Modularizing Invasive Aspect Languages

Thomas Cleenewerck Theo D'Hondt
Vrije Universiteit Brussel
{tcleenew,tjdhondt}@vub.ac.be

Abstract

In domain-specific aspect languages we observe that aspects are translated to base code and subsequently require a complex integration into base code while guaranteeing the correctness of the aspect and the base code in the woven code. We call this phenomenon invasively composed aspects. Weavers for invasive aspect languages operate on the base language level and offer dedicated support for crosscutting code. Unfortunately, current implementations poorly modularize the implementation of invasive aspect languages. This hampers their (unanticipated) evolution and severely reduces the reusability of their constructs. We suggest an approach where the specification of the crosscutting behavior is expressed on a higher semantic level. To this end, we raise the abstraction level of base languages towards the specific domain of the aspect languages. As such, we enable a modular, declarative approach. We illustrate our approach with KALA, a domain-specific aspect language.

Categories and Subject Descriptors D3.4 [*Programming Languages*]: Processors—Translator writing systems and compiler generators

General Terms Languages, Design

Keywords Modularity, Generative Programming, Aspect-oriented Programming, Language Engineering, Domain-specific Languages, KALA, Linglets

1. Introduction

By shifting from general-purpose aspect languages towards more domain-specific aspect languages, the abstraction level of aspect languages is raised. In such languages, aspects are no longer solely described in terms of general-purpose language constructs like method invocations in object-oriented languages. Instead, aspect languages are impregnated with specific language constructs which are tailored towards the concepts of a particular problem domain. Aspects can then be described in terms of the concepts of the problem domain. As these concepts are not natively supported by the base language, the implementation of a domain-specific aspect language first expresses the concepts of the aspect domain into more general-purpose code fragments of the base language, and as we will explain later in more detail subsequently invasively integrates them into the base code. We refer to this kind of aspects as *invasively composed aspects*. Such aspects have been encountered in the implementation of KALA, a domain-specific aspect language for advanced transaction management (Johan Fabry and D'Hondt 2008) and in the implementation of HLBR, a domain-specific aspect language for connecting business rules (Cibran 2007).

In general, *invasive composition* is a difficult and delicate undertaking as it must know how and where to modify a program without breaking its functionality (Fabry et al. 2007; Brichau 2005). This means that, in the case of the invasive integration of aspect and base code, the correctness of both the aspect and the base code must be ensured in the woven code. Therefore the implementation of the aspect language constructs require detailed knowledge about the base code in order to modify the base correctly. The required knowledge even increases when multiple invasive aspect language constructs operate on the same base code, because constructs must know about all the other constructs which potentially might have already changed the base code. This shared knowledge not only contaminates the implementation of the aspect language constructs, but erects many dependencies that result in a tightly coupled aspect language implementation. It has been argued that this leads to a maintenance nightmare (Voelter and Groher 2007) and to change propagation (Brichau 2005) throughout the language implementation.

We find that the lack of modularization in the bulk of the aspect-specific and general-purpose language development techniques (Cleenewerck 2007a) prohibits unanticipated evolution and reusability of aspect language constructs. This is unfortunate for several reasons, as these software qualities are especially important in the context of domain-specific aspect languages. First, domain-specific languages need to evolve along with their domain (Cleenewerck 2007b,a; lmar Juergens and Pizka 2006; Fabry et al. 2007; Bosch and Dittrich). Second, software consists of several kinds of crosscutting concerns. Hence, it does not only suffices to address each concern with a separate domain-specific aspect language, but several languages need to be used in conjunction with one another. We believe that modularized aspect language constructs are a fundamental step to be able to control and define the interactions among constructs which initially belonged to different languages.

The challenge of invasively composed aspects lies thus in their invasive composition into the base code. Invasive composition also manifests itself in general-purpose aspect languages and has been recognized as a general problem in software composition (Assmann 2003). This paper focusses on the specific context of invasive composition in domain-specific aspect languages as we want to exploit the high abstraction level of domain-specific aspect languages to our advantage.

It is our position that the gap between the abstraction level of domain-specific aspect languages and bases language has been wrongly addressed. In previous work (Cleenewerck 2005, 2007a), we developed an approach to disentangle and decouple semantics of language constructs that require invasive composition. Our experiments show that, in *highly structured languages* (Cleenewerck 2005) a lot of the integration can be automated and facilitated by

using a more declarative formalism. In the case of invasively composed aspects, integration is performed on base languages which more general-purpose than the domain-specific aspect language. As the base languages are less structured, the approach is far less effective. In order to attain a declarative composition technique for invasively composed aspects, propose to increase the structure of the base language by raising the abstraction level of the base language towards the specific domain of a given aspect language. The declarative composition technique preserves the modularization of the aspect language constructs. Moreover, it will allow a robust and concise way of ensuring the correctness of aspect code and the base code in the woven code. In this paper, we present a case study in modularizing such a domain-specific invasive aspect language by implementing a subset of KALA.

2. Motivating Example

In this section we illustrate the integration complexities of invasively composed aspects found in domain-specific aspect languages. The correct integration of the translational semantics of invasively composed aspects depends on various constraints and dependencies such as their ordering and the transactional context in which they are used.

2.1 KALA

The running example which we use in this paper is the implementation of KALA (Johan Fabry and D'Hondt 2008), a domain-specific aspect language (DSAL) for advanced transaction management. KALA has been designed to modularize the aspect of advanced transaction management. Tangled aspect code occurs when an aspect itself consists of multiple crosscutting sub-concerns (Johan Fabry and D'Hondt 2008). It is a prime example of invasively composed aspects as the weaving of tangled aspects requires a complex integration of its different subconcerns.

With KALA, Java methods can be declared transactional via a declarative specification. The KALA code excerpt in Figure 1 declares the method `transfer` of the class `Cashier` as a transaction.

```
1  Cashier.moneyTransfer(Account, Account, int) {
2    alias(parent Thread.currentThread())
3    name(self Thread.currentThread())
4    begin  {  dep(self wd parent) }
5    commit {  del(parent self)
6              view(self parent)  }
7    abort  {  terminate(self) } }
```

Figure 1. Transactional Declaration of a Java Method in KALA.

Transactional properties are grouped in four different *transactional contexts* (see Figure 1): `initialize` (lines 2-3), `begin` (line 4), `commit` (line 5-6) and `abort` (line 7). Each of them correspond to the lifecycle of a transaction.

KALA provides a set of language constructs for advanced transaction management. With each construct a particular property of a transactional method can be stated. Some of these constructs are illustrated in Figure 1. There are five kinds of transactional properties. *Naming and grouping* constructs: `name` and `alias` respectively identify and publish transactions in the run-time transaction monitor, using keys which are computed by Java expressions (lines 2-3). *Dependencies* establish dependencies among the transaction in order to control how different translations should be executed (line 4). *Delegations* delegate the responsibility for committing the intermediate changed data (line 5). *Views* declare what intermediate data changes can be viewed by a transaction (line 6). Finally, *terminates* remove a transaction from the run-time transaction monitor (line 7).

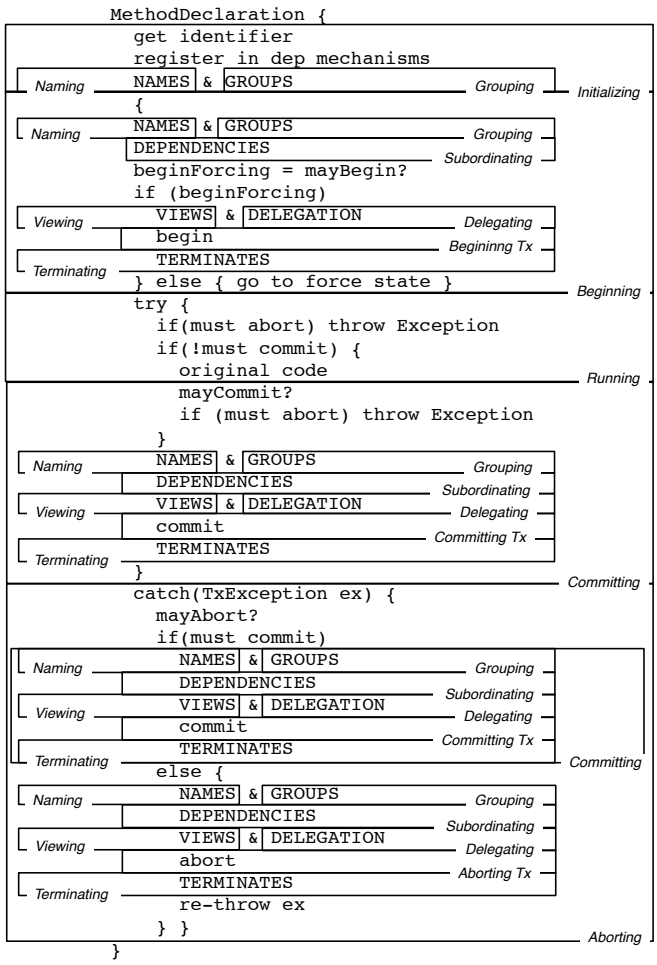

Figure 2. Simplified outline of a transactional method marked with the intentions of the statements.

Now that the different kinds of language constructs of KALA are explained, the next section discusses the invasive character of these language constructs.

2.2 Invasively composed aspects in KALA

In KALA, each construct denotes a transactional property and has a particular kind of semantics that configures the run-time transaction monitor. More importantly, the semantics of each construct plays a well defined role in the overall semantics of a transactional declaration. Figure 2 shows the pseudo code which is produced when translating a KALA transactional declaration to Java code. Such code is known as demarcation code. The expressions in capital indicate where the code produced by the different language constructs have to be put. The boxes denote the different intentions within the demarcation of a transactional method and reflect the expertise of the domain-expert.

KALA transaction declarations nicely illustrate the notion of an invasively composed aspect: the tangled web of boxes reveals that a complex and well defined ordering of transactional properties is required. For example, naming and grouping must always be executed first because the other constructs depend on them. Terminate statements are always executed after the actual transactional lifecycle change (i.e. begin, abort, commit) is executed.

The integration of the translational semantics of each construct even depends on the transactional contexts and is thus not solely determined by the constructs themselves. In Figure 2 the various kinds of constructs to express transactional properties are boxed in their transactional context. The code shows that the distribution of the constructs differs for certain transactional contexts. For example, dependencies and naming & grouping are not put together with the views & delegation in case their transactional context is begin. Table 1 lists all the constructs with their transactional context dependent integration semantics.

	integration semantics	
constructs	order	transactional context
naming & groups	before initialize	begin
naming & groups	first	-
dependencies	after naming	-
views & delegations	after naming	abort or commit
views & delegations	before begin	begin
terminates	after begin	begin
terminates	after abort	abort
terminates	after commit	commit

Table 1. Integration semantics of the different constructs.

3. Crosscutting code generators

KALA is a small language to implement. The language can therefore be relatively easily implemented as one monolithic entity. The integration semantics of the individual language constructs is then absorbed into a single entity that composes the different language constructs. However, a monolithic design is not the most appropriate fit for our needs: the whole language implementation must be changed to incorporate unanticipated changes in terms of language constructs such as adding new constructs to KALA, changing the semantics of language constructs and composing it with other aspect languages. To avoid changing the whole implementation, the language should be implemented such that the translation and integration of each aspect language construct is modularized. In the next section, we will explain in detail to what extent we can modularize language constructs in the implementation of DSALs.

3.1 Code Templates

Code templates are a general notion that is used by a large and heterogeneous group of language development techniques (LDTs) to express the semantics language constructs. In its most basic and common form a code template is a code fragment that is parameterized with other code fragments. These parameters are bound to other code templates by querying for the desired information. The differences among the various implementation techniques including macro's, template engines (XSLT (Laird)), attribute grammars (JastAdd (Hedin and Magnusson 2003)), rewrite rule systems (ASF+SDF (van Deursen et al. 1996) and Stratego (Visser 2001)), plain interpreters and reflection models (Reflex (Tanter 2006)) lie in how code templates are constructed, how they are parameterized and how they are composed.

As the Linglet Transformation System (LTS) (Cleenewerck 2007a) provides unique mechanisms that allow us to modularize code templates, we skip a general introduction and immediately proceed by explaining how code templates can modularize the semantics of language constructs in LTS.

3.2 Modularized Code Templates

LTS is an prototype-based object-oriented development technique for the implementation of languages. The system is divided into two layers: a kernel for defining language constructs and a reflective layer for establishing the interactions among the language constructs.

```
linglet Alias {
  syntax { "alias" "(" name lookupkey ")" }
  generate {
    #Statements{ txmonitor.register('name,'lookupkey) }
  } }
```

Figure 3. Parameterized templates in LTS.

A linglet defines the syntax and the semantics of a single language construct in isolation from any other language construct. In its most basic form, as any other LDT, the translational semantics of language constructs can be expressed in the form of a *parameterized code template*. The code template of a language construct is parameterized with meta-variables, which refer to the semantics of the parts of the language construct. In Figure 3 the semantics of the `Alias` language construct is implemented in the `generate` method using a code template. The `#`-construct of LTS creates code templates of the code between curly brackets that follow the operator. The identifier between the `#` and the code denotes the language construct which should be used to parse the given code fragment. The quoted variables `name` and `lookupkey` refer to the semantics of the name and the key part of an alias.

LTS distinguishes between two kinds of results: *local* and *nonlocal* results. The local results are typically used in parameters of code templates. They have simple integration semantics: simply inject the code bound to the parameter into the location specified by the parameter. In the code example of Figure 3, the semantics of `name` and `key` are directly composed in the set of statements that register the transaction.

However, it is not trivial to integrate the semantics of the transactional properties in the demarcation code. With mere parameterization, the composition of code is limited to predefined holes provided by each template. The problem is that templates cannot be designed up-front, anticipating every potential involvement of other templates as that breaks their modularity. In LTS we refrain from doing so by allowing code templates to be returned as nonlocal results. Each nonlocal result can define in which other code template it must be integrated and how. This renders templates modular as each construct encodes its translational semantics as well as its integration semantics and other templates do not have to specify how to handle this.

The code in Figure 4 shows how a set of `view` statements can be integrated into the demarcation code of Figure 2. The set of statements called `views` specializes two methods `corresponds:` and `combine:` of some strategy. The `corresponds:` method (lines 2-3) determines when the statements can be combined with another set of statements. The actual combination is stated in the `combine:` method (lines 4-8).

```
1  views
2    corresponds: master {
3       master linglet hasType: ast linglet type.  }
4    combine: master { | stat |
5      stat := master find:
6          #Statements{ beginForcing = mayBegin?
7                       if(beginForcing) {  } }.
8      stat consequent addFirst: ast.  }
```

Figure 4. Example of the INR strategy

3.3 Modularized Integration Strategies

How the integration semantics is specified and how the integration is performed depends on the kind of *interaction strategy* that

is used. Interaction strategies are reflective modular extensions of the LTS system. This enables them to change the behavior of linglets, so that linglets can interact with another while the linglets themselves remain modular. For the purpose of this paper, a strategy is required such that nonlocal results are properly integrated in the code produced by the different aspect language constructs without requiring that constructs are polluted with integration semantics specific for nonlocals produced by other constructs.

The INR strategy is one of the strategies which are typically used in LTS to integrate nonlocal results. The strategy is depicted as a cloud in Figure 5. It intercepts whenever nonlocal results are returned[1]. Upon interception, it tries to integrate the nonlocals. As is shown in Figure 5, the INR strategy intercepts the nonlocal `Statements` and tries to integrate them in the Java method containing the outline of the demarcation code for a method.

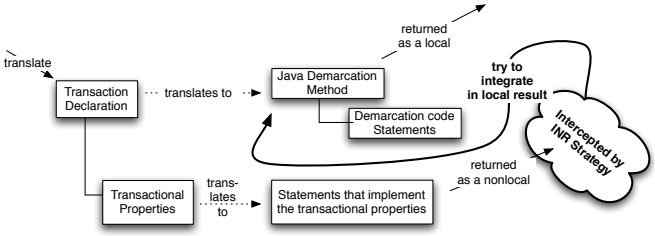

Figure 5. Translation in LTS.

3.4 Non-modularized Integration Semantics

DSALs translate may translate to a wide range of target languages ranging from general-purpose languages to domain-specific languages. The challenge of invasive composition we tackle in this paper is due to the use of what we call *generic languages* as a target language. Generic languages have very few distinguishable language constructs which can be almost arbitrarily combined. General-purpose languages are a prime example of generic languages, but also some domain-specific languages are generic languages e.g. HTML, FPIC (Kamin and Hyatt 1997).

The declarativeness of the INR strategy depends on the *genericity* of the language. In generic languages, the many language constructs that can be arbitrarily combined with one another lead to an explosion of the amount of possible locations and possible integrations. Hence, we are forced to rely on detailed layout of other code templates to determine the exact locations and specify the integration semantics. This renders the integration semantics non-declarative, fragile and potentially hazardous as it can easly break the code of other templates. This is for example the case in the integration semantics of `views` shown in Figure 4. The `combine:` method (lines 4-8) contains detailed knowledge and assumptions of the demarcation code to figure out where the views should be integrated. Deriving this from the base code means that we depend a specific implementation details i.e. a couple of statements that execute `mayBegin?`, assign it to the variable `beginForcing` which is subsequently tested in a subsequent if-statement.

By increasing the structure of the base language via raising its abstraction level, the integration logic can be written in terms of the new abstractions. As such the logic could be more declarative and contain fewer details. Moreover INR can then better disambiguate the possible locations and integrations based on the grammar, resulting in a semi-automatic integration. If for example, the `beginning` intension would be explicitly present in the demarcation code, the integration of views would only have to refer to this intension. Therefore its coupling with the demarcation could have been significantly reduced.

[1] Technically, it intercepts each AST node composition and tries searches for any remaining nonlocals

4. Constructing Intermediate Languages from Base Languages using Connotations

We raise the abstraction level of the base languages by creating a new intermediate language. In this language, new constructs are provided that explicitly state the intention of expressions in terms of generic language constructs. The intentions are not merely a loose collection. They are structured by the language specification of the intermediate language. As such we obtain a conceptual framework of intentions, stipulating when and how intentions can be used. In the next section we will show how the use of intentions to express code templates renders templates semantically richer such that we can effectively modularize the semantics of aspect language constructs.

The intermediate languages are enriched generic base languages with abstractions taken from the domain of the aspect languages. The constructs with which we enrich Java to better express the invasively composed aspect of transaction management denote the different intentions in the template of a transactional method (see the boxes in Figure 2). Some constructs denote the intention of transactional properties such as `naming`, `grouping`, `viewing`, `delegating`, `subordinating`, `terminating`. Other constructs refer to the lifecycle of transactions like `initializing`, `beginning`, `aborting`, `committing`. Yet other constructs denote the transitions of the lifecycle executed by the transaction monitor such as `beginningtx`, `abortingtx`, `committingtx`.

When the code templates of KALA describe the intentions of the produced Java code, integration semantics can be declaratively specified on a rich semantic level. There is no need to explicitly detail where code can be integrated and prevent incorrect changes to produced code. For example, with a *comes before* relationship (say `CB`) one can ensure that views of a commit are executed before the transaction commits by stating `CB(viewing, committingtx)`.

The development of an intermediate language complicates the implementations conceptually and imposes new maintenance and evolution problems. Moreover, when defining a plain entirely new language we would only shift the invasive composition problem to the implementation of the higher-level intermediate language in terms of the original base language. The challenge is thus to design an intermediate language such that the effort of creating an intermediate language is minimized and avoid a translation from intermediate language to the original base language. To this end, we chose to embed the intermediate language in the generic language by extending the base language with additional constructs.

New language constructs are created by connotating the constructs of the generic language. A *connotation* of a base language construct captures a specific intention that can be expressed by using that base language construct. New language constructs are thus syntactically indistinguishable from their generic counterpart.

Connotations get transparently integrated in the base language. They introduce an alternative for a construct in the base language. Hence, the new construct can be used wherever the base construct can be used. For example, `beginningtx` is a subtype of `Statements`, and as such whenever a set of statements is expected a `beginningtx` can be used.

Connotations specialize the behavior of base language constructs with additional semantics:

1. Specification of their integration semantics. By attaching the integration semantics to the connotations themselves, their integration semantics no longer has to be specified in other code

templates. As such, the semantics of language constructs can be modularized.

2. Specification of their usage context. The usage context may span over several productions in the base language grammar, the language extension mechanism allows language developers to abstract away from those parts of the grammar which do not influence this usage context.

5. Language connotations in LTS

Language connotations are natively supported in LTS. This allows us to design the necessary strategies to declaratively and modularly realize the integration of the modularized translational semantics of invasive aspect language constructs. The creation of intermediate languages in LTS involves two steps: the creation of new linglets to define new constructs and their integration in a language. These two steps are explained in the two following subsections.

5.1 Defining New Constructs

New language constructs of intermediate languages are created just like any other language construct. Languages in LTS are defined in a language specification (LS) by composing a set of linglets. The linglets in a LS are in essence defined by specialization: an existing linglet definition is specialized by binding their syntactic parameters to other linglets and by adding or overriding methods. Specialized linglets can be aliased with a new type.

Connotations are defined by aliasing specializations of linglets. In Figure 6 the `beginningtx` connotation of Java statements for KALA is shown. The connotation is defined as an alias of the linglet `Statements`.

```
beginningtx = Statements.
```

Figure 6. The definition of the `beginning` connotation.

5.2 Defining the Intermediate Languages

Aliases of linglet specializations denote a new type which is a subtype of the specialized linglet. Because a type is substitutable by a subtype, the new constructs will automatically be allowed to be used wherever the generic language construct is allowed, hence they seamlessly blend in the language.

Although substitutability is already integrating the language features in the language, as we explained in Section 4, we furthermore need to override or further constrain the usage context of a connotation. As this usage context spans over many productions, and as context-free grammars are not designed to handle context, the usage context of connotations in LTS is specified by specializing the INR strategy.

Recall that INR is a strategy for composing non-local code fragments. It provides three methods `corresponds`, `combine` and `integrate`. The location(s) in the code where the non-local fragments need to be integrated are determined by the `integrate` method. The strategy checks whether the non-local fragment corresponds with an already existing part of the code using the `corresponds` method. If so, the two are merged via the `combine` method. When a connotation is defined by specializing a new linglet, this behavior can thus also be specialized. As such, we can easily specify where a connotation is allowed to be used although it spans over many productions.

In Figure 7 the connotation `beginningtx` of a Java statement for KALA is shown. We specify how this language construct extends the Java language by specializing the `Statements` linglet. In this case, we override the method `corresponds:`. The method `corresponds:` is a method from the INR strategy that determines whether a given AST node can be combined with an existing AST of the produced program (called `master`). The specialization stipulates that a `beginningtx` can only be a part of group of statements that are connotated as `beginning`.

```
beginningtx = Statements
   corresponds: master {
      (previous corresponds: master) and: [
         (master hasType: 'beginning')
         or: [ master ancestor: 'beginning' ]  ] }.
```

Figure 7. Extending the Java language with connotations for KALA in LTS.

Remark that we did not hardcode the exact integration location of `beginningtx` but only added an additional constraint. The actual location of integration is first selected by the `integrate:` method of the INR strategy which relies on the grammar and the type of the `beginningtx` to determine whether a given integration position is a correct one. This reuse clearly shows that changes in the generic language, in this case Java, will not likely cause a cascade of changes in intermediate languages.

5.3 Integration Strategy

LTS does not offer a built-in strategy to specify and perform the integration of non-local fragments. Language developers can customize LTS with their own new strategies or customizations of existing strategies. The strategy which we use in our examples is constructed on top the INR strategy.

The integration strategy that will be used to implement KALA primarily operates on Java statements. An excerpt of that strategy definition is shown in Figure 8. The strategy consists of two parts: a public part and a private part.

- The public part will mainly be used to define the translational semantics of the invasive aspect language constructs. It consists of three methods `comesFirst:`, `comesBefore:` and `comesAfter:`. They allow developers to state the order of the statements when integrated: which statement that comes first, and which statements that execute before or after another statement. Each of them are given a default implementation. Connotations need to specialize the appropriate method(s) for specifying their specific integration semantics.
- The private part is mainly used to effect the semantics defined by the public part. The method `insertFirst:` inserts a statement before the rest, the methods `insertBefore:` and `insertAfter:` insert a statement before or after another one. These methods are used to respond to certain method calls triggered by the INR strategy: `corresponds:` and `combine:`. These methods are called by the strategy when it detects that groups of statements correspond and decides to combine the two groups. The methods `corresponds:` and `combine:` are specialized for the linglet `Statements`. The `corresponds:` method states that two statements correspond only if the existing statement (called the `master`) has the same type of the non-local (called the `ast`). The `combine:` method integrates a group of statements in an already existing group of statements. For each statement we check that it needs to come first by using the `comesFirst:` method. If that is is the case, the actual integration is performed by the `insertFirst:` method.

A simple example connotation called `naming`, shown in Figure 9, is using the above integration strategy. This connotation specializes statements by overriding the method `comesFirst:`. It

```
Statement
   comesFirst: s { false }
   comesBefore: s { false }
   comesAfter: s { false }
   insertFirst: statements {
      statements body add: master on: 1. }
   insertAfter { ... }
   insertBefore { ... }.

Statements
   corresponds: master {
      master linglet type == ast linglet type. }
   combine: master {
      ast body do: [ :statement | | comesfirst |
         comesfirst := master body forAll: [
            :masterstat |
                statement comesFirst: masterstat   ].
         comesfirst ifTrue: [ ast insertFirst: master ]
                    ifFalse: [ master body add: master ]
      ] }.
```

Figure 8. Integration strategy for Java Statements.

states that a naming connotation should always be integrated before any other set of statements[2].

```
naming = Statements
            comesFirst: s { true }
```

Figure 9. Defining integration semantics in the language specification of Java for KALA in LTS.

6. Implementing Invasive Aspect Language Constructs using Connotations

In this section we will show that by using connotations the translational semantics including the integration semantics of invasive aspect languages can be defined modularly and declaratively. The translation of each invasive aspect language construct is defined by a code template. The code templates are constructed with connotations. This allows the semantics of other language constructs to express their integration semantics in terms of the connotations used in other language constructs and not in terms of other language constructs themselves. So with connotations, invasive aspect language constructs do not need to explicitly refer to other aspect language constructs, do not depend on their translational semantics and do not require extra semantics to anticipate other aspect language constructs.

6.1 Code templates

Figure 10 shows how the code template of a transactional method can be expressed in LTS using connotations. As connotations are also language constructs, the `#` can be used to create connotated code fragments as well.

The outline of a transactional method (`txmethod`) is created by `#MethodDeclaration{...}` (lines 18-27). It is composed out of other templates using quoted variables: `inittx`, `beginning`, `running` and `abort`. The outline is the local result of the `Transaction` linglet (line 30). The translational semantics of the transactional properties (`txproperties`) (lines 28) are returned as nonlocal results (line 29). They are intercepted by the INR strategy and composed in the correct way with the outline of a transactional method.

Quotation within the `#` in LTS is polymorphic. This means that either the exact type or any subtype of the value of a quote variable

[2] Note that this will be further constrained by the `begin`, `commit` and `abort` constructs, depending on the transactional context in which naming is used.

```
1  Linglet Transaction {
2     syntax { name "{" ( body )* "}" }
3     generator { | name initbegin inittx
4        begin begintx abort aborttx run
5        commit committx copycommit txmethod |
6        name := ast name generate.
7        inittx := #initializing{ ... }.
8        begintx := #begintx{ begin }.
9        begin := #beginning{ mayBegin?
10                            if (no forcing) { }
11                               'begintx
12                            } else { go to force state }
13       } insertAfter: anAST  {
14                begintx insertAfter: anAST }.
15       commit := #committing{ ... }.
16       run := #running{ ... }.
17       abort := #abortting{ ... }.
18       txmethod := #MethodDeclaration{ 'name {
19        'inittx
20        {
21          'begin
22          try {
23             'run  }
24          catch(TxException ex) {
25             'abort }
26        }
27       }
28       txproperties := ast body generate.
29       txmethod nonlocals addAll: txproperties.
30       txmethod   }  }
```

Figure 10. Connotating code fragments.

will be accepted in the parsed code fragment. As such connotations of statements can be used within a code fragment even where the language specification only specifies statements.

6.2 Intention misalignment

In Section 5.1 we have defined connotations in LTS as specialized generic language constructs. The connotations used in our example denote the intention of statements. However, not all intentions align with distinct statements. Consider for example the `beginning` code template in Figure 10. In Figure 2, it is shown that `terminate` statements that are to be executed after a begin should be integrated right after the `beginningtx` and not after the `if-else` statements. To specify this, the `begin` connotation overrides the default behavior of the `insertAfter:` method: it redirects to the `begintx` connotation.

7. Related Work

Crosscutting code generators for invasively composed aspects are encountered in many software engineering disciplines where some form of meta-programming is used. The most common established disciplines involving meta-programming are generative programming, transformation systems and aspect technology.

In generative programming as well as in transformational approaches, templates are the primary means to define the semantics of invasive aspect language constructs. In our approach, connotations raise the abstraction level of the base language and linglets modularize code templates because integration semantics can be directly attached to the code templates themselves.

Also most aspect technologies compose code fragments at the level of generic base languages. In languages such as AspectJ (Colyer 2005), the pointcuts are fragile (Gybels and Brichau 2003) as they are expressed in terms of syntactical patterns of the program. More expressive pointcut languages only delve deeper in the implementation rendering them as dependent on the actual implementation. The deduction of the role or the intention of a part of an implementation in generic languages rapidly becomes very complicated (Tourwé et al. 2004) or even impossible in general.

Extensible aspect language frameworks like Josh (Chiba and Nakagawa 2004), AspectBench (Avgustinov et al. 2005) and Reflex (Tanter 2006) range from AspectJ-like extensions over full compilers to powerful AOP kernels. Also in these language frameworks, composition and integration is performed at the level of generic base languages. Despite the dedicated nature of these frameworks, the semantics of aspect language constructs are not modularized: their semantics is implemented using plain parametrized templates (see Section 3.1).

Pointcuts in approaches like Compose* (Bergmans and Aksits 2001) and AspectWerkz (Boner 2004) can refer to annotations. Annotations enrich statements with metadata and are put in the base code. However, a general annotation mechanism allows arbitrary kinds of annotations, in other words they lack a conceptual framework to define, annotate the code and to use these annotations. In contrast connotations form an intermediate language. They also differ with annotations as the later is consists of only data while the former are accompanied with integration semantics.

Another part of the definition of crosscutting code is how the crosscutting code affects the program. Most aspect related technologies offer a limited set of integration facilities i.e. before, after and around integrations of method invocations. Whereas in our approach more complex integration semantics can be defined e.g. integrations delve deep into the code, compute integration locations and satisfy non-trivial ordering.

8. Conclusion

The semantics of invasive aspect language constructs requires to invasively change the semantics of other constructs. Current approaches fail to modularize their semantics because they specify the integration semantics in terms of a generic base language, and do not offer the appropriate techniques to attach integration semantics to code fragments. In our approach we exploit the richer domain-specific nature of invasively composed aspects, by enriching the generic base language with new constructs of the domain of the aspects. These new constructs connotate constructs of the generic base language and reflect the intention of produced code fragments. The rich constructs increase the abstraction level of the generic language. As a result, the integration semantics can be defined modular and declarative, abstracting away from delicate and highly specific code manipulations.

The implementation is performed in the Linglet Transformation Systems (LTS). We showed that the language connotations naturally blend in the general purpose language. Moreover, the ability to define custom(ized) strategies allows us to (1) modularize the specification of the integration semantics of invasive language constructs, and (2) results in a declarative integration which can be tailored towards particular invasive language constructs.

References

U. Assmann. *Invasive Software Composition*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003. ISBN 3540443851.

Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible aspectj compiler. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 87–98, New York, NY, USA, 2005. ACM. ISBN 1-59593-042-6. doi: http://doi.acm.org/10.1145/1052898.1052906.

Lodewijk Bergmans and Mehmet Aksits. Composing crosscutting concerns using composition filters. *Commun. ACM*, 44(10):51–57, 2001. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/383845.383857.

Jonas Boner. AspectWerkz AOP BOF, AOP in J2EE, technical session, javaone., 2004.

Jan Bosch and Yvonne Dittrich. Domain-Specific Languages for a Changing World. URL `citeseer.ist.psu.edu/82695.html`.

Johan Brichau. *Integrative Composition of Program Generators*. PhD thesis, Vrije Universiteit Brussel, 2005.

Shigeru Chiba and Kiyoshi Nakagawa. Josh: an open aspectj-like language. In Gail C. Murphy and Karl J. Lieberherr, editors, *AOSD*, pages 102–111. ACM, 2004. ISBN 1-58113-842-3.

Maria Cibran. *Connecting High-Level Business Rules with Object-oriented applications: an approach using Aspect-Oriented Programming and Model-Drive Engineering*. PhD thesis, Vrije Universiteit Brussel, 2007.

Thomas Cleenewerck. *Modularizing Language Constructs: A Reflective Approach*. PhD thesis, Vrije Universiteit Brussel, 2007a.

Thomas Cleenewerck. Disentangling the Implementation of Local-to-Global Transformations in a Rewrite Rule Transformation System. In *Proceedings of the Symposium on Applied Computing Conference*, 2005.

Thomas Cleenewerck. Implementing languages with plans for growth, the 6th belgian-netherlands software evolution workshop, university of namur, belgium, 2007, 2007b.

Adrian Colyer. Aspectj. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 123–143. Addison-Wesley, Boston, 2005. ISBN 0-321-21976-7.

Johan Fabry, Éric Tanter, and Theo D'Hondt. Relax: implementing kala over the reflex aop kernel. In *DSAL '07: Proceedings of the 2nd workshop on Domain specific aspect languages*, page 3, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-659-8. doi: http://doi.acm.org/10.1145/1255400.1255403.

Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 60–69, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-660-9. doi: http://doi.acm.org/10.1145/643603.643610.

Görel Hedin and Eva Magnusson. JastAdd: An Aspect-oriented Compiler Construction System. *Sci. Comput. Program.*, 47(1):37–58, 2003. ISSN 0167-6423. doi: http://dx.doi.org/10.1016/S0167-6423(02)00109-0.

Eric Tanter Johan Fabry and Theo D'Hondt. Kala: Kernel aspect language for advanced transactions. In *Science of Computer Programming*. Science Direct by Elsevier, 2008.

Samuel N. Kamin and David Hyatt. A special-purpose language for picture-drawing. In *DSL'97: Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*, pages 23–23, Berkeley, CA, USA, 1997. USENIX Association.

Cameron Laird. XSLT Powers a New Wave of Web Applications. http://www.linuxjournal.com/article/5622, 2002.

lmar Juergens and Markus Pizka. The Language Evolver Lever – Tool Demonstration. In *Electronic Notes in Theoretical Computer ScienceVolume 164, Issue 2, Proceedings of the Sixth Workshop on Language Descriptions, Tools, and Applications (LDTA 2006)*, pages 55–60, October 2006.

Éric Tanter. Aspects of composition in the Reflex AOP kernel. In *Proceedings of the 5th International Symposium on Software Composition (SC 2006)*, volume 4089 of *Lecture Notes in Computer Science*, pages 98–113, Vienna, Austria, March 2006. Springer-Verlag.

T. Tourwé, J. Brichau, A. Kellens, and K. Gybels. Induced intentional software views. *Elsevier Journal on Computer Languages, Systems & Structures*, 30(1-2):35–47, 2004.

Arie van Deursen, Jan Heering, and Paul Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing Co., 1996.

Eelco Visser. Stratego: A Language for Program Transformation Based on Rewriting Strategies. *Lecture Notes in Computer Science*, 2051:357–361, 2001.

Markus Voelter and Iris Groher. Handling variability in model transformations and generators. In *DSM '07: Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling*, Montreal, Canada, 2007.