SPECTACKLE: Toward a Specification-based DSAL Composition Process *

David H. Lorenz
Open University of Israel
lorenz@openu.ac.il

Oren Mishali
Open University of Israel
omishali@openu.ac.il

Abstract

DSAL composition frameworks are tools used in the process of composing multiple DSAL mechanisms into a single multi-DSAL weaver. The DSAL composition process involves first of all specifying the desired interactions between the DSAL mechanisms being composed, and then producing a multi-DSAL weaver based on the composition specification. However, the effectiveness and the usability of the DSAL composition process is hindered by the lack of tool support for specifying the composition, and by the remaining coding effort required to implement the specification and the individual DSALs in the composition framework.

This work describes an improved DSAL composition process that is based on an explicit specification of the composition and the individual mechanisms involved. A new tool named SPECTACKLE is defined for analyzing the composition and for specifying the desired interactions. Based on these specifications, a significant part of the code of the mechanisms and that of the multi-DSAL weaver are generated by the composition framework. The composition process is illustrated in the context of the AWESOME composition framework.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Frameworks; D.2.5 [*Software Engineering*]: Domain-specific—DSLs; D.3.4 [*Programming Languages*]: Processors.

General Terms Domain Specific Aspect Language (DSAL).

Keywords DSAL, Composition Frameworks, Specification-Based Process.

* This research was supported in part by the *Israel Science Foundation (ISF)* under grant No. 926/08.

1. Introduction

DSAL composition refers to a process in which a multi-DSAL weaver is composed out of individual DSAL mechanisms. The DSAL composition process begins with a *composition specification* that determines how the DSALs interact. The process ends when an implementation of a multi-DSAL weaver is capable of weaving code written in the multiple DSALs.

In this process, the role of a *composition designer* is to provide the composition specification. Once a specification is provided, the role of a *composition implementer* is to realize the multi-DSAL weaver according to the specification. The DSAL composition process is assisted by *composition frameworks* (such as AWESOME [3]). However, such composition frameworks are essentially implementation-oriented:

- composition frameworks do not assist the composition designer in formulating a desired composition specification; and
- coding effort required on the composition implementer part in order to realize the composition specification is not inconsiderable.

Currently, the composition framework mainly targets the composition implementer. We believe that in a realistic setting, where multiple DSALs are being combined, the specification of the composition should also be guided by composition frameworks. For instance, tools for querying basic composition properties and for investigating potential feature interactions should be a part of the composition framework.

Moreover, even the assistance that is currently provided by composition frameworks to the composition implementer leaves much to be desired. Typically, the specification is implemented imperatively and not declaratively. This makes the composition process less intuitive and demands greater coding effort. For example, in AWESOME, the composition configuration logic is coded by implementing an aspect in ASPECTJ (or several aspects, e.g., Figure 3). This is a significant improvement over a tangled crosscutting implementation, but is still a matter of writing code, not a specification.

Lastly, composition frameworks deal mainly with composition of existing mechanisms. The task of implementing the individual mechanisms remains outside their scope. Therefore, also here, due to a tool gap, it is difficult to implement an aspect mechanism. For instance, in AWESOME concrete JAVA methods should be defined that implement the weaving process of the mechanism. A declarative approach may be applied also here, to minimize the manual coding effort needed.

In this paper we report on ongoing efforts toward a specification-based DSAL composition process. The envisioned process is illustrated in the context of the AWESOME composition framework, and includes the following main features:

- **A formal composition specification.** The composition specification, which is in current state-of-the-art composition frameworks informal, is made formal.
- **An explicit description of an aspect mechanism.** The properties characterizing each aspect mechanism are kept in a dedicated machine-readable *manifest* file.
- **A tool for analyzing the composition.** A new tool named SPECTACKLE, reads the manifests and assists the composition designer in analyzing the composition and then in generating the composition specification.
- **A generative specification-based implementation.** The composition specification with the DSAL manifests are passed to AWESOME, and significant code parts of the individual aspect mechanisms and of the composition configuration are automatically generated.

The result is an improved DSAL composition process. The composition process becomes more effective and thus more affordable. Another advantage of a specification-based composition process is support for reasoning about the composition. The specification expresses the composition logic in natural and high-level terms, making it more accessible to the development team. This is important, especially when domain experts that are not comfortable with code-level reasoning are involved.

Outline. In this paper, the enhanced specification-based DSAL composition process is illustrated over AWESOME. Section 2 presents an example composition of three aspect languages that is used throughout the paper. In Section 3, we demonstrate the SPECTACKLE tool and illustrate how it helps the composition designer to analyze the example composition and to specify the desired DSAL interactions. In Section 4 we explain how AWESOME uses the produced specifications to simplify the implementation of the multi-DSAL weaver.

2. Example of DSALs

To concretely illustrate the benefit of a specification-based DSAL composition process, we briefly describe three aspect language that we may wish to combine. The languages are ASPECTJ [2], a general purpose aspect language; COOL [5], a DSAL that handles synchronization of JAVA methods; and VALIDATE [1], a simple DSAL that we have defined that supports validation of input parameters.

In ASPECTJ the modularization of crosscutting functionality is encapsulated in language constructs called *aspects*. COOL introduces aspect-like constructs called *coordinators*. A coordinator encapsulates synchronization logic of one or more JAVA classes. Within a coordinator, one may define dedicated constructs for handling the specific synchronization. For instance, a `selfex` declaration identifies the methods that can be executed by at most one thread at a time; a `mutex` declaration identifies a set of methods that cannot be executed concurrently by different threads. The following is a coordinator for a `Printer` class:

```
coordinator Printer {
  selfex sendJob, cancelJob;
  mutex {sendJob, cancelJob, getJob};
}
```

The methods `sendJob` and `cancelJob` update internal variables of an object, therefore they are guarded by both `selfex` and `mutex`. On the other hand, the method `getJob` is read only and defined only within a `mutex` declaration. As a result, several threads may activate `getJob` concurrently. However, execution of `getJob` by one thread prevents the execution of `sendJob` and `cancelJob` by other threads.

VALIDATE is a simple DSAL that supports validation of input parameters of methods, constructors and fields (field assignments). The input validation of a particular program element, e.g., a method, is contained in a validation *command*. Within a command, $(*i*) is used to access the *i*'th parameter. In addition, a library of predicates exists for defining the validation criteria. Validation commands for a specific class are grouped in a *validator* aspect. As an example, consider the following validator for class `Printer` which confirms that the argument for the method `getJob` is a positive integer.

```
  validate getJob(int id): $1 > 0;
}
```

3. The SPECTACKLE Tool

SPECTACKLE is a tool for analyzing a multi-DSAL composition as well as specifying the desired feature interactions. When initially tackling a multi-DSAL composition, and also when incrementally adding a new DSAL to an existing composition, many questions arise regarding the nature of the composition. Examples of questions that may be asked are:

- Which mechanisms already participate in the composition?

- What join points in the program may each mechanism affect?
- Are there join points that may be affected by multiple mechanisms, and if so, in what manner?

SPECTACKLE is a command line tool that helps to answers such questions. The input to SPECTACKLE is a set of manifest files characterizing the aspect mechanisms that participate in the composition. Each manifest is defined by the author of the corresponding DSAL, and lists properties of a particular aspect mechanism, e.g., the id of the mechanism, the kind of join points that it may affect, and the types of advice that the mechanism introduces (in the next subsection these properties are further explained). The intended user of SPECTACKLE is the composition designer.

3.1 Basic Exploration of the Composition

We first demonstrate a basic usage of the SPECTACKLE tool. In this example, the tool is used to explore the multi-DSAL composition and resolve an emerging feature interactions.

The composition designer starts by exploring the basic properties of the composition. Essentially, this means to identify the aspect mechanisms that take part in the composition, and to query for the basic properties of each participating mechanism. The command `mech` presents a list of all the mechanisms in the composition. For our example composition, `mech` produces:

```
spectackle> mech
validate
cool
aspectj
```

The composition designer continues exploring for the properties of each mechanism. The following transcript is a basic exploration of the COOL mechanism:

```
spectackle> adv cool
lock → before
unlock → after

spectackle> gran cool
method-invocation → execution(method)
```

The first command `'adv cool'` lists the advice types that COOL defines. In the same manner that ASPECTJ defines advice of type `before`, `after`, and `around`, any other aspect mechanism defines its own types of advice. Each line in the output refers to a single advice type, where the left-hand side (before the arrow) is the name of the advice type in COOL's terminology. In AWESOME, all advice types are normalized to a common base (ASPECTJ advice). The right-hand side of the output line indicates the normalization.

The second command `'gran cool'` shows the granularity [4] of the COOL mechanism. That is, the kind of join points in the base system that COOL may affect (advise). The left-hand side of the output line describes the join point kind in a platform independent fashion, whereas the right-hand side is the mapping to a common join point scheme defined by AWESOME. Overall, the output implies that COOL may affect the behavior of the program by inserting `lock` or `unlock` advice before or after method executions, respectively.

3.2 Exploring and Configuring Co-Advising

After gaining a general understanding about the mechanisms participating in the composition, the composition designer proceeds with investigating the possible interactions between them. As noted, each mechanism introduces one or more advice types. These advice types are expressed in the terminology of the corresponding DSAL yet they are mapped to a common base (ASPECTJ). In our example composition there are six advice types; three of ASPECTJ, two of COOL, and one of VALIDATE. The number of advice types may significantly increase as new DSALs are added to the composition. Naturally, advice belonging to different aspect mechanisms may operate at a shared join point. This kind of interaction is called co-advising [6].

SPECTACKLE provides means for exploring and configuring the co-advising in a composition. The composition designer may investigate the co-advising by issuing the command `'jp base -adv'`. The command outputs a list showing the advice types that may be applied at each kind of join point. The output is shown in Figure 1.

Each section in the output of the command (Figure 1) lists the advice types that may surround a join point of a particular kind. For instance, the first section shows the advice that may be applied at any join point of kind *call(method)*. The possible `before` advice appear in the line before the join point, `around` advice in the same line of the join point, and `after` advice are shown in the line after the join point. Since in our composition *call(method)* join points are not in the granularity of COOL nor of VALIDATE, only ASPECTJ advice may affect method calls.

The second section in the view is more informative. Here all possible advice types surround the *execution(method)* join point kind. This implies that all of them may be applied at join points of this kind. Note that the advice that operate `before` and `after` the join point appear in red sans serif font. It is an indication that *no order* is specified for these particular advice, which means that their execution order is arbitrary. This of course may be undesired; for instance, if an `aspectj.before` advice is executed before a `cool.lock` advice, then code executed within `aspectj.before` is not synchronized. If the code accesses shared application resources, it may lead to incorrect behavior.

Therefore, all sections with red advice (three in our case) indicate a possible conflict that should be resolved. The

```
spectackle> jp base -adv

aspectj.before
call(method)  aspectj.around
aspectj.after

aspectj.before  cool.lock  validate.validate
execution(method)  aspectj.around
aspectj.after  cool.unlock

aspectj.before
call(constructor)  aspectj.around
aspectj.after

aspectj.before  validate.validate
execution(constructor)  aspectj.around
aspectj.after

aspectj.before
initialization aspectj.around
aspectj.after

aspectj.before
preinitialization  aspectj.around
aspectj.after

aspectj.before
staticinitialization  aspectj.around
aspectj.after

aspectj.before
get(field)  aspectj.around
aspectj.after

aspectj.before  validate.validate
set(field)  aspectj.around
aspectj.after

aspectj.before
handler  aspectj.around
aspectj.after
```

Figure 1. Co-advising view for a composition of ASPECTJ, COOL, and VALIDATE.

SPECTACKLE command `'adv set'` supports resolution of advice ordering conflicts. For instance, in the set of commands in Figure 2, the composition designer sets an ordering for `before` and `after` advice, and then verifies the result.

The `jp` command with the `-kind` flag presents the same co-advising view but for a specific kind of join point. We can see that now the `before` and `after` advice are colored in green (`typewriter` font), which means that an advice order

```
spectackle> adv set -before validate.validate
cool.lock aspectj.before
advice order was set

spectackle> adv set -after aspectj.after cool.unlock
advice order was set

spectackle> jp -kind execution(method) -adv
validate.validate cool.lock aspectj.before
execution(method)  aspectj.around
aspectj.after  cool.unlock
```

Figure 2. Setting an advice order

is defined, and that they appear according to their order of execution.

4. From Specification to Implementation

SPECTACKLE supplies AWESOME with the composition specification and with the manifest files of the aspect mechanisms that participate in the composition. This enables AWESOME to generate parts of the multi-DSAL weaver code that would otherwise needed to be coded manually by the composition implementer. In this section, we explain how a specification-based composition approach reduces manual coding efforts.

4.1 Configuring Advice Order

In Section 2, the composition designer specified in SPECTACKLE an order between pieces of `before` advice: a `validate` advice is executed first, followed by any COOL `lock` advice, and eventually any ASPECTJ `before` advice. It was also specified that ASPECTJ's `after` advice should precede COOL's `unlock` advice. Here are the corresponding entries that SPECTACKLE creates in the composition specification file:

before-advice-order: validate.validate, cool.lock, aspectj.before
after-advice-order: aspectj.after, cool.unlock

Provided with this kind of specification, AWESOME is now able to automatically generate an aspect that is woven into the multi-DSAL weaver code and configures the specified advice order. Recall that previously the composition implementer needed to implement the configuration aspect *by hand*.

In Figure 3, a simplified version of the configuration aspect is presented. The aspect advises the `multiOrder` method, which is a part of the AWESOME weaving process. The method is provided with `multiEffects`, a list of all advice (effects) that are going to be woven at a specific

```
public aspect AdviceOrderConfig {
  List around(MultiMechanism mm, List multiEffects, BcelShadow shadow):
    execution(List MultiMechanism.multiOrder(List, BcelShadow))
      && this(mm) && args(multiEffects, shadow) {

      int coolPos = mm.getMechanismPos(COOLWeaver.class);
      int ajPos = mm.getMechanismPos(AJWeaver.class);
      int validatePos = mm.getMechanismPos(ValidateWeaver.class);
      List<IEffect> result = new ArrayList<IEffect>();

      // multiEffects is a List of List<IEffect>
      List<IEffect> coolEffects = (List<IEffect>)multiEffects.get(coolPos);
      List<IEffect> ajEffects = (List<IEffect>)multiEffects.get(ajPos);
      List<IEffect> validateEffects = (List<IEffect>)multiEffects.get(validatePos);

      // setting the desired advice order
      result.addAll(coolEffects);
      result.addAll(ajEffects);
      result.addAll(validateEffects);

      return result;
}
```

Figure 3. An ASPECTJ aspect configuring an advice order

join point shadow. Each element in the list is in itself a list, holding the effects of a particular mechanism. The method extracts all the effects from the inner lists and returns a single flattened list of effects. The aspect makes sure that the advice are flattened according to their specified execution order.

The code is far from being complicated but it still requires a considerable coding, basic understanding of the AWESOME weaving process, and of low-level APIs (e.g., the `BcelShadow` class). Hence an automatic generation of the aspect saves time and effort. Moreover, the specification is much more explicit and precise, and thus promotes reasoning and communication.

5. Conclusion

An improved process for composing multiple DSALs was described and illustrated in the context of the AWESOME composition framework. The process begins with an analysis of the composition using a novel tool called SPECTACKLE. The analysis helps the composition designer to formulate the composition specification. AWESOME is provided with the specification to facilitate automatic code generation.

DSAL composition frameworks make the development of applications using multiple DSALs possible. Yet there are still shortcomings hindering the adoption of the approach. We believe that a specification-based composition process of the type described here, with the appropriate tool support, has the potential to make the multi-DSAL development more practical and more accessible.

References

[1] Y. Apter, D. H. Lorenz, and O. Mishali. A debug interface for debugging multiple domain specific aspect languages. In *AOSD'12*, Potsdam, Germany, March 25-30 2012. ACM. To appear.

[2] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersen, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings European Conference on Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, Berlin, Heidelberg, and New York, 2001. Springer-Verlag.

[3] S. Kojarski and D. H. Lorenz. Awesome: An aspect co-weaving system for composing multiple aspect-oriented extensions. In *Proceedings of the 22nd Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'07)*, pages 515–534, Montreal, Canada, October 21–25 2007. ACM Press.

[4] S. Kojarski and D. H. Lorenz. Identifying feature interaction in aspect-oriented frameworks. In *Proceedings of the 29th International Conference on Software Engineering (ICSE'07)*, pages 147–157, Minneapolis, MN, May 20-26 2007. IEEE Computer Society.

[5] C. V. Lopes and G. Kiczales. D: A language framework for distributed programming. Technical Report SPL-97-010, Palo Alto Research Center, 1997.

[6] D. H. Lorenz and S. Kojarski. Understanding aspect interactions, co-advising and foreign advising. In *Proceedings of ECOOP'07 Second International Workshop on Aspects, Dependencies and Interactions*, pages 23–28, Berlin, Germany, July 30 2007.