Implementing COOL in JAMI

Steven te Brinke
s.tebrinke@student.utwente.nl

ABSTRACT
JAMI aims to be a generic aspect interpreter framework which can be used to prototype a wide range of AOP languages. However, only toy AOP languages have been implemented in JAMI. One of these toy AOP languages is a small subset of the coordination language COOL. By extending the COOL implementation, we implement a complex, existing AOP language to show that JAMI really supports a wide range of AOP languages. During the implementation, some problems were encountered, of which the solutions are discussed in this paper. Using eight test cases, we identified the proper working of our implementation. We conclude that JAMI is a framework that allows an easy implementation of many DSALs.

Keywords
AOP, Aspect-oriented programming, COOL, Domain specific aspect languages, JAMI

1. INTRODUCTION
Aspect-Oriented Programming (AOP) is a programming paradigm that makes it possible to express crosscutting concerns in a modular way. Examples of crosscutting concerns are access control, caching and synchronization. Domain specific aspect languages (DSALs) are AOP languages which focus on a particular concern. COOL is a DSAL for the specification of synchronization. JAMI is a framework for prototyping DSALs which aims to support a wide range of DSALs. Aspects can be automatically combined into an executable by an aspect weaver, or can be combined at run time by an interpreter. JAMI offers such an interpreter. At present, only toy DSALs have been implemented in JAMI. One of these DSALs is a small subset of COOL. To validate the suitability of JAMI for implementing a wide range of DSALs, a complex, existing language should be implemented. Thus, we extend the current COOL implementation.

2. BACKGROUND
2.1 JAMI
JAMI is an aspect interpreter framework that can be used to prototype domain specific aspect languages (DSALs). The common runtime platform provided by JAMI offers the possibility to combine several DSALs within a single application. JAMI aims to be a generic framework which allows a wide range of AOP languages to map to this common platform. Because it is assumed that DSALs will be developed independently, DSALs that are unaware of each other can also be combined within a single application. However, the behaviour of the resulting program could vary according to the execution order of the different DSALs. Therefore, JAMI offers a uniform constraint model that facilitates ordering constraints within as well as between DSALs.

The most important concepts of JAMI are: joinpoints, pointcuts, advice, selector-advice-bindings and aspects. A joinpoint represents a point in the control flow of a program where the execution of the program is intercepted. A pointcut matches a set of joinpoints, based on contextual properties. An advice defines the behaviour of an aspect. A selector-advice-binding connects a pointcut to an advice. An aspect groups multiple selector-advice-bindings.

Listing 1 Example of COOL for a bounded buffer

```
coordinator BoundedBuffer {
  selfex put, take;
  mutex {put, take};
  cond full = false, empty = true; // sync. state
  put: requires !full;
    /guard
    on_exit {empty = false;
      if (usedSlots == array.length)
        full = true;
    }
  take: requires !empty;
    /guard
    on_exit {
      full = false;
      if (usedSlots == 0) empty = true;
    }
}
```

2.2 COOL
COOL is a language for dealing with self- and mutual-exclusion, synchronization state, guarded suspension and notification, which we explain through the example in listing. This example shows the COOL code for a bounded buffer. The coordinator is specified for all instances of the class BoundedBuffer. In these instances, the methods put and take are self-exclusive, thus can be executed by at most one thread at a time. The methods put and take are also mutually exclusive, thus they cannot be executed concurrently by different threads. The coordinator also contains two condition variables, full and empty, that contain the synchronization state. Based on this synchronization state, the guards perform guarded suspension and notification. For example, the first guard specifies that a put action will be suspended until the buffer is not full. After a put action, the buffer is not empty any more and might become full, if all slots are used. The on_entry and on_exit blocks are called method managers.
Apart from condition variables, which are boolean, a coordinator may have ordinary variables, which can have any primitive type. Thus, ordinary variables can record much more data, but guards can be specified on condition variables only.

In [2], COOL is fully specified, including a source code weaving implementation in Java. According to Kojarski and Lorenz [3], COOL is expressive, concise, readable, and easy to understand; expressing the same concern in a general purpose aspect language like AspectJ [6] requires more lines of code and is harder to explain and understand.

Listing 2  Example of COOL for a buffer

```java
coordinator Buffer {
    selfex put, take;
    mutex {put, take};
}
```

2.3 The current implementation of COOL in JAMI

The subset of COOL already implemented in JAMI is very small. Only self- and mutual-exclusion are supported and inheritance of coordinated classes does not work as desired. This section describes the already implemented mapping to JAMI.

In JAMI, each COOL coordinator is modelled as an aspect. This aspect contains two selector-advice-bindings for each method involved in the coordination. One binding will be executed upon entering, the other one upon leaving the method. As an example of mapping COOL to JAMI, we use an unbounded buffer as shown in listing 2. The selector-advice-binding for entering the method put is shown in figure 1. It matches only joinpoints that call a method named put of objects of the type Buffer when the joinpoint type is MethodCall. Before the method call is executed, the advice EnterSyncedContextAction is executed. This advice contains an exclusion set containing all methods that block this method call if another thread is active within that method. This set is calculated using the selfex and mutex specifications in the coordinator. The described mapping contains all features of the current implementation of COOL in JAMI.

3. PROBLEM STATEMENT

So far only toy examples of DSALs are implemented in JAMI. Thus, it is not known whether JAMI has good support for a wide range of aspect languages. Also, the research on the composition of multiple advices is difficult with only small DSALs implemented. We validate JAMI by implementing a complex, existing language to see if it can actually be implemented and if that can be done in a convenient way. Thus, without basically working around the framework all over the place. This results in the problem statement: Does JAMI provide a good framework for the implementation of advanced DSALs?

3.1 Research questions

Because COOL is an advanced and well-known DSAL, this language is chosen to be implemented. This results in the following research questions:

1. Does JAMI provide a good framework for the implementation of advanced DSALs?

4. RELATED WORK

In [7], the Awesome framework is described, including an implementation of COOL in this framework. The Awesome framework uses a weaver based approach instead of an interpreter based approach like JAMI. Thus, the implementation of COOL in Awesome is quite similar to the source code weaving presented by Lopes [2].

XAspects [8] implements a system to map DSALs, including COOL, to AspectJ source code. This mapping is source code weaving based.

Thus, currently there does not exist an implementations of COOL that is interpreter based.

5. EXTENDING THE COOL IMPLEMENTATION

5.1 Aspects

Every coordinator will become an aspect containing the coordinator variables, information about the method states and two selector-advice-bindings for every coordinated method.

Aspects have the possibility to create data fields with certain instantiation policies, for example per object, per class and singleton. When a per object data field is created, every coordinated object will have its own value. This is exactly how the variables of a per object coordinator work, thus these are implemented as per object data fields. A per class coordinator has only one value for every data field, even if it is coordinating multiple classes, thus this one will use singleton data fields for its variables.
A method state records which threads are executing a method. Thus, it knows how often every thread is executing the method. A method state is recorded in a MethodState object, like the one specified by Lopes [2]. Every aspect has a registry containing these method states. This registry is contained by a data field using the same instantiation policy as described above.

For every coordinated method, the aspect contains two selector-advice-bindings. The structure of such a binding is the same as explained in section 2.3. As described in the following sections, some changes are made to the joinpoint selectors and the advices are extended with quite some functionality.

5.1.1 Joinpoint selectors
Every method has a jointpoint selector as shown in figure 2. A SelectByObjectSuperType selector is used to select not only objects of type Buffer, but also any subclass thereof. In a multi class coordinator, the selected object type differs per method. Methods are selected by signature because names might be ambiguous. The reasons for these changes are explained in sections 5.3 and 5.2.

5.1.2 Advices
5.1.2.1 Entering a method
When thread \( t \) tries to enter a method \( m \), \( t \) should be blocked until the requirements and exclusion set of \( m \) are satisfied. As soon as these are satisfied, the on entry method managers should be executed, \( t \) should be registered as running \( m \) one time less and \( t \) should actually enter \( m \). This is implemented in an advice called LeaveSyncedContextAction, as is shown in figure 3 and listing 4.

5.1.2.2 Leaving a method
When thread \( t \) leaves a method, the on exit method managers should be executed, \( t \) should be registered as running \( m \) one time less and \( t \) should actually exit \( m \). This is implemented in an advice called LeaveSyncedContextAction, as is shown in figure 3 and listing 4.

5.2 Method overloading
Lopes [2] has defined COOL on a subset of Java without method overloading. Because it is desired to support the use of coordinators on every Java program, we introduce support for method overloading. The only change is that a method reference should contain the formal parameters when the method is overloaded. For example, \( \text{put}(Object) \) or \( \text{put}(String) \) when both exist, because \( \text{put} \) is ambiguous in that case. This change is backward compatible with the original specification, because if no method overloading is used, a method reference without formal parameters clearly is unambiguous.

5.3 Inheritance of coordination

![Figure 2: Example of a JAMI joinpoint selector](image)

![Figure 3: Example of JAMI advices](image)

---

**Listing 3** Semi code for EnterSyncedContextAction

```java
while (another thread is executing a method ∈ exclusionSet
 \( \lor \) requires does not evaluate to true)
wait for notification

for (m : methodManagers)
execute m

register that the method is entered
```

---

**Listing 4** Semi code for LeaveSyncedContextAction

```
register that the method is left
for (m : methodManagers)
execute m

notify threads waiting in this coordinator
```

---

![Figure 4: Coordination on inherited classes](image)
We use the class structure shown in figure 5 to explain inheritance of coordination. We reference to class A as \(A\), class B as \(B\), coordinator A as \(A_C\) and coordinator B as \(B_C\).

The characteristics of coordination defined by the original COOL specification are:

- Any subclass of \(A\) is coordinated by \(A_C\), except the classes for which a new coordinator is specified and all their subclasses.
- All explicit super calls in \(B\) are uncoordinated, that is they use neither \(B_C\) nor \(A_C\), because the method containing the super calls is already coordinated.

JAMI joinpoint selectors are unaware of the existence of other selectors. Thus, coordination cannot be cancelled by the existence of other selectors and it is only possible to coordinate none or all subclasses. To achieve the desired behaviour, we use a constraint model. The model itself and how we use it in our implementation are explained in the next sections.

### 5.3.1 Constraint model

Nagy [9] has specified a generic constraint model to solve the issues caused by multiple advices matching the same joinpoint. This model distinguishes three categories of constraints: structural constraints, ordering constraints and control constraints.

Structural constraints specify which advices have to be or cannot be mutually present at a shared join point. Violation of such a constraint will result in throwing an exception. For example, \(\text{exclude}(x, y)\) specifies that if \(x\) is present, \(y\) should be absent and also if \(y\) is present, \(x\) should be absent. If at any point during execution as well as \(x\) matches a certain joinpoint, an exception is thrown.

Ordering constraints specify a partial order for the execution of advices. For example, \(\text{pre}(x, y)\) specifies that \(x\) should be executed before \(y\).

Control constraints specify conditional execution of advices. For example, \(\text{cond}(c, y)\) specifies that \(y\) should be executed only if \(c\) evaluates to true.

The constraint model implemented in JAMI is based on the model specified by Nagy [9]. The ordering constraints and control constraints are part of the JAMI constraint model, but the structural constraints are not.

### 5.3.2 Constraints

Using the constraint model of Nagy [9], a constraint to cancel the execution of \(A_C\) when \(B_C\) is present can easily be specified as \(\text{cond}(\neg B_C, A_C)\), where \(\neg B_C\) should be read as \(B_C\) is not present. However, this might cause problems with multi class coordinators, as shown in figure 5. Because \(C\) is a superclass of \(D\), \(C-F\) will be cancelled by \(D-E\) and because \(E\) is a superclass of \(F\), \(D-E\) will be cancelled by \(C-F\). Thus, \(D-E\) and \(C-F\) cancel each other. In JAMI, constraints are specified on selector-advice-bindings. This offers a finer grained constraint model that allows us to specify the desired constraints, because of the possibility to let \(D-E\) cancel only the selector-advice-bindings of \(C-F\) applying to \(C\) and not the ones applying to \(F\).

Figure 5: Coordination on multiple inherited classes

Every coordinator has a binding for every coordinated method, thus the set of constraints to cancel every binding of \(A\) when a binding of \(B\) is present is: \(\{\text{cond}(\neg B_m, B_m) \mid m \in \text{methods}(A), b_{m,A} \in \text{bindings}(m, A), b_{m,B} \in \text{bindings}(m, B)\}\). This set of constraints is correct under the assumption \(\forall m \in \text{methods}(A)\), \(\text{bindings}(m, A) \neq \emptyset \implies \text{bindings}(m, B) = \emptyset\). Thus, every method that has a binding in \(A\) should also have a binding in \(B\). However, for a method that is coordinated in \(A\), but not in \(B\), clearly \(B\) does not need to have an advice. To ensure the assumption holds, an empty advice should be created when a method that is coordinated in the superclass, is not coordinated any more. An empty advice will never block because there are no conditions, thus this will not result in any other behavioural difference.

### 5.3.3 Super calls

Detecting if a call is a super call is quite difficult. For example, \(A\) defines a method \(a\) and \(B\) does not define this method, thus \(a\) is implemented in \(A\) only. In Java, there is no behavioural difference in calling \(a\) or \(\text{super}:a\) from a method of \(B\), because the implementation is only present in \(A\). However, in COOL the first call is a coordinated one but the second is not. This behaviour can be implemented only if the difference between the two calls can be detected. Because the only difference is the strategy Java uses to locate the method to invoke [10], JAMI cannot distinguish super and virtual method calls. Further research is necessary to identify if it is actually possible to distinguish super and virtual method calls. Currently, it cannot be done, making it impossible to implement the exact behaviour specified by COOL, thus we propose a small change in behaviour.

We propose to coordinate all super calls in \(B\) with \(B_C\). This leads to the next behavioural changes in comparison with the original specification of COOL:

- A call from method \(m\) to \(\text{super}:m\) is coordinated in the same way as recursion. Because the thread already has permission to run \(m\), the call will not be blocked. Thus, the only behavioural difference is that the method managers are called again.
- A call from method \(m\) to \(\text{super}:n\) will be a coordinated one. Thus, the method managers of \(n\) are called and \(n\) might be suspended, but according to the original specification no coordination should be performed.

The first one only leads to different behaviour when a method manager records its own history, for example by counting how
often it is called. The second one only introduces more coordination, thus exclusion constraints will not be violated by this change, but progress constraints might be violated.

The reason super calls are rewritten in the implementation by Lopes is to ensure a call to \( B \) will never be coordinated by \( A \). Our implementation achieves the same. The behavioural differences between Lopes and our implementation only contain characteristics that are not discussed in their detail by Lopes. Thus, we assume these changes have no effect on the quality of the DSAL by itself, the only negative effect is the incompatibility with the original COOL specification.

5.4 Field access
Every coordinator should have access to all fields of the coordinated class, even the private ones. We use reflection to resolve all field references to actual fields. These references are resolved when loading the coordinator, thus a coordinator can only be loaded when all coordinated classes are available. In the source code weaving implementation of Lopes, the coordinated classes only have to be available at compile time, at run time a multi class coordinator can be loaded when some coordinated classes are absent. In our implementation this is not possible because an interpreter based approach does not have these separate steps.

By default, private fields are not accessible by the interpreter. However, we will call `setAccessible(true)` on any field that is not accessible. This is possible only when the security manager allows it, but when Java is run from the command line the default security manager does not allow it.

6. IMPLEMENTATION

6.1 General approach
Based on the grammar specified by Lopes and the Java grammar, we have specified a complete grammar for COOL. The grammar specified by Lopes accepts more expressions than the actual possible ones, and she leaves it to the Java compiler to catch these accepted but invalid expressions. Because we do use our own compiler only, we should specify a fully correct grammar. Any Java programmer will expect COOL expressions to work exactly like Java expressions, except for the parts explicitly left out. Thus, we specified a grammar that should achieve this. However, the Java language is so large that some details might have slipped through our fingers.

Our focus was supporting all concepts of COOL to show that the full language can be implemented in JAMI. Because of time constraints, we had to leave some details out of the implementation. We only left out expressions if a similar expression was already implemented. For example, we left greater than or equal (\( \geq \)) out, but implemented equality (\( = \)). Thus, the first one can be implemented like the latter. The only difficulty will be the implementation of the semantics of the operation itself, as specified in the Java language. However, the implementation of the semantics involves only one function that does not interact with JAMI. Thus, leaving it out has no influence on our research. All expressions we left out are commented out in the source code. This way, it is easy to see what is left out and complete the implementation.

After parsing, the type checker will check the AST and throw exceptions if something is incorrect. It will also resolve references and annotate the AST with extra data that the translator will use. Because the type checker does not interact with the JAMI framework, it was not part of our focus. We did not optimize its output for readability and probably some details will not be checked correctly. Thus, to make the use of COOL really user-friendly, the type checker needs quite some improvements. However, this can be done by changing the type checker only. Thus, no interaction with JAMI is required and it has no influence on our research.

When type checking has been done, the translator will generate a `Coordinator` object. This coordinator object contains its granularity, per class or per object, which classes it coordinates, which variables are declared and for every coordinated method a `MethodCoordinator`. A `Coordinator` can be translated to the JAMI metamodel directly. By doing so, an aspect will be generated as specified in section 5.1.

The constraints should be specified with knowledge of multiple coordinators. We have extended the `MetamodelBuilder`, the base class for any JAMI program, to `COOLMetamodelBuilder`, which facilitates the generation of constraints. After adding all coordinators to the `COOLMetamodelBuilder`, it will generate all desired constraints including the necessary empty advices.

### Listing 5 Example of using COOL
```java
class COOLExample extends COOLMetamodelBuilder {
  public void run() {
    FileToCoordinator reader =
        new FileToCoordinator("BoundedBuffer.cool");
    reader.loadCoordinator();
    addCoordinator(reader.getCoordinator());
    initCoordinator();

    builderInitialized();
    // The actual program should be inserted here
  }
}
```

An example of loading `BoundedBuffer.cool` is shown in listing 5. Firstly, the coordinator is loaded by a `FileToCoordinator` object. The loading involves parsing, checking and translating. Secondly, the loaded coordinator is added to the metamodel builder. These two steps are repeated until all coordinators are added, in our example we will use only one coordinator. Finally, `initCoordinator()` will ensure all aspects and constraints are initialized. Now, all COOL specific actions are done and we can proceed with calling `builderInitialized()` and running the actual application.

Our complete implementation is available at the JAMI website: [http://jami.sf.net](http://jami.sf.net)

6.2 Design choices

6.2.1 COOL specific metamodel elements
JAMI does provide a common metamodel for languages implemented using this framework. However, as can be seen in figure 3 we use our own metamodel for guards and method managers. This is because the execution of these guards and method managers should be synchronized and composing advices in JAMI does not offer this possibility. Because COOL is designed to be able to handle all coordination, it is unlikely that other DSALs
will also use synchronization in advice execution. Thus, we assume this problem is specific to COOL and do not extend JAMI to support synchronized advice composition. However, it makes our implementation of COOL not as generic as desired. When implementing other DSALS in JAMI, similar problems might be encountered.

6.2.2 Multiple coordinators on a single class
COOL does not prevent specifying multiple coordinators on a single class. However, Lopes [2] does not specify the desired behaviour in this case. Our implementation will work with multiple coordinators on a single class, as long as there is only a single coordinator for every method. A method with multiple coordinators will also have the desired exclusion constraints. However, it might cause unexpected deadlocks, because the different coordinators are executed in unspecified order.

We assume it is not really desired to have multiple coordinators on a single method because this decentralizes the synchronization code. Thus, we propose to issue a warning in such a case.

6.2.3 Changes to COOL
For our implementation, some small changes are made to COOL. These changes are the addition of method overloading (section 6.2.3) and a slightly different behaviour of super calls (section 5.3.3). The addition of method overloading is an extension of COOL that is fully compatible with the original specification. But super calls have more coordination in our implementation than in the original COOL specification, which might result in some incompatibilities. The first change is defined in terms of the Java language. Thus, any other implementation based on Java will have the same problem to resolve. As a result, the need for this change does not imply any shortcomings of JAMI, but is a result of the implementation language chosen for JAMI. However, the second change is because JAMI does not detect the small difference between super and virtual method calls that is present in Java. Assuming that this difference is detectable, it is a small shortcoming of JAMI.

6.3 Solutions to implementation problems
The problems encountered while implementing COOL are documented in this section. Because changes to JAMI are preferred over changes to COOL, we will describe the solutions in terms of JAMI. The few changes we made to COOL are already explained in section 6.2.3.

6.3.1 Select by object super type
JAMI did not contain a SelectByObjectSuperType selector. Thus, we have written this selector as an addition to JAMI. The implementation was straightforward because the joinpoint model is designed for this kind of selectors. The problem is that JAMI only contains a small set of selectors, thus extensions will probably be necessary. However, these can be done without changing any existing code.

6.3.2 Methods throwing exceptions
After executing a coordinated method, the synchronized context should be left, no matter if the method returns normally or by throwing an exception. Currently, JAMI does not have a joinpoint for the moment a method is left by throwing an exception. Thus, our implementation might have incorrect behaviour when used on methods throwing exceptions. Some changes should be made to JAMI to add this joinpoint to the framework. As soon as that is done, the only change that is necessary to make our implementation of COOL handle methods throwing exceptions correctly, is in the method coordinator replacing the method return selector by a method after selector.

6.3.3 Selecting methods by signature
Methods are matched by signature to allow the use of method overloading. However, the current implementation of SelectByMethodSignature does match the signature of the call instead of the method. Thus, when calling the method put(Object o) as put("A String") it is matched with the signature put(String) instead of put(Object). This has a large impact on the usefulness of our implementation because it is very common to pass subtypes to a method. However, we will not use this in our test cases, thus it has no influence on our research. The problem should be resolved by changing SelectByMethodSignature, because the current behaviour clearly is not the desired one for this selector.

6.3.4 The use of reflection
JAMI is implemented using AspectJ [6]. AspectJ weaves quite a lot of utility methods in any class to allow JAMI to catch all desired joinpoints. Because we use reflection on the weaved bytecode to determine which methods exist in a coordinated class, these utility methods are reported too. Thus, it is possible to coordinate these utility methods. As a solution, a way to identify these methods as utility methods could be offered, for example by annotating them, allowing them to be filtered away from the coordinatable methods.

6.3.5 Initial value of variables
Coordinator variables and conditions can have an initial value. However, the data fields we use to implement them, do not have the possibility to set an initial value. The possibilities for setting an initial value depend on the instantiation policy. A singleton field has the possibility to set the initial value. Thus, in a multi class coordinator, initial values are set correctly. However, a per object instantiation policy does not offer this possibility. It will construct a new object if the field is read before it is written. By default, this is done using the default constructor, but it is possible to define the constructor arguments. Constructor arguments are defined by a list of fields, of which the values will be used as constructor arguments. By using singleton fields, these fields can be set with an initial value. However, that only works well if the object has a copy constructor. Because all primitive types are wrapped in objects and these objects do not have a copy constructor, an initial value cannot be set this way. However, all these objects except Character do have a constructor with as argument a String. Thus, as a workaround we create a singleton variable with the initial value as a String and initialize all variables using this String. Because Character only has a constructor with as argument the primitive type char, there is no way to give a per object Character an initial value. Also for arrays an initial value cannot be given because an array object has no constructor. Further research is necessary to identify a way to make variable initialization more flexible.

7. PERFORMED TESTS
To test our implementation, we have written eight test cases. These test cases include the four examples given by Lopes [2], which cover all important points in the semantics of COOL. To ensure synchronization is necessary, the methods with synchronization constraints will sleep for some time. This way, the different
threads will run concurrently and it is very likely that synchronization is necessary. By reading the output of the test cases, we can identify when synchronization constraints are enforced. Because these test cases depend on timing, they are non-deterministic and it is impossible to test every possible run. However, our implementation is built using proven principles. Thus, we assume that we can conclude the correct working of our implementation from these test cases.

Of the eight test cases, seven test cases succeed. Only the test case regarding exceptions fails, as described in section 6.3.2. The problems described in the sections 6.3.3 and 6.3.5 are not covered by the test cases. Because the test cases cover the important semantics of COOL, we can conclude the proper working of our implementation, except for the described problems.

8. FUTURE RESEARCH

Selecting methods by signature does not work properly, as described in section 6.3.3. Further research is necessary to solve this problem.

Setting initial values of variables is not flexible enough, as described in section 6.3.5. Further research should identify a way to make it more flexible.

8.1 Automatically loading coordinators

In our implementation it is necessary to load all coordinators before the actual program is executed. It would be desired to load coordinators automatically when they are required. In order to do so, it should be possible to identify when the coordinators for a certain class should be loaded and the location to load them from.

8.1.1 When to load coordinators

The coordinators for class C are only necessary when a method of C is executed or if the coordinator of any subclass S of C is loaded, because in that case constraints should be generated. This can be achieved by loading the coordinator as soon as the classloader loads a class. Loading a class C is necessary to be able to execute any methods of C and because the classloader loads all superclasses of the loaded class, C will automatically be loaded when S is. Assuming the classloader will not load any class that remains unused, we will load only the necessary coordinators. Because loading a coordinator requires access to the coordinated class objects, the load should be done after the classload is complete and care should be taken with multi class coordinators. Further research is necessary to identify the possibilities of adding classload joinpoints to JAMI.

8.1.2 How to locate coordinators

Lopes [2] has defined a file name convention for coordinators like the one for Java classes: a coordinator for the class C is saved in the file C.cool. By putting all coordinators in the same directory as the Java class files, coordinators can be located easily. Because a coordinator is written with knowledge about the Java class, it seems to be a good requirement to use the same directory. Thus, locating single class coordinators is no problem.

Multi class coordinators are more difficult to locate. The file name should consist of all coordinated classes, separated by a dash. For example, a coordinator for the classes A and B is saved in a file called A-B.cool or B-A.cool. If the full qualified classnames are a.A and b.B, it might be desirable to use a.A-b.B.cool, because there could be more classes called A. The coordinator file can be saved in package a, b or the default package. Thus, locating multi class coordinators introduces some difficulties that need further research.

9. CONCLUSION

We have successfully implemented COOL in JAMI. This is shown by the tests we performed (section 7). These test cases include the four examples given by Lopes [2], which cover all important points in the semantics of COOL. Still some small problems exist in our implementation (section 6.3) and are left to be resolved by changes to JAMI. None of these changes will be structural changes to the JAMI framework:

• When a method returns by throwing an exception, it should be possible to match this return (section 6.3.2). This will be an addition to JAMI, not a change of its functionality.

• Changing SelectByMethodSignature (section 6.3.3) involves changing one selector.

• Setting the initial value of variables (section 6.3.5) might be troublesome. However, it is only a problem of certain instantiation policies.

Our implementation slightly deviates from the original specification of COOL (section 6.2.3), because of incompatibilities between the specification and Java. It might be possible to resolve one of these changes if JAMI can intercept the behaviour of Java in all its detail. We conclude that JAMI does not have any limitations that required changing important properties of our COOL implementation.

We had to define COOL specific metamodel elements (section 6.2.1), because COOL has synchronization requirements for its execution. This makes our implementation less generic than expected. Other DSALs will not have these requirements, but might also have some specific structures that cannot be mapped to JAMI directly and require quite some code to be written. However, we assume it is not possible to construct a framework that allows an easy mapping of any specific structure to the framework. JAMI does offer a framework that makes the common actions of matching joinpoints and the use of aspect variables very easy. This way, the implementation of a DSAL is simplified, although it might still be difficult. However, JAMI might not be able to perform all desired actions since currently only a small amount of selectors is available. This will probably be the most encountered problem while implementing other DSALs, but extensions are easy to write.

Even though our implementation of COOL is not as generic as anticipated, our implementation gives an indication how suitable JAMI is for implementing DSALs. Based on our experience, we expect that JAMI is a framework that allows an easy implementation of many DSALs.

REFERENCES


