers to this semantics, and if a different semantics is chosen, different reasoning might be needed. This operational semantics shows the need to reason about the part of computation between the first moment it arrives at some shadow join-point and the moment it leaves this join-point, which includes all the aspect applications performed at the join-point. We need some new terminology to make this reasoning easier. First of all, we need a name for the period of interest:

DEFINITION 1. *A sequence of states $s_1, \ldots, s_k$ in a computation of the woven system is called a* pointcut occurrence *of aspect A if s_1 is the state when a join-point of A is first reached (that is, s_1 is matched by the full pointcut descriptor* *of A, and the previous state is not), and s_k is the state when the computation is about to leave the corresponding shadow join-point, after application of all the appropriate aspect advices according to the current weaving policy (that is, s_k is matched by the static part of the pointcut descriptor of A, and the next state is not).*

Two aspects share a join-point if they have at least one overlapping pointcut occurrence. Note that overlapping pointcut occurrences do not have to coincide, as it might be a case that an execution arrives at a state s matched by the pointcut descriptor of aspect B, and by the static part of pointcut descriptor of A and B, but not matched by the dynamic part of A's pointcut descriptor, and only the execution of aspect B at s will result in a state in which both static and dynamic parts of A's pointcut hold. In such a case the pointcut occurrence of A will be contained in the pointcut occurrence of B, but not vice versa. For example, consider the case of aspects A and B applied to a grades managing system. Let aspect B be responsible for giving bonuses and factors to grades, and let aspect A be in charge of enforcing a required grades format, by rounding non-integer grades and replacing all the grades above 100 by 100. Both aspects are applied before the publication of the grades, so the static parts of their pointcuts are the same. However, aspect A should be applied only if the grade to be published is not an integer or exceeds 100. Clearly, a computation might arrive at a place before grade publishing with an integer grade below 100, thus matched by the dynamic part of B's pointcut only (and not of A's), but as a result of B's modifications, a non-integer grade or a grade above 100 is obtained, bringing the computation to a state that is matched by A's pointcut as well.

Previously, two kinds of join-points have been examined: shadow join-points and actual join-points. A shadow join-point of aspect A, as mentioned above, is a place in the code of the base program that is matched by the static part of A's pointcut. An actual join-point of A is a state in a computation of the system at which the advice of A is actually applied. However, for the purpose of our analysis, a third type, *arrival join-points*, is needed:

DEFINITION 2. *A state s in a computation of the woven system is called an* arrival join-point *of aspect A if s is the first state of a pointcut occurrence of A - the state when a join-point is first reached in that occurrence.*

Note that an arrival join-point differs from a shadow join-point because it is matched also by the dynamic part of A's pointcut descriptor. It also differs from an actual join-point: when the case of shared join-points was not possible, every arrival join-point reachable in the woven system became an actual join-point, but now it does not have to be so, because other aspects can intervene.

Note also that the pointcut of A identifies states either before or after some events of interest, but the definitions above are applicable for both cases. And if A is an `around` advice, it either has a `proceed`, and then can be viewed as a combination of two advice pieces - one before, and one after the corresponding event, or A has no `proceed`, and then can be viewed as a `before` advice, one of the effects of which is a change in the program counter of the base system. If indeed A's advice changes the program counter of the base system, the end of its execution is also the end of the current pointcut occurrence - both according to our intuition and to the definitions above.

Now let a state s be a join-point matched by aspect A, that appears inside pointcut occurrence π. We distinguish between four possible cases of other aspects' behavior that can influence the result of weaving A into a system:

1. Aspect B executed before A in π changes a value of some variable used by A as an input to its computations.
2. Aspect C executed after A in π changes a value of some variable updated by a computation of A.
3. Aspect D executed before A in π brings the system to a state s' which is not a join-point of A any more.
4. Aspect E executed after A in π invalidates the condition on which the join-point predicate depended, thus removing a join-point of A after A has already been executed at it.

The following analysis enables us to determine whether the above described influences actually cause an interference: Let the semantics of A be given by the assumption-guarantee pair (P_A, R_A), where R_A is the guarantee of A that must hold in any woven system containing A, provided the system into which A has been woven satisfied the assumption of A, P_A. Note that A can be woven into a system that does not satisfy P_A, but then R_A is not guaranteed to hold in the resulting system. We denote by $V_{in}(A)$ a set of variables A uses as input to its computations, and by $V_{out}(A)$ - a set of variables in which A stores the result of its computations.

Case 1. **Change Before (CB).** In case an aspect B executed before A at s changes a value of a $v \in V_{in}(A)$, the result of A's calculations might differ from the one we would get if the value of v has not been changed from the moment the computation arrived at s till the moment the advice of A was executed. If the guarantee of A contains a requirement for A's correctness, and this requirement is formulated in terms of a specific connection between the value of v when we arrive at a join-point and the value of v after the computation of A is finished, R_A will be violated in this case. (This can happen, for instance, in variant (1) of the Logging and Encrypting example above: we anticipate that the message string written to the log is the one created by the user and readable by the user, but if Encrypting is executed before Logging, what we actually get in the log is the encrypted message, because the contents of the message was changed by the Encrypting aspect.) Note that if the requirement for correctness of A's calculations binds the values at the end of A's execution only to the values at the beginning of execution of A (as in variants (2) and (3) of the Logging and Encrypting example, with Encryption before Logging), it will not be violated in this case.

Case 2. **Change After (CA).** In case some aspect C executed after A at s changes a value of a $v \in V_{out}(A)$, the guarantee of A will be violated if it required preservation of the result of A's computation till some future point in the execution where the value of v is used. (As in variant (2) of the example, when Logging occurs before Encryption). Otherwise, as in variant (3) of the example with Logging before Encryption, the guarantee of A will not be influenced. If, indeed, a requirement for preservation of the value of v till some state *use_v* is part of A's guarantee, then part of A's assumption should be that in the base system the value of v is not modified from the actual place of application of A's advice till arrival to the *use_v* state.

Case 3. **Invalidation Before (IB).** In this case there is no state in A's pointcut occurrence at which A is executed. Such a situation happens, for example, with Logging and Authorization aspects from Section 4 when the Authorization aspect is applied before Logging and the authorization of the user fails, thus preventing message sending, and removing the join-point of the Logging aspect. In variant (4) of the Logging specification, this leads to violation of the guarantee of Logging, as a message was prepared for sending and should have been logged, but the Logging aspect never has a chance to be applied, because the authorization failure finishes the pointcut occurrence.

Case 4. **Invalidation After (IA).** In this case A is executed at some point at which it shouldn't have been applied, because when arriving to the point of interest, the weaver "does not know" that the reason for A's application will be removed by one of the aspects coming after A. If the specification of A requires that it is applied only if followed by some event in the future, and this following event is removed by another aspect, then the specification of A is violated. This is the case, for example, in variants (1), (2) and (3) if the Authorization aspect is applied after Logging and the authorization of the user fails. Note that in variant (4), on the other hand, the guarantee of Logging is not violated if Logging precedes Authorization.

3. SPECIFICATION OF ASPECTS WITH POSSIBLY SHARED JOIN-POINTS

3.1 Guided Specification Construction

In order to be able to detect situations in which application of other aspects at a common join-point may contradict the specification of the examined aspect, the specification of the aspect must be expressive enough. The LTL specifications include:

- "$\mathsf{G}\,\varphi$" ("Globally") - meaning that the formula φ is true from the current state on.
- "$\mathsf{F}\,\varphi$" ("Finally") - from the current state a state in which φ holds can be reached.
- "$\mathsf{O}\,\varphi$" ("Once") - dual to "Finally": a state satisfying φ occurred earlier in the computation.
- "$\varphi\ \mathsf{U}\ \psi$" ("Until") - a state in which ψ holds is reached later in the computation, and until then φ holds.
- "$\varphi\ \mathsf{W}\ \psi$" ("Weak Until") - almost like "Until", but a state in which ψ holds does not have to be reached. In this case φ holds from now on forever.

We say that a computation satisfies an LTL formula if this formula holds in its first state. From the analysis in Section 2 the need for the following predicates arises:

- *at(ptc)*: assuming that *ptc* is the predicate defining A's pointcut, the predicate *at*(*ptc*) means that the computation has just arrived at a join-point of A. It is useful for reasoning about what happened in the computation after the moment it arrived at a possibly shared join-point. In fact, this is the predicate marking the arrival join-points of A.
- *after_prev_asp(A)*: this predicate becomes true at the moment the weaver has applied all the aspects that preceded A at the current shadow join-point, according to the algorithm of the current weaver. The usage of this predicate is twofold: First, the user has to refine the definition of A's pointcut by taking into consideration the new predicate, *after_prev_asp(A)*, because now A should only be applied at states satisfying both *ptc* and *after_prev_asp(A)*, so the pointcut of A becomes $ptc \wedge after\_prev\_asp(A)$ (which matches the definition of the set of all the actual join-points of A). Second, the predicate is used in assumptions added to A's specification when the cases of "change before", "change after" and "invalidation before" presented in Section 2 are possible, to be able to reason about the behavior of the base system from the moment its computation arrives at a join-point of A till the moment A's advice is actually executed.
- *promise_ful(A)*: this flag is used by the weaver in order to give each of the aspects sharing a shadow join-point exactly one chance to be applied at it, as is explained later in Section 3.2. The flag $promise\_ful(A)$ is false when the computation arrives at a shadow join-point, becomes true at the moment the execution of A's advice begins, and remains true until the computation leaves this shadow join-point.
- *asp_ret(A)*: this predicate describes the possible return states of the aspect. This is needed for some of the cases below. Typically, the aspect return state has the same control location as the join-point state (the values can change, but not the program counter of the join-point). For the Logging aspect, for example, the base state is actually not changed, and only the log (local to the aspect) is modified. However, it does not have to be so in general. Thus in order to define the *asp_ret(A)* predicate, the user is proposed a default predicate, automatically constructed by the system as described in [11]. The idea of construction is to create a system containing representations of all the possible computations of an aspect from all its possible initial states, without actually applying the aspect to any specific base system (this is done using the MAVEN tool [9] and some built-in functionality of the NuSMV [3] model-checker), and then to build a predicate describing all the states of this system that satisfy

the return conditions of the aspect. This default predicate can then be manually modified.

Using the above predicates, all the requirements mentioned in Section 2 can be expressed, though not all the predicates are needed for all the cases of specification : sometimes some of them can be abstracted out without loss of precision of modeling and of subsequent verification. Below we present a way to express each of the additional requirements.

The construction of the refined specification can be automatic, but user-guided: several guiding questions will be presented to the user, and the answers to these questions will determine the new requirements. The construction process will thus be as follows:

Step 1. Here we will treat the dependency of our aspect, A, on its input variables, in order to find out whether the values of the input variables need to be preserved between the arrival and the actual join-points of A (in order to be able to treat the "change before" case from Section 2). The user is asked the following question:

Q. 1: Are there any input variables of A for which the advice of A depends on the value as it is at the arrival join-point and not as it is when the advice of A actually starts its execution?

- If yes, the user should provide a list of variables for which such a dependency exists.
- For each variable v in the list, we add the following CB (for "Change Before") statement to the assumption of A:

  $$CB(v) = \mathsf{G}[(at(ptc) \wedge v = V) \rightarrow ((v = V \;\mathsf{W}\; (after\_prev\_asp(A) \wedge v = V))]$$

  where V is a logical variable keeping the value of v as it was at the arrival to the join-point.
- If there are no variables in the list, nothing is added to the specification of A at this step.

Step 2. Here we treat the case when part of the effect of the aspect is modification of some state variables, and this effect should be preserved till some point in the future of the computation. This is important for the "change after" case from Section 2. The questions asked here are:

Q. 2: Are there any state variables of the system into which A is woven the value of which should be preserved after A's execution is finished? (For example, variables modified by A, or variables that are semantically connected to A's local variables.)

- If yes, the user is asked to fill in a table with two columns: the first column is the name of the variable, v, and the second is a state predicate use_v describing the state of the woven system until which the value of v should be preserved. For example, for variant 2 of the Logging aspect, that logs messages as they are sent to the server, the message should not change between the moment it has been logged and the moment it is actually sent. Thus the use_v predicate will describe the moment of actual sending of the message (see Section 4 for more details). After the table is filled out, for each variable v with state predicate use_v in the table, we add the following CA (for "Change After") statement to the assumption of A:

  $$CA(v) = \mathsf{G}[(asp\_ret(A) \wedge v = V) \rightarrow (v = V \;\mathsf{W}\; (use\_v \wedge v = V))]$$

  where V is a logical variable keeping the value of v as it was at the end of the execution of A's advice.
- If there are no variables in the list, nothing is added to the specification of A at this step.

Step 3. In this step we construct requirements corresponding to the "invalidation before" case in Section 2. Before the problem of common join-points in modular verification was considered, there existed an implicit assumption that all the arrival join-points of an aspect are its actual join-points. But when a join-point might be shared, this is not necessarily so, because the join-point can be invalidated; thus an additional explicit assumption of this possibility is needed. The user is asked the following question:

Q. 3: Does it have to be that each time an arrival join-point of A is reached, A is eventually executed at it? That is, is it an error if previously executed aspects invalidate the condition for A's application?

- If no, nothing is added to the assumption of A in this step.
- If the answer was "yes", the following IB (for "Invalidation Before") statement is added to the assumption of A:

  $$IB \triangleq \mathsf{G}[at(ptc) \rightarrow (ptc \;\mathsf{W}\; (after\_prev\_asp(A) \wedge ptc))]$$

Step 4. The goal of this step is to enable the verification process to treat the case of "invalidation after" from Section 2. We ask the user the following questions:

Q. 4.1: Does the reason for a state to be A's join-point lie in the future of the computation? That is, does A's pointcut descriptor refer to any event following the join-point? For example, is the advice of A a "before" advice?

- If no, nothing is added to the assumption of A in this step.
- If the answer was "yes", the next question is asked:

Q. 4.2: Is it an error if the advice of A is performed, but the presumably-following event does not follow? (For example, because the future computation was changed by other aspects)

- If the answer is "no", nothing is added to A's assumption in this step.
- If the answer is "yes", the user is required to provide a state predicate, *foll_event*, meaning that the desired following event has just occurred. The user is then prompted to provide some optional restrictions on the values immediately after A's execution, the values at the moment the desired event occurs, and the connections between them (including, for example, value preservation). The restrictions should be given in the form of two predicates: *vals_after_asp* and *vals_at_foll_event*. The default value for both predicates is *true*.
- The following IA (for "Invalidation After") statement is then added to the assumption of A:

$$IA \triangleq \mathsf{G}[(asp\_ret(A) \wedge vals\_after\_asp) \rightarrow F(foll\_event \wedge vals\_at\_foll\_event)]$$

After the above automatic modifications, the specification constructed both captures the requirements of the user regarding the desired effect of aspect application, and contains sufficient assumptions to make the modular verification results applicable to systems with aspects sharing join-points.

3.2 Influence on aspect modeling

For the purpose of automatic modular verification of aspects ([9, 11]) and interference detection ([10]), the following corrections to the modeling process are performed in order to obtain a correct weaving of advice models into the generic representations of suitable base systems:

- As follows from the discussion in Step 3 of Section 3.1 (treating the case of "invalidation before"), the pointcut definition of A should be refined to be $ptc' = ptc \wedge after\_prev\_asp(A)$, so that ptc' marks actual and not arrival join-points of A, because only at these points the advice of A is now executed. This change of the aspect model is done automatically.
- In order to model an aspect with possibly shared join-points, we need to be able to model returning of the advice to the join-point from which its execution started, so that the same advice will not be applied again at this point, but the other aspects will be able to execute. When several aspects can share a join-point, the weaver has to give them all exactly one chance to be applied at it. This can be viewed as fulfilling a promise to each one of these aspects. Thus a flag *promise_ful(A)* is added to the variables of the weaver for each aspect A.

3.3 Full Specification and Verification Process

Given a library of aspects, two things are important for its usage: correctness of each aspect alone with respect to its assume-guarantee specification, and interference detection among the aspects. The question of possibly shared join-points is already important when the specification of individual aspects is defined. At this stage one of the tools for detection of potential interference at common join-points detection can be run, e.g. [1], and only if a potential interference is detected the specification refinement described in Section 3.1 has to be performed for the potentially interfering aspects. (If no tool for potential interference detection is run, all the aspect specifications should undergo the process from Section 3.1, to ensure the soundness of the verification process.)

After all the aspects in the library are specified and augmented as described above, existing tools for modular aspect verification ([9, 11]) and interference detection ([10]) should be run. Modularity of verification here means that the correctness of the individual aspects and of their combinations is verified independently of any concrete base system, thus dividing the whole verification process into two independent parts: whenever an aspect, or a collection of aspects, are to be actually woven into a base system, one part of verification is to check that the base system satisfies the assumptions of all the aspects given, and the other part is to ensure that all the aspects are correct w.r.t. their assume-guarantee specifications, and do not interfere. Such a modularity enables us to check the correctness and interference-freedom of the library of aspects off-line and once and for all, and not each time some aspects are actually woven into a given base system, thus the verification effort is very much reduced. Another advantage of modular verification is that the models verified are smaller, as we never need to actually examine a woven system, and this enhances the model-checking process (and sometimes even makes it possible).

4. EXAMPLES

We illustrate our analysis and verification approach on a collection of aspects that can be a part of a communication-aspects library. The aspects presented here are applicable for systems with message-passing, and they are variants of the aspects used as an example in [1]. They are also mentioned in Section 2.

Logging aspect (L) logs the message - sending in the system. As described earlier, there are four variants of the logging aspect in the library:

L_1: Logging all the sent messages as the user originally attempted to send them.

L_2: Logging all the messages that were actually sent to the server (the message is logged as it was sent).

L_3: Logging the frequency of message sending.

L_4: Logging all the attempts to send a message (the message is logged as it was originally attempted to be sent by the user).

Now we need to construct the specifications for the above variants of Logging aspect. The following predicates and variables definitions will be used in the construction:

- *msg_to_send*: a predicate which is true when a message is about to be sent, that is, when message sending is attempted. That is the moment before the message-sending procedure is actually called, and the parameters to the method call are represented by the variables *msg_c* and *msg_t*, containing the two parts of the message to be sent: the contents and the creation time, respectively.

- *msg_send*: a predicate which is true at the moment a message (with contents *msg_c* and creation time *msg_t*) is sent.
- *in_log* (< *str* >) : a predicate that is true if the string "str" appears in the log.

The pointcut of all the variants of the aspect is the moment before the message-sending procedure is called. More formally, $ptc = msg\_to\_send$. The guarantees of the aspects emerge in the usual way from the purpose of each of them, and are written more formally below. If there would be no possibility of sharing join-points, the assumption of all the logging aspect variants would be that if a message is sent, there indeed was an attempt to send this message (i.e., this very same message has been passed as a parameter to the message-sending procedure). More formally,

$$P_L \triangleq \mathsf{G}([msg\_send \wedge msg\_c = C \wedge msg\_t = T] \\ \rightarrow \mathsf{O}[msg\_to\_send \wedge msg\_c = C \wedge msg\_t = T])$$

However, we are aware of the possibility of each of the aspects to share a join-point with other aspects, such as Encryption and Authorizatoin, thus additional assumptions for the aspects are constructed according to the procedure from Section 3.1.

Specification for L_1*:.* A possible guarantee for L_1 is:

$$R_{L1} \triangleq \mathsf{G}([at(msg\_to\_send) \wedge msg\_c = C \wedge \\ msg\_t = T \wedge F(msg\_send)] \\ \leftrightarrow [\mathsf{F}(in\_log(< X, T >))])$$

meaning that messages that appear in the log are all the sent messages, but as they were first attempted to be sent by the user. (Note that the fact that each message is accompanied by creation time information ensures a one-to-one correspondence between messages and lines in the log.)

The answers for the assumption-construction questions for L_1 are as follows:

Q.1: "Yes". The aspect depends on the contents and time information of the message as they were at the join-point, thus the values of the variables *msg_c* and *msg_t* should be preserved. Thus, substituting into the template $CB(v)$, the following statements are added to the assumption of L_1:

$$CB(c) = \mathsf{G}[(at(msg\_to\_send) \wedge msg\_c = C) \rightarrow \\ ((msg\_c = C) \\ \mathsf{W}\ (after\_prev\_asp(L1) \wedge msg\_c = C))]$$

and

$$CB(t) = \mathsf{G}[(at(msg\_to\_send) \wedge msg\_t = T) \rightarrow \\ ((msg\_t = T) \\ \mathsf{W}\ (after\_prev\_asp(L1) \wedge msg\_t = T))]$$

Q.2: "Yes". The time information of the message should be kept intact till the moment the message is actually sent. There is one entry in the table: the variable *msg_t*, matched by the *msg_send* predicate. Thus the addition to the aspect assumption at this stage is:

$$CA(t) = \mathsf{G}[(asp\_ret(L_1) \wedge msg\_t = T) \rightarrow \\ ((msg\_t = T)\ \mathsf{W}\ (msg\_send \wedge msg\_t = T))]$$

Q.3: "No". If the message will not be sent, it should not appear in the log, thus the advice of L_1 should not be applied for it. Nothing is added to the assumption of L_1 at this stage.

Q.4.1: "Yes". The advice of L_1 is a "before" advice.

Q.4.2: "Yes". It is an error if a message that is not sent and will not be sent appears in the log. The desired following event is the event of sending the message (defined by its creation time only, as that is what matters for the purpose of L_1). Thus $foll\_event = msg\_send$, $vals\_after\_asp = (msg\_t = T)$ and $vals\_at\_foll\_event = (msg\_t = T)$. Substituting into the IA template, we obtain the following addition to L_1's assumption:

$$IA = \mathsf{G}[(asp\_ret(L_1) \wedge msg\_t = T) \rightarrow \\ F(msg\_send \wedge msg\_t = T)]$$

Specification for L_2*:.* A possible guarantee for L_2 is:

$$R_{L2} \triangleq \mathsf{G}([\mathsf{F}(msg\_send\ \wedge msg\_t = T \wedge msg\_c = C)] \leftrightarrow \\ [\mathsf{F}(in\_log(< C, T >))])$$

meaning that a message appears in the log if and only if it has been, or will be, sent.

The construction of the assumption for L_2 is performed similarly to that of L_1, with only two differences: An additional variable, *msg_c*, should be preserved after the aspect finishes its computation (affecting the $CA(v)$ and IA statements), and no values from arrival join-point should be kept (making $CB(v)$ *true*). Together we obtain that the addition to the assumption of L_2 as a result of the guided specification construction procedure consists of the following statements:

$$CA(c) = \mathsf{G}[(asp\_ret(L_2) \wedge msg\_c = C) \rightarrow \\ ((msg\_c = C)\ \mathsf{W}\ (msg\_send \wedge msg\_c = C))]$$

$$CA(t) = \mathsf{G}[(asp\_ret(L_2) \wedge msg\_t = T) \rightarrow \\ ((msg\_t = T)\ \mathsf{W}\ (msg\_send \wedge msg\_t = T))]$$

and

$$IA = \mathsf{G}[(asp\_ret(L_2) \wedge \\ msg\_c = C \wedge msg\_t = T) \rightarrow \\ F(msg\_send \wedge msg\_c = C \wedge msg\_t = T)]$$

Specification for L_3*:.* A possible guarantee for L_3 is:

$$R_{L3} = \mathsf{G}([\mathsf{F}(msg\_send\ \wedge msg\_t = T)] \leftrightarrow \\ [\mathsf{F}(in\_log(< T >))])$$

meaning that the log contains all the creation-times of the sent messages.

The construction of the assumption for L_3 is almost the same as for L_2, except for the fact that the value of *msg_ts_c* need not be preserved after the aspect finishes its computation (thus giving the same $CA(t)$ and IA statements as for L_1). Thus the addition to the assumption of L_3 is

$$CA(t) = \mathsf{G}[(asp\_ret(L_3) \wedge msg\_t = T) \rightarrow \\ ((msg\_t = T)\ \mathsf{W}\ (msg\_send \wedge msg\_t = T))]$$

and

$$IA = \mathsf{G}[(asp\_ret(L_3) \wedge msg\_t = T) \rightarrow \\ F(msg\_send \wedge msg\_t = T)]$$

Specification for L_4*:.* A possible guarantee for L_4 is:

$$R_{L4} \triangleq \mathsf{G}([at(msg\_to\_send) \wedge msg\_c = C \wedge msg\_t = T] \\ \leftrightarrow [\mathsf{F}(in\_log(< C, T >))])$$

meaning that the log contains exactly the messages attempted to be sent by the user.

The construction of the assumption for L_4 is almost the same as for L_1, with the following differences only:

- The answers to Question 2.1 and Question 4.1 are negative, as the logged message does not have to be sent, so $CA = true$ and $IA = true$ in this case.
- The answer to Question 3 is positive, as all the message sending attempts should be logged, including those aborted because of authorization failure.

Thus the additions to the assumption of L_4 are:

$$CB(c) = \mathsf{G}[(at(msg\_to\_send) \wedge msg\_c = C) \rightarrow \\ ((msg\_c = C)\ \mathsf{W} \\ (after\_prev\_asp(L_4) \wedge msg\_c = C))]$$

$$CB(t) = \mathsf{G}[(at(msg\_to\_send) \wedge msg\_t = T) \rightarrow \\ ((msg\_t = T)\ \mathsf{W} \\ (after\_prev\_asp(L_4) \wedge msg\_t = T))]$$

and

$$IB = \mathsf{G}[at(msg\_to\_send) \rightarrow (msg\_to\_send\ \mathsf{W} \\ (after\_prev\_asp(L4) \wedge msg\_to\_send))]$$

Encrypting aspect (E) is responsible for encrypting messages before sending. E should guarantee that each time a message is sent, it is encrypted. In fact, there is more to E: each time a message is received, it is decrypted. But this part is irrelevant to our example, so we'll ignore it here. E's guarantee can be written as:

$$R_E \triangleq \mathsf{G}(msg\_send \rightarrow encrypted(msg\_c))$$

where the predicate *encrypted(msg_c)* means that the contents of the sent message are encrypted. The assumption of E, constructed by the procedure in Section 3.1, emerges from the fact that the encrypted message value should be preserved till (and if) it is actually sent:

$$P_E = CA(msg\_c) = \mathsf{G}[(asp\_ret(E) \wedge msg\_c = C) \rightarrow \\ (msg\_c = C\ \mathsf{W}\ (msg\_send \wedge msg\_c = C))]$$

Authorization aspect (A) ensures that a message is sent to the server only if the current user has the needed permissions to communicate with the server. A's guarantee can be

$$R_A \triangleq \mathsf{G}(msg\_send \rightarrow permit\_usr\_send)$$

where the predicate *permit_usr_send* means that the user has enough permissions to send the message. When constructing the assumption of A, all the answers to the questions asked happen to be negative, thus A does not need to assume anything about the base system, and we can take

$$P_A \triangleq true$$

After the aspects are specified as above, if the usual verification procedure is applied, several cases of interference will be detected, as shown in Figure 1. A cell in the table corresponding to aspects pair $< M; N >$ can have one of the following values:

- "—" means that there is no interference when weaving first M and then N into any appropriate system;
- "X" - if the check is irrelevant, for example, we do not check interference among the aspect and itself. In our case we also do not check interference between different variants of the logging aspect, because we assume that only one of these aspects is woven into a system each time.
- Otherwise, there is interference among the two aspects if M is woven before N, and the cause of the interference is written in the cell, according to the classification from Section 2: "CB" stands for Change Before, "CA" - for Change After, "IB" - for Invalidation Before, and "IA" - for Invalidation After.

second / first	E	A	L1	L2	L3	L4
E	X	---	CB	---	---	CB
A	---	X	---	---	---	IB
L1	---	IA	X	X	X	X
L2	CA	IA	X	X	X	X
L3	---	IA	X	X	X	X
L4	---	---	X	X	X	X

Figure 1: Interference checks summary.

For example, the cell $< E, L_1 >$ is marked by CB, meaning that the Encryption aspect, if woven first, invalidates the assumption of the first variant of Logging, and that the violated part of the assumption is related to the "change before" case from Section 2: changing parameter values between arrival and actual join-points of L_1. And the cell

$< L_1, A >$ is marked by IA as the Authorization aspect, when woven after the first variant of Logging, invalidates the part of Logging specification related to the “Invalidation After” case from Section 2: removing a join-point of the aspect after its advice has already been applied.

5. CONCLUSIONS

This paper has concentrated on a problematic issue of aspect semantics: the possible interference that can arise from shared join-points. As an aid to programmers, an interactive semi-automatic augmentation of the specification is suggested. The questions asked and the results of formal verification should help the user understand the fine points of such interactions, and how they could affect the correctness of their aspect systems.

6. REFERENCES

[1] M. Akşit, A. Rensink, and T. Staijen. A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In *AOSD*, pages 39–50, 2009.

[2] L. Bergmans. Towards detection of semantic conflicts between crosscutting concerns. In *AAOS Workshop at ECOOP'03*, 2003.

[3] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new Symbolic Model Verifier. In *CAV'99*, LNCS 1633, pages 495–499. Springer, 1999. `http://nusmv.itc.it`.

[4] Curtis Clifton and Gary T. Leavens. MiniMao$_1$: Investigating the semantics of proceed. *Science of Computer Programming*, 63(3):321374, 2006.

[5] Simplice Djoko Djoko, Rémi Douence, and Pascal Fradet. Aspects preserving properties. In *PEPM*, pages 135–145, 2008.

[6] R. Douence, P. Fradet, and M. Sudholt. Composition, reuse, and interaction analysis of stateful aspects. In *Proc. of 3th Intl. Conf. on Aspect-Oriented Software Development (AOSD'04)*, pages 141–150. ACM Press, 2004.

[7] Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In *GPCE*, pages 173–188, 2002.

[8] Pascal Durr, Lodewijk Bergmans, and Mehmet Aksit. Reasoning about semantic conflicts between aspects. In *ADI'06*, pages 10–18, 2006.

[9] M. Goldman and S. Katz. MAVEN: Modular aspect verification. In *Proc. of TACAS 2007*, volume 4424 of *LNCS*, pages 308–322, 2007.

[10] E. Katz and S. Katz. Incremental analysis of interference among aspects. In *FOAL '08*, pages 29–38. ACM, 2008.

[11] E. Katz and S. Katz. Modular verification of strongly invasive aspects. In *Languages: From Formal to Natural*, volume 5533 of *LNCS*, pages 128–147. Springer, 2009.

[12] Günter Kniesel. Detection and resolution of weaving interactions. *T. Aspect-Oriented Software Development*, 5:135–186, 2009.

[13] I. Nagy, L. Bergmans, and M. Aksit. Declarative aspect composition. In *Software Engineering Properties of Languages and Aspect Technologies (SPLAT) Workshop*, 2004.

[14] István Nagy, Lodewijk Bergmans, and Mehmet Aksit. Composing aspects at shared join points. In *NODe/GSEM*, pages 19–38, 2005.

[15] Dong Ha Nguyen and Mario Südholt. VPA-based aspects: Better support for aop over protocols. In *4th IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, pages 167–176, 2006.

[16] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *T. on Programming Languages and Systems*, 26(5):890–910, 2004.

[17] N. Weston, F. Taiani, and A. Rashid. Interaction analysis for fault-tolerance in aspect-oriented programming. In *MeMoT'07*, pages 95–102, 2007.