Infrastructure for Domain-Specific Aspect Languages: The ReLAx Case Study

Johan Fabry
INRIA Futurs - LIFL,
Projet Jacquard/GOAL
Bâtiment M3
59655 Villeneuve d'Ascq, France
Johan.Fabry@lifl.fr

Éric Tanter *
DCC - University of Chile
Blanco Encalada 2120
Santiago, Chile
etanter@dcc.uchile.cl

Theo D'Hondt
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2,
1050 Brussel, Belgium
tjdhondt@vub.ac.be

Abstract

Domain-specific aspect languages (DSALs) bring the well-known advantages of domain specificity to the level of aspect code. However, DSALs incur the significant cost of implementing or extending a language processor or weaver. Furthermore, this weaver typically operates blindly, making detection of interactions with aspects written in other languages impossible. This raises the necessity of an appropriate infrastructure for DSALs. The case study we present here illustrates how the Reflex kernel for multi-language AOP addresses these issues, by considering the implementation of a DSAL for advanced transaction management, KALA. We first detail the implementation of KALA in Reflex, illustrating the ease of implementation of runtime semantics, syntax, and language translation. We then show a straightforward and modular extension to KALA at all these levels, and demonstrate how Reflex helps in dealing with interactions between KALA and another DSAL for concurrency management. These invaluable assets enable faster development of DSALs as well as their ability to coexist within one application, thereby removing the most important impediments to their re-emergence in the aspect community.

Keywords domain-specific aspect languages, language design and implementation, Reflex, KALA.

1. Introduction

Initial research on AOP focused on domain-specific aspect languages (DSALs), like RG [20] and AML [15]. DSALs bring all the well-known advantages of domain specificity to aspect programmers, such as conciseness and abstraction. However, this research has quickly become overshadowed by work on general-purpose languages, such as AspectJ [17]. Important reasons for this trend are the following three impediments to the growth of DSALs.

The first impediment is that the implementation of a language processor or aspect weaver requires a large amount of effort. Each DSAL tends to be implemented using an ad-hoc weaver, which is not necessarily reusable for other DSALs. Second, such weavers can be hard to extend to adapt to changes in the language. This is because they are not designed with extensibility in mind. A third major impediment is that such weavers are usually not aware of other aspect weavers that may affect the same application, and therefore blindly weave their code into the application. As a result, aspect composition, particularly when there are interactions, becomes problematic. There is no indication of interaction and possible conflicts, let alone a possibility for simple conflict resolution.

The above three impediments can however be addressed through the use of an appropriate infrastructure for DSALs. Using a platform that provides adequate support for implementing DSALs, it becomes easier to experiment with them, their extensions, as well as with interactions between aspects defined in different DSALs. This is precisely the objective of Reflex, a kernel for multi-language AOP [30]. Reflex provides as a base a large number of generic facilities for the creation of aspects, in addition to which support is included for detection and resolution of interaction conflicts, and last but not least, support for language definition and transformation based on state-of-the art DSL technologies [3]. This allows DSALs and their extensions to be implemented faster, and provides direct support for detection of interactions between different aspects, as well as for their resolution [26, 27].

This paper validates the multi-language AOP approach by studying the design and implementation of a non-trivial DSAL on top of Reflex. We consider KALA, a DSAL for advanced transaction management [11, 12]. KALA is a good candidate for this work not only because of its significant size but also because of its complexity and specific syntax and scoping rules. The in-depth discussion of the implementation of KALA in Reflex provided here shows that *(a)* Reflex provides appropriate support for DSAL development, allowing for faster and easier implementation of such languages; *(b)* the DSL technologies used by Reflex enable straightforward modular extension of DSALs; *(c)* Reflex adequately detects and reports on interactions between aspects defined in different DSALs, as well as supporting the resolution of these interactions.

The paper is structured as follows: Section 2 discusses multi-language AOP and Reflex. Section 3 introduces the domain of advanced transaction management and KALA. Section 4 gives an operational description of KALA, and discusses its implementation in Reflex. Section 5 completes the language implementation by treating both the KALA syntax definition and the assimilation of KALA code into Java code for Reflex. Section 6 evaluates our solution by considering both an extension to KALA, and a case of interaction with another DSAL. Section 7 discusses previous, related, and future work. Section 8 concludes.

* É. Tanter is partially financed by the Millenium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile.

AOSD '07 March 12-16, 2007, Vancouver

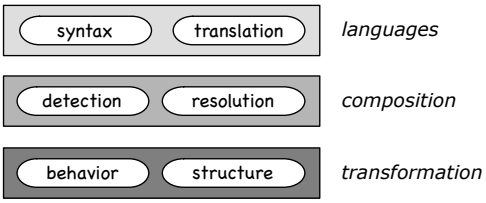

Figure 1. Architecture of a versatile kernel for multi-language aspect-oriented programming.

2. Multi-language AOP and Reflex

This section briefly introduces the necessary background concepts on multi-language AOP and the Reflex AOP kernel.

2.1 Multi-language AOP

In order to be able to define and use different aspect languages, including domain-specific ones, to modularize the different concerns of a software system, we have previously proposed the architecture of a *versatile kernel* for multi-language AOP [29] and our current Java implementation, Reflex [30].

An AOP kernel supports the core semantics of various AO languages through proper structural and behavioral models. Designers of aspect languages can experiment rapidly with an AOP kernel as a back-end, as it provides a high level of abstraction for driving transformation. Furthermore, a crucial role of an AOP kernel is that of a mediator between different coexisting AO approaches: detecting interactions between aspects, possibly written in different languages, and providing expressive means for their resolution.

The architecture of an AOP kernel hence consists of three layers (Fig. 1): a transformation layer in charge of basic weaving, supporting both structural and behavioral modifications of the base program; a composition layer, for detection and resolution of aspect interactions; a language layer, for modular definition of aspect languages. It has to be noted that the transformation layer is not necessarily implemented by a (byte)code transformation system: it can very well be integrated directly in the language interpreter [14]. As a matter of fact, the role of a versatile AOP kernel is to *complement* traditional processors of object-oriented languages. Therefore, the fact that our implementation in Java is based on code transformation should be seen as an implementation detail, not as a defining characteristic of the kernel approach.

2.2 Reflex in a Nutshell

Architecture. Reflex is our Java implementation of versatile kernel for multi-language AOP. As such, it follows the architecture of an AOP kernel (Fig. 1):

- The **transformation layer** is based on a reflective core extending Java with behavioral and structural reflective facilities. The model of behavioral reflection is based on that presented in [31], and explained in more details hereafter.
- The **composition layer** ensures automatic detection of aspect interactions, and provides expressive means for their explicit resolution. The composition facilities of Reflex were presented in [26], and recently, advanced mechanisms for declarative composition of structural aspects were introduced [27].
- The **language layer** is based on the MetaBorg approach for unrestricted embedding and assimilation of domain-specific languages [3]. Concrete syntax for the Reflex kernel API using this approach was presented in [28]. In Section 5 we report on the definition of KALA and its assimilation to Reflex using MetaBorg.

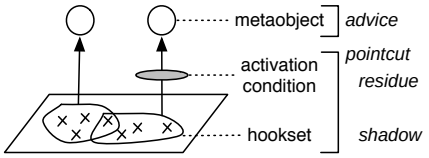

Figure 2. The link model and correspondence to AOP concepts.

Links. The central abstraction supported at the level of the kernel to drive behavioral transformation is that of explicit *links* binding a set of program points (a *hookset*) to a *metaobject*. A link is characterized by a number of attributes, among which the control at which metaobjects act (before, after, around), and a dynamically-evaluated activation condition. Fig. 2 depicts two links, one of which is not subject to activation, along with the correspondence to the AOP concepts of the pointcut/advice model. The aforementioned links are called *behavioral links* to distinguish them from *structural links*, which are used to perform structural actions [27], as briefly illustrated in Sect. 6.1 when extending KALA.

A link can therefore be seen as a *primitive aspect*, that is, a single cut/action pair. Higher-level aspects (*a.k.a.* composite aspects) typically consist of several such pairs, and may as well include structural primitive aspects (*e.g.* inter-type declarations).

Hooksets. A hookset is specified by defining predicates matching a reification of program elements, following a class-object structural model similar to that of Javassist [6]: an `RPool` object gives access to `RClass` objects, which in turn give access to their members as `RMember` objects (either `RField`, `RMethod`, or `RConstructor`), which in turn give access to their bodies as `RExpr` objects (with a specific type for each kind of expression). These objects are causally-connected representations of the underlying bytecode, offering a source-level abstraction over bytecode. Reflex is implemented as a Java 5 instrumentation agent operating on bytecode, typically at load time. During installation of behavioral links, *hooks* are inserted in class definitions at the appropriate places according to hooksets, in order to provoke reification at runtime, following the protocol specified for each link.

Metaobjects. A metaobject implements the action associated to an aspect. In Reflex it can actually be any standard Java object, whose existence may even precede the actual definition of the link (*e.g.* `System.out` can serve as a metaobject for a link). Reflex makes it possible to customize the actual protocol between the base program and metaobjects, on a per-link basis. For instance, a call descriptor can specify that the `println` method of `System.out` be called passing only the intercepted method name as parameter.

3. KALA in a Nutshell

In this section we first introduce advanced transaction models (ATMS), via two well-known models. We then present the KALA domain-specific aspect language for ATMS, illustrating its use with example code for these two advanced transaction models.

3.1 Advanced Transaction Models

Transactions are the cornerstone of concurrency management in multi-tier distributed systems. Originally designed to provide concurrency management for short and unstructured data accesses to databases, they are however now used outside of this domain. This observation is not new, and significant research has been performed to address the shortcomings of classical transactions through the use of advanced transaction models [10, 16]. An overview of these models is outside of the scope of this paper. Instead we briefly introduce what are arguably the two best-known advanced transaction

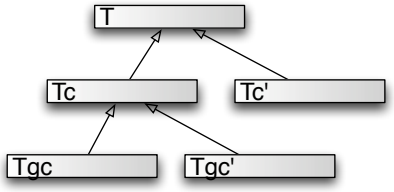

Figure 3. The nested transactions ATMS.

models: *nested transactions* [21] and *sagas* [13]. Nested transactions allow a hierarchically structured computation to be matched to a tree of transactions, while sagas can be used to split a long-lived transaction into a number of shorter steps.

Nested transactions. This model is one of the oldest and arguably the best-known ATMS [21]. It enables a running transaction T to have a number of *child transactions* T_c (as shown in Fig. 3). Each T_c can *view* the data used by T. This is in contrast to classical transactions, where the data of T is not shared with other transactions. T_c may itself also have a number of children T_{gc}, forming a tree of nested transactions. When a child transaction T_c commits its data, this data is not written to the database, but instead it is *delegated* to its parent T, where it becomes part of the data of T. If a transaction T_x is the root of a transaction tree, *i.e.* it has no parent, its data is committed to the database when it commits. Another characteristic of this model is that if a child transaction T_c aborts, the parent T is not required to abort, *i.e.* when it ends it may choose to either commit or abort.

Sagas. The model of sagas [13] is, next to nested transactions, one of the oldest ATMS and also arguably one of the most referenced ATMS in the community. Sagas is tailored towards long-lived transactions. Instead of one long transaction T, a saga S splits T into a sequence of sub-transactions T_1 to T_n (as shown in Fig. 4). Each sub-transaction is a normal classical transaction and this sequence is executed completely before the saga commits. To abort or rollback a running saga S, the currently running sub-transaction T_i is aborted and the work of already-committed transactions T_1 to T_{i-1} has to be undone, as their results have already been committed to the database. To allow this, the application programmer has to define for each sub-transaction T_i a *compensating transaction* C_i that performs a semantical compensation action. To undo the work of T_1 to T_{i-1}, C_1 to C_{i-1} are ran by the runtime transaction monitor in inverse sequence, *i.e.* starting with C_{i-1}.

The ACTA Formalism. In addition to a large amount of advanced transaction models –each addressing a specific subset of the shortcomings of classical transactions– a formalism has been developed for advanced transaction models. This formalism is called ACTA [8]. ACTA allows a wide variety of advanced models to be described formally. An in-depth treatment of ACTA is outside of the scope of this paper. Suffice it to say that ACTA specifications for a given model formally describe properties that are exhibited by transactions in this model.

Towards aspects. From the viewpoint of an application, an ACTA specification can be seen as formally defining the properties of the *concern* of advanced transaction management. This leads us to aspect-oriented programming. Indeed, transaction management is a well-known aspect, and a significant amount of work has already been done to aspectize transaction management [19, 22, 25]. However, none of this work goes beyond classical transactions. Using the ACTA formalism as a base, we have developed a DSAL for ATMS: KALA, which we present next.

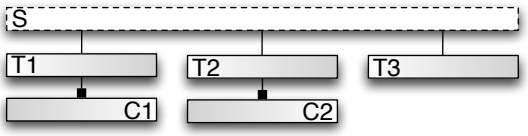

Figure 4. The sagas ATMS.

3.2 KALA: an Aspect Language for Advanced Transaction Models

KALA is a domain-specific aspect language for the domain of *advanced* transaction models, based on the ACTA formalism. KALA reifies the concepts of the ACTA formal model as statements in the language. Our implementation of KALA targets Java applications: a base Java application can be made transactional, using KALA, with transactions that exhibit the properties of an advanced transaction model. An in-depth treatment of KALA, the design process and the tradeoffs made is provided in [12]; in this paper, we solely provide a short description of the language using two example program fragments: one for nested transactions and one for sagas.

KALA Programs. A KALA program declares transactional properties (discussed below) for a number of transactions based on the life-cycle of a given transaction. As is the norm in multi-tier transactional systems, the life-cycle of every transaction coincides with the life-cycle of a method. The transaction begins when the method begins, commits when the method ends normally, and aborts if the method ends with a given type of exception. A KALA declaration consists of a signature and body. The signature identifies a method, and therefore a transaction, possibly using wildcards, similar to type and method name patterns in AspectJ [17].

Consider the KALA code shown in Fig. 5. Line (1) is the *KALA signature*, which identifies the transactional methods. As a result, all data accesses to shared data within the method (and within methods called by this method) are included in the transaction. To indicate that instances of a given class contain shared data, *i.e.* that they are transactional objects, the class must implement the `Resourceable` interface. This interface declares one method: `getPrimaryKey()`, that should return a unique identifier for the object. Note that having to implement this interface implies an incomplete separation of concerns. This is because the implementation has to be done at the base level of the application, and not at the level of the transaction aspect. We address this incomplete separation of concerns in Section 6.1.

In the *KALA body*, transactional properties are declared for this transaction, and possibly for other transactions. The properties take effect at given times in the life-cycle of the transaction: properties can be declared to apply at begin time, commit time and abort time. This is done by placing these declarations, which are KALA statements, in a `begin` block (5)-(6), `commit` block (7)-(9) and `abort` block (10)-(11), respectively. Outside of these blocks, a number of statements can be placed in the *preliminaries* (2)-(4). We shall talk about the preliminaries later.

Transactional properties. The transactional properties of a method that can be declared to apply either at begin, commit or abort time are taken from the ACTA formal model: *views*, *delegation* and *dependencies*. Each of these properties is reified as a statement in KALA, respectively `view`, `del` and `dep` statements.

The view property declared in (6) states that the current transaction, which is a child transaction, can see the data of its parent transaction. This property is established when the transaction begins. The delegation property of (7) states that upon commit, the

```
util.strategy.Hierarchical.child*() {                    1
 alias(parent Thread.currentThread() );                  2
 name(self Thread.currentThread());                      3
 groupAdd(self "ChildrenOf"+parent);                     4
 begin { dep(self wd parent); dep(parent cd self);       5
         view(self parent); }                            6
 commit { del(self parent);                              7
          name(parent Thread.currentThread());           8
          terminate("ChildrenOf"+self); }                9
 abort { name(parent Thread.currentThread());           10
         terminate("ChildrenOf"+self);} }               11
```

Figure 5. KALA code for children in nested transactions.

```
Cashier.transfer(Account src, Account dest, int a){
 alias(saga Thread.currentThread());
 autostart(Cashier.transfer(                             12
  Account src, Account dest, int a)<dest, src, a>        13
   { name(self "CompOf"+saga); } );                      14
 begin{ alias(comp "CompOf"+saga);
        dep(saga ad self); dep(self wd saga);
        dep(comp bcd self); }                            15
 commit{ alias(comp "CompOf"+saga);
         dep(comp cmd saga); dep(comp bad saga);}}       16
```

Figure 6. KALA code for a step in a bank transfer saga.

child transaction delegates its data changes to its parent transaction. This concisely expresses the most important characteristics of nested transactions as discussed previously.

A dependency statement, `dep`, sets relationships between points in the life-cycle of two transactions. For example, a dependency can force a transaction to commit if another transaction aborts (the `cmd` dependency), it can restrict one transaction to start only if another transaction has committed (the `bcd` dependency), or to start only if another transaction has aborted (the `bad` dependency) . Combinations of dependencies can be used to, for instance, sequence different transactions or trigger the beginning of a compensating transaction. A wide variety of dependencies have been defined in the ACTA formal model, and are available in KALA. We do not discuss these in detail here, instead we refer to [8, 12]. The dependency `self wd parent` (5) states that if the parent aborts before this transaction ends, then this transaction will be forced to also abort. `parent cd self` states that if the parent wants to commit, it has to wait until this transaction has ended.

Naming transactions. Dependencies, views and delegation need to be able to *denote* the two transactions they affect; therefore there is a need for a variable binding mechanism. Within KALA code, such a binding is known as an *alias*. An alias is looked up through the use of a global naming service, which is declared using the `alias` statement (2). This statement takes as argument the alias for a transaction, *i.e.* the variable name, and a Java expression that evaluates to a key that is used to look up the transaction reference in the name service. This expression, as well as all expressions we mention in the remainder of this section, has access to the actual parameters of the method and to aliases which have already been resolved. Special cases are the alias `self`, which is always bound to the currently running transaction, and the *null transaction*, which is the result of a lookup failure. KALA statements which have as an argument the null transaction fail silently.

Adding transactions to the naming service is performed using the `name` statement, which takes as argument an alias and a Java expression that evaluates to the key for the naming service. On Fig. 5, the current thread is first used as a key to lookup the parent transaction (2), then to register the current transaction (overriding the binding) (3), and finally, upon commit or abort, the parent binding is restored (8),(10). The scope of aliases within a KALA declaration follows the usual lexical scoping rules: aliases obtained in the preliminaries of a declaration are accessible thoughout the remainder of the KALA code for that declaration; aliases placed in begin, commit and abort blocks are only accessible at that time.

Grouping transactions. KALA provides support for named groups of transactions. A transaction can be added to a group using the `groupAdd` statement: (4) adds the current transaction to the group of children of the `parent` transaction. All KALA statements have an overloaded behavior for groups, *e.g.* setting a view from a transaction to a group of transactions implies setting the view to each member of the group. The only non-obvious case is when a group is a destination of a delegation statement. As semantically this has no sense –delegating some changes to a group of transactions–, a failure is produced. Note that for conciseness in the remainder of the text, we shall refer to the collection of `name`, `alias` and `groupAdd` statements as naming statements.

Terminating transactions. Because dependencies may refer to transactions which have already ended, it is impossible to perform automatic garbage collection of names and dependency relationships when transactions have ended. Instead the KALA programmer is made responsible for such cleanup operations. This is performed through the `terminate` statement, which takes as argument a Java expression. This expression is resolved to a name of the transaction or group of transactions to be collected. Termination of transactions can be performed within a `begin`, `commit` and `abort` block. For instance, (9) and (11) state that if a nested transaction finishes (by commit or abort), it terminates the group of its child transactions. Note that if a transaction is terminated when it has not yet ended, it is immediately forced to rollback.

Autostarting transactions. An important number of advanced transaction models require that, when some properties are satisfied, a new transaction is automatically started. An example is the use of compensating transactions in the Sagas model, which we have discussed above. We term these kinds of transactions *secondary transactions*. A secondary transaction runs outside of the main control flow of the application, and does not need to be run in order to have a successful completion of the original transaction. KALA provides support for secondary transactions through the `autostart` statement: it specifies the signature of the method corresponding to the secondary transaction to start in parallel, a list of actual parameters, and optionally a nested KALA declaration for this transaction. Autostarts are specified in the preliminaries and their nested KALA code has access to all aliases defined in the preliminaries, following the rules of lexical scope.

An example of an autostart is given in Fig. 6, which shows part of the KALA code for one step of a bank transfer saga. This step performs the actual bank transfer between two bank accounts. The compensating transaction for this transfer (12)-(14) is the inverse operation, *i.e.* calling the `transfer` method with the `src` and `dest` arguments swapped (13). The dependencies set in (15) and (16) restrict the compensating transaction to run only if the saga rolls back after this step has committed. Without these dependencies in place, the compensating transaction defined in the autostart would immediately start, and run in parallel with the transaction of the top-level KALA declaration, which is not the desired semantics.

Summary. KALA is an expressive aspect language for ATMS because it is the direct realization of the ACTA formalism; it provides clear advantages over general-purpose aspect languages, such as conciseness, coherent scoping rules, and appropriate abstractions corresponding to the domain at stake, among others.

4. ReLAx: Implementing KALA in Reflex

We now enter in more details with respect to the working of KALA and its implementation in Reflex.

4.1 Operational Description of KALA

Generally, transactions are managed at runtime by a component known as a *TP monitor*, whose task is to manage concurrent accesses to shared data: individual transactions notify the TP monitor of their intent to read or write shared data, and the TP monitor allows or disallows these accesses, to prevent race conditions.

KALA is no exception to this rule. KALA works in close cooperation with a TP Monitor, called *ATPMos*. ATPMos was specifically developed for advanced transaction models and is also based on the ACTA formalism. At runtime, beyond the normal tasks of a TP Monitor, ATPMos keeps track of dependencies and view relationships and is able to perform delegation between transactions; it also provides the naming services required by KALA (naming, grouping, termination). A detailed discussion of ATPMos is outside the scope of this paper (more information is in [11]).

At each point in the life-cycle of a transaction, the responsibilities of KALA therefore are: To instruct ATPMos to place dependencies and views, to perform delegation and termination, and to coordinate with ATPMos to ensure that dependencies are met. While a transaction runs, KALA informs ATPMos of all reads and writes to shared data, before they are performed. Autostarts are entirely managed by KALA; ATPMos provides no specific support for them: it sees them as normal transactions. The flow chart in Fig. 7, discussed below, outlines how KALA works.

Preliminaries. First, general setup is performed: obtaining a unique transaction identifier from ATPMos, and setting up the alias environment, which keeps bindings for aliases. The environment is initialized with the binding of `self` to the obtained transaction identifier, as well as with the bindings of formal parameters of the transactional method (as specified in the KALA code) to their actual values. Alias environments can be nested: if a lookup fails in an environment, it is performed in the parent, if present.

Next, the naming statements of the preliminaries are executed. As a rule, all naming is performed at the beginning of a phase, in the sequence of the statements in the KALA code. Recall that `alias` statements add bindings from names to transaction or group identifiers in the alias environment, `name` statements add these bindings to the naming service of ATPMos, and `groupAdd` statements add transaction identifiers to the grouping service of ATPMos.

Finally, for each `autostart` statement a thread is defined that calls the method specified in the autostart statement. This transactional method is parameterized by the KALA body nested in the `autostart`, overriding any other KALA declarations for that method. Furthermore, the current alias environment is given as a parent environment of the created transaction: this allows the KALA declarations in the `autostart` to refer to aliases defined in the enclosing KALA definition. The autostart thread is started, and allowed to run until its preliminaries are finished. This allows for the registration of a global name that can be used in the enclosing KALA code, such as in Fig. 6, where the autostart of a saga registers itself with a name, which is then looked up in the `begin` and `commit` block of the enclosing KALA definition.

Begin. In the begin phase, a nested alias environment is created, and naming operations are performed. Then, dependencies are set in ATPMos, as they may impact the begin of an autostart or of the current transaction. For instance, in Fig. 6, the autostart is a compensating transaction that may not begin unless this transaction has committed and the saga aborts, which is specified both in the begin and abort block. After dependencies have been set, the autostarts are allowed to proceed with their begin phase.

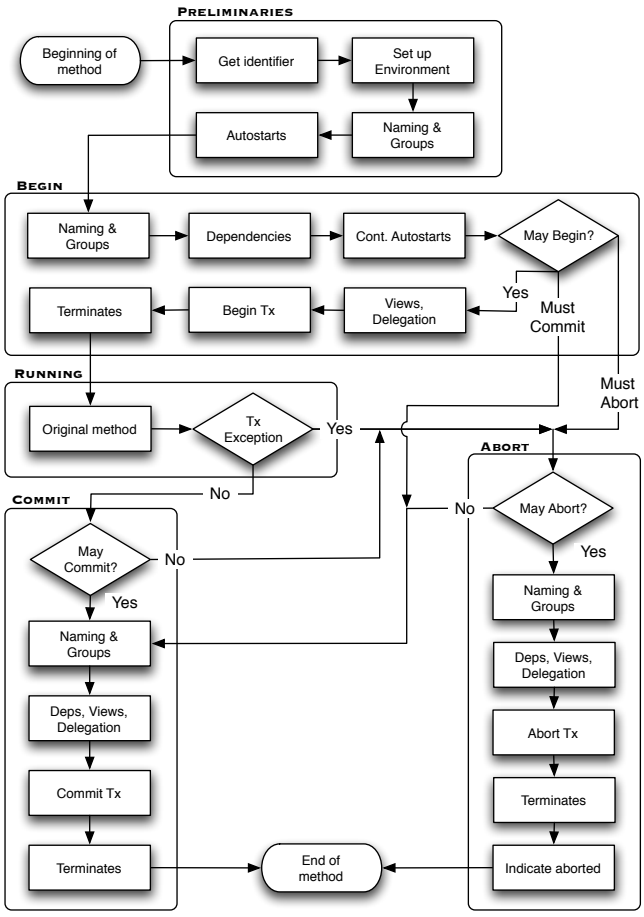

Figure 7. Flow chart of a KALA transaction.

At this point, ATPMos is asked if, according to the dependencies currently placed on this transaction, it may begin; otherwise, this call blocks. An example of this is the case of a compensating transaction in a saga (Fig. 6). This is because the compensating action should only be performed when the saga aborts. The call to ATPMos may finally return with three possible values (Fig. 7): the transaction may be allowed to begin, or it may be immediately be forced to commit or to abort. The latter two cases occur if the dependencies currently placed require immediate commit or abort of the method. The compensating transaction mentioned above is such a case: if the overall saga has committed, it will never need to run, and therefore has to immediately abort. If the transaction is allowed to begin, views are set and delegation is performed, ATPMos is informed that the transaction is about to begin, and termination is performed. If the transaction must commit or abort, control flow proceeds in the corresponding phases.

Running. The running phase of the transaction corresponds to running the code of the method, *i.e.* the application logic, but with an interception of all getters and setters of transactional objects. The interception calls ATPMos to inform it that this shared data is going to be read or written. This call may block, in order to prevent race conditions, and may throw a transactional exception, *e.g.* in case that a deadlock needs to be broken. If such an exception is thrown, either by ATPMos, or by the application logic, the control flow proceeds with the abort phase.

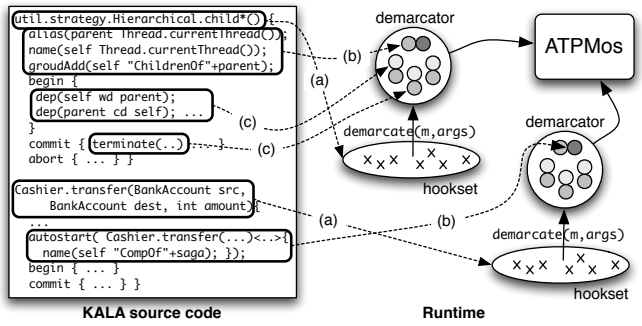

Figure 8. Mapping of KALA declarations to runtime objects.
There is one link per KALA declaration. Method signatures are mapped to hooksets (a). Demarcators are parameterized by configuration objects reifying KALA statements (a pair for preliminaries (b), and one for each block (c)); they are given control upon transactional method executions, and interact with ATPMos.

Commit. The commit phase starts with a choice point for the enforcement of dependencies, similar to the choice point in the begin phase. If the transaction may commit, the actions are straightforward; the only difference with the begin phase is that dependencies are done after the choice point, because they are considered to hold only if the transaction actually commits. If it must abort, control flow proceeds with the abort phase.

Abort. The abort phase is mostly identical to the commit phase. There are two differences, which we discuss here. First, if the transaction is forced to commit, control flow does not proceed to the beginning of the commit phase, but it skips the choice point. This is to avoid loops if a transaction is both forced to abort and forced to commit because of conflicting dependencies. Although such a conflict might be a bug in the specification, we have chosen to let the transaction end instead of letting the application loop endlessly [11]. Second, a transactional exception is thrown to the caller of the method at the end of the phase. This is done to inform the caller that this transaction ended in an abort, *i.e.* that the work expected of this method was not performed successfully. Note that the caller may also be a transactional method, and hence will also abort, unless the exception is caught by the application logic.

4.2 Reflex Definitions for KALA

Having discussed the operational description of KALA, we now give an overview of how this is implemented using Reflex.

Declaration links. We consider one primitive KALA aspect per KALA declaration. The cut of a KALA aspect is defined by the method signature of the declaration. In terms of execution points, this corresponds to a Reflex hookset (Fig. 8 (a)), matching the execution of the methods specified by the pattern. The action of a KALA aspect occurs *around* the specified methods, and is implemented in a Java object, called a `Demarcator`. There is one Reflex link per KALA declaration, binding the hookset to the demarcator, and specifying the information that must be passed at runtime: the name of the method and its actual parameters (Fig. 8).

Demarcator. The runtime behavior of KALA programs is implemented in the `demarcate` method of a `Demarcator`. This method is generic for all KALA programs: it just ensures the correct control flow, following Fig. 7, interacting with ATPMos. Actions that are specific to KALA programs are delegated to a number of KALA configuration objects that reify KALA statements and interpret them, as discussed next. There is one demarcator per KALA declaration: a demarcator is instantiated with a number of configuration

```
class ReLAxConfig extends ReflexConfig {
 void install(Hookset hs, String[] formals,          17
      NamingEval p_nt, AStart[] as, NamingEval b_nt,
      KProps b_props,  NamingEval c_nt, KProps c_props,
      NamingEval a_nt, KProps a_props) {
  Demarcator d = new Demarcator(formals, p_nt, as,   18
      b_nt, b_props, c_nt, c_props, a_nt, a_props);
  BLink demarcate = Links.get(hs, d);                19
  demarcate.setControl(Control.AROUND); ...          20
  demarcate.install(); } ... }                       21
```

Figure 9. Method of the configuration class that installs a link for a given reified KALA declaration.

objects, as well as with the list of formal parameters of the transactional method. A `Demarcator` is reentrant so it is shared between all running instances of a given KALA declaration.

Configuration objects. There are three categories of configuration objects, corresponding to different KALA statements:

- **Transactional properties** are a simple reification of dependencies, views and delegation statements as structured data. A `KProps` object is a triplet of bi-dimensional string arrays, one per kind of property. For instance, the statement `dep(self wd parent)` is represented as an array {`"self","wd","parent"`}, within the array of dependencies. The actual interpretation of these values consists in looking up the identifiers in the alias environment, then calling ATPMos to set the property (if a group is involved, the property is set for each member of the group). Lookup failures are reported as errors and no action is taken.
- **Naming evaluators** are objects interpreting a number of naming and termination statements. Naming and termination statements are not pure data, they include expressions that need to be evaluated at runtime, including identifiers that must be looked up in the alias environment. Therefore a set of naming and termination statements is represented as a dedicated Java class implementing their expressions. This class is a subclass of `NamingEval`, which defines generic evaluation and lookup mechanisms.
- **Autostarts** are represented as runnable objects, subclass of `AStart`, whose `run` method calls the method indicated in the autostart statement. The values for arguments to this method call are looked up in the alias environment. Furthermore, the `AStart` object sets the `Demarcator` object of the method that it calls to a new metaobject. This new object is configured by the nested KALA declaration of the autostart and the alias environment it contains has as parent the current alias environment.

A `Demarcator` is initialized with four pairs of configuration objects, one for each of the sections of a KALA declaration (Fig. 8): a naming evaluator and autostarts for the preliminaries (b), and for begin, commit and abort blocks a naming evaluator and a transactional properties object (c). Fig. 9 shows the `install` method of the base configuration class used for ReLAx: given all the above parameters, the method creates the demarcator (18) and the link binding the given hookset to the demarcator (19). Link attributes, like control, scope, etc., are then set (20), before the link is installed (21).

Transactional objects. In addition to the above, KALA includes a secondary aspect: that of intercepting executions of getter and setter methods of classes that implement the `Resourceable` interface. This aspect is implemented as a single link, binding a hookset matching the above executions to other methods of the `Demarcator` (`preWrite` and `preRead`). These methods simply inform ATPMos of reads and writes to shared data, as discussed previously. Because these methods are stateless and reentrant for all

```
module Kala imports Java-15-Prefixed Pattern                    (22)
 exports context-free syntax
  KDecl* ->  CompilationUnit                                    (23)
  FQMPattern KBody                -> KDecl                      (24)
  "{" Prelim? BeginBlock?
     CommitBlock? AbortBlock? "}" -> KBody                      (25)
  PrelimStm*                      -> Prelim                     (26)
  "begin"  "{" BlockStm* "}" -> BeginBlock                      (27)
  "commit" "{" BlockStm* "}" -> CommitBlock                     (28)
  "abort"  "{" BlockStm* "}" -> AbortBlock                      (29)
  AStartStm     -> PrelimStm                                    (30)
  NamingStm     -> PrelimStm                                    (31)
  NamingStm     -> BlockStm                                     (32)
  DepStm        -> BlockStm                                     (33)
  ViewStm       -> BlockStm                                     (34)
  DelStm        -> BlockStm                                     (35)
  TermStm       -> BlockStm                                     (36)
  "autostart" "(" MethSig ASActuals
                        KBody ")" ";" -> AStartStm              (37)
  "alias" "(" KBinding ")" ";"          -> NamingStm            (38)
  "name"  "(" KBinding ")" ";"          -> NamingStm            (39)
  "groupAdd" "(" KBinding ")" ";"       -> NamingStm            (40)
  "dep" "(" JavaId JavaId JavaId ")" ";"-> DepStm               (41)
  "view" "(" Min? JavaId JavaId ")" ";" -> ViewStm              (42)
  "del" "(" JavaId JavaId  ")" ";"      -> DelStm               (43)
  "terminate" "(" JavaExpr  ")" ";"     -> TermStm              (44)
  JavaId JavaExpr                       -> KBinding             (45)
```

Figure 10. Syntax definition of KALA in SDF.

KALA programs, the link is installed only once, when the first KALA program is being woven.

5. Definition and Assimilation of KALA

After this informal discussion of both the operational semantics of KALA and the way it is supported in Reflex as a framework, we present how the actual KALA language is defined in our infrastructure. This includes the concrete syntax definition, as well as the automatic transformation of a KALA program into Reflex configuration code in plain Java.

5.1 Declarative Syntax Definition

The syntax definition of KALA is performed using SDF, a modular syntax definition formalism [33]. Fig. 10 shows the syntax definition of KALA in SDF: it is defined as an SDF module importing the Java 5 syntax as well as a module for pattern syntax (taken from the SDF definition of AspectJ [2]), shown in line (22). SDF productions are declared in the reverse manner from the traditional BNF notations, as illustrated in Fig. 10 where (23) states that a number of KALA declarations are valid as a `CompilationUnit` non-terminal. This non-terminal is the root of the Java language SDF definition, therefore we are actually extending the Java language with the KALA syntax. Note that although this allows Java and KALA code to be mixed in one file, the KALA assimilator presented below only processes KALA code.

A KALA declaration consists of a fully-qualified method pattern followed by a body (24); a `KBody` is made up of 4 optional sections (25): preliminaries, begin block, commit block, and abort block. Preliminaries are a list of `PrelimStm` (26), while blocks are made up of `BlockStm` (27)-(29). A `PrelimStm` can be either an autostart (30) (defined on line (37)), or a naming statement (31). A naming statement is also valid as a `BlockStm` (32), along with dependency, view, delegation, and termination statements (33)-(36). A naming statement can either be an `alias` (38), a `name` (39), or a `groupAdd` (40). These statements include binding expressions, binding a Java identifier to an expression (45). Dependency, view, delegation, and termination statements also make use of the imported Java non-terminals `JavaId` and `JavaExpr` (41)-(44).

```
AssimKDecl :                                                    (46)
KDecl(meth, KBody(prelim, begin, commit, abort)) ->
 |[ Hookset    ~hs      = ~<AssimMethSig> meth ;                (47)
    String[]   ~formals = ~<AssimFormals> meth;                 (48)
    NamingEval ~pre-nts = ~<AssimNTs> prelim;                   (49)
    AStart[]   ~as      = ~<AssimAStarts> prelim;               (50)
    NamingEval ~b-nts   = ~<AssimNTs> begin;                    (51)
    KProps     ~b-props = ~<AssimProps> begin;                  (52)
     // ...same for commit and abort blocks...
    this.install(~hs, ~formals, ~pre-nts, ~b-nts,               (53)
     ~b-props, ~c-nts, ~c-props, ~a-nts, ~a-props); ]|
 where <newname> "hs" => hs                                     (54)
  ;  // etc. for all variable names
```

Figure 11. Rule for assimilating a KALA declaration.

The KALA syntax definition is very compact and declarative thanks to the SDF notation. We have only omitted a few lines of details such as the method patterns. It is also modularly extensible, as will be illustrated in Section 6.1.

5.2 Reflex Code Generation

With the SDF definition above, the MetaBorg toolset generates a parser for KALA that produces an abstract syntax tree in the ATerm format [32]. The actual AST nodes that are produced for the non-terminals of the grammar are specified using constructor declarations, omitted here for conciseness. The AST is then processed by an assimilator defined declaratively using the Stratego language [34]. By defining assimilation rules, KALA declarations are converted into Reflex configuration code, in plain Java.

Assimilating declarations. Fig. 11 shows the main assimilation rule, which deals with KALA declarations. An assimilation rule has a name (46), and specifies how a term (AST node) matching the pattern on the left-hand side is transformed into the right-hand side. We make use of the embedding of Java within Stratego, so the result of the transformation is directly written in Java code between the `|[` and `]|` separators [3]. Within this block, *metavariables* are referred to using the `~` escape. A `where` clause (54) can be specified for applying further rules to some elements and bind them to variables which can be used in the right-hand side of the rule.

The assimilation rule of a KALA declaration generates the hookset, the formal parameters array, and the configuration objects that are needed to create the corresponding link. Then the `install` method of Fig. 9 (17) is called (53). These statements are inserted in the `initReflex` method of a class extending `ReLAxConfig` (Fig. 9). A configuration class is instantiated and called at startup in order to perform the necessary link definitions. We generate one configuration class per KALA source file. A single source file can of course contain more than one declaration, resulting in a Reflex configuration class that installs more than one parameterized link.

Assimilating parameters. The different configuration objects in Fig. 11 are denoted by an identifier in order to refer to them when calling the `install` method (53). To ensure hygiene, identifiers are automatically generated by Stratego. This is why the hookset variable in (47) is the metavariable `~hs`, which is determined in the `where` clause of the rule, which applies the `<newname>` utility rule to the `"hs"` symbol (54). The result of this transformation is then bound to the variable we use in the Java code. This means Stratego will generate hookset variable names `hs_0`, `hs_1`, etc., as needed.

Line (47) specifies that the right-hand side of the assignment for the hookset is obtained by applying the `AssimMethSig` rule to the `meth` term (the AST node representing the method signature). The list of formal parameters of the method is also obtained by

```
AssimProps :
 stms ->
  |[ new KProps(new String[][]{~deps },          55
                new String[][]{~views},
                new String[][]{~dels }) ]|
  where <try(filter(AssimDep))>  stms => deps     56
      ; <try(filter(AssimView))> stms => views    57
      ; <try(filter(AssimDel))>  stms => dels     58

AssimDep :                                        59
 DepStm(Id(src), Id(dep), Id(dest)) ->
   var-init |[ { "~src", "~dep", "~dest" } ]|     60
 // similar for views and delegation
```

Figure 12. Rule for assimilating transactional properties.

```
AssimNTs :
stms ->
 |[ new NamingEval(){                                          61
     void evalNaming(KALAEnv e,TxManager t){~n_stms}           62
     void evalTerm(KALAEnv e,TxManager t){~t_stms}} ]|         63
  where <try(filter(AssimNaming))> stms => n_stms              64
      ; <try(filter(AssimTerm))> stms   => t_stms              65
AssimNaming :
 NameStm(KBinding(Id(id), expr)) ->
   |[ this.nameOp(e, t, "~id", ~expr_ok); ]|                   66
 where <topdown(try(LookupId))> expr => expr_ok                67
 // similar for alias and groupAdd statements
AssimTerm :
 TermStm(Term(expr)) ->                                        68
   |[ this.terminateOp(e, t, ~expr_ok); ]|
 where <topdown(try(LookupId))> expr => expr_ok
LookupId :
 ExprName(Id(id)) -> |[ e.lookup("~id") ]|                     69
```

Figure 13. Rules for assimilating naming and termination.

applying a rule to this same term (48). Following this, the eight configuration objects (Sect. 4.2) needed are obtained via application of dedicated rules: the preliminary naming statements (49), the autostart objects (50), the naming and termination statements of the begin block (51), its transactional properties (52), etc.

Assimilating properties. Statements that deal with transactional properties –*i.e.* dependencies, views and delegation– are assimilated into a configuration object `KProps` (Fig. 12). A `KProps` object is a bi-dimensional array of strings, as explained in Sect. 4.2. The creation of this array is shown in (55); the content of each column in this configuration object is obtained via applying other assimilation rules, one for each type of property: dependencies (56), views (57), and delegation (58). The use of the `try` and `filter` strategies ensures that the rule is applied to all terms (the statements) and that the process goes on if a term does not match. Fig. 12 shows the assimilation rule for dependencies (59): if a statement is a dependency, it is assimilated into a variable initializer with the three corresponding values (source, dependency, and destination) (60).

Assimilating naming and termination. Naming and termination are more complex statements to assimilate (Fig. 13), because they directly relate to the scope of identifiers in KALA. For each part of a KALA declaration (preliminaries, begin, commit and abort), a `NamingEval` object is created and passed as parameter to the `Demarcator` (recall Sect. 4.2) (61). A naming evaluator has two methods, `evalNaming` and `evalTerm`, which are filled in with statements generated by the assimilation of naming (62) and termination (63), respectively. The application of these assimilations is defined in the `where` clause of the main assimilation rule (64)(65).

```
module KalaT imports Kala                                     71
 exports context-free syntax
   "transactional" TypePattern
                   PKMethod? ";" -> KDecl                     72
   "(" JavaId ")"                -> PKMethod                  73
```

Figure 14. SDF module extending KALA with transactional declarations.

As an example, Fig. 13 shows the case of a `name` statement (the operation is similar for `alias` and `groupAdd`). The generated statement is a call to the `nameOp` method defined in the superclass `NamingEval`, which takes as parameter the current environment and transaction manager, the name to bind, and the expression to which the name should be bound to (66). Note however that the expression is processed in order to replace all occurrences of identifiers with a lookup for the identifier in the alias environment (67). This is because a naming statement can include aliases and formal parameters of the method (recall Sect. 3.2); these names are not valid in the generated Java method, so they are transformed into an alias environment lookup expression (69). The assimilation of a termination statement is very similar (68).

6. Evaluation

In this section we evaluate the benefits of our infrastructure by considering an extension to the KALA language, and discussing a scenario of composition of KALA with another domain-specific aspect language for concurrency management.

6.1 Extending KALA

We first illustrate the extensibility of our implementation by considering a simple extension to the KALA language. Recall from Section 3.2 that the identification of transactional objects is made explicitly by the programmer: classes of transactional objects must implement the `Resourceable` interface, which declares a `getPrimaryKey()` method. This method returns a unique identifier for a transactional object.

The extension we consider here is to provide transactional declarations in KALA, such as:

```
transactional *Data;
```

to specify that all classes whose name ends with `Data` should be made transactional, automatically. If a class of transactional objects already implements a method that can serve the purpose of `getPrimaryKey`, this can be specified:

```
transactional *Data (getID);                                  70
```

The `getID` method will be called by a generated `getPrimaryKey` method, which therefore just serves as an adapter for the resourceable protocol on which KALA relies.

In the following, we first give the syntax of this extension, then describe how to operationally implement the transformation with Reflex. Finally, we describe the assimilation of the language extension with Stratego. This example illustrates the conciseness and modularity of domain-specific aspect language extensions.

Syntax definition. The syntax extension is very concise, as illustrated in Fig. 14. A new SDF module extending the KALA module of Fig. 10 is defined (71): it simply gives an alternative production for the `KalaDecl` non-terminal, which corresponds to transactional declarations (72). Classes are identified by a type pattern, which is a non-terminal inherited from the `Pattern` module used for the standard KALA language (and which comes from the SDF of AspectJ [2]). Optionally, the primary key method is specified, as a Java identifier (73).

```
class TrAdd implements SMetaobject {
  String pkmethod = null;
  TrAdd(String m){
    if(m.length != 0)
      pkmethod = "Object getPrimaryKey(){" +          74
                 "return this." + m + "();}";  }
  void handleClass(RClass c){
    c.addInterface(RClass.forName("Resourceable"));  75
    if(pkmethod != null) c.addMethod(pk_method);     76
} }
```

Figure 15. The implementation of the TrAdd metaobject.

AssimKDecl :

```
TransDecl(type, Some(PKMethod(Id(x)))) ->            77
 |[ ClassSelector ~cs = ~<AssimClassSel> type ;      78
    SLink ~sl = Links.get(~cs, new TrAdd("~x"));     79
    ~sl.install(); ]|                                80
 where <newname> "cs" => cs; <newname> "sl" => sl
```

Figure 16. Rule for assimilating transactional declarations.

Operational description. For classes matching the type pattern, Reflex must add the `Resourceable` interface, and if so specified, generate the standard primary key method that calls the existing one. Both steps are easily implemented using the structural abilities of Reflex [30]. A structural link is defined, binding a class selector that matches classes according to the given type pattern to a metaobject that performs the operations above. The implementation of `TrAdd` is straightforward (Fig. 15): if given a non-empty string when created, the metaobject builds a string representing the source code of the primary key method to add (74). The action of the metaobject, defined in `handleClass` simply consists of adding the resourceable interface (75) and, if necessary, the generated method (76).

Assimilation. The assimilation of the KALA extension is also defined modularly and concisely. A new Stratego module importing the KALA assimilation module defines an alternative assimilation rule for KALA declarations (Fig. 16). The parse tree node corresponding to a transactional declaration is `TransDecl` (77). The assimilation consists in first creating the class selector corresponding to the type pattern (78), and then creating a structural link with the class selector and a `TrAdd` metaobject created with the primary key method name (79). Finally, the structural link is installed (80).

6.2 Composing KALA

We now illustrate a major advantage of using a versatile kernel for multi-language AOP as discussed in Sect. 2.1: the detection and resolution facilities provided to handle interactions between aspects defined in different languages. Aspect composition is a multi-faceted issue, and our objective here is not to cover it exhaustively; in-depth discussion of aspect composition in Reflex can be found in [26] and [27].

In previous work, a library for concurrent programming providing the Sequential Object Monitor (SOM) abstraction was proposed [5]. SOM is implemented as a Java library, and also has a small DSAL for configuring it. With SOM, one can specify which objects have to be turned into monitors, and in addition specify a *scheduler* that has complete control over the scheduling strategy of concurrent requests over the monitor it is associated with. SOM presents several advantages over hand-coding this functionality, in particular with regard to modularity and efficiency. A SOM specification in the DSAL is very simple:

```
schedule: BufferData with: GetPriorityScheduler;
```

```
LinkSelector kalaLinks = new LinkSelector(){              81
 public boolean accept(Link l){
  return l.getProperty(Reflex.LANG).equals("KALA");}};  82
LinkSelector somLinks =  /* similar */;                  83
Rules.declareError(kalaLinks, somLinks,                   84
  "SOM and KALA cannot affect the same objects");         85
```

Figure 17. Error declaration when KALA and SOM links interact.

This specification ensures that (non-thread safe) `BufferData` objects are turned into object monitors whose scheduling policy gives priority to `get` requests (in addition to ensuring the usual conditional synchronization of `put` and `get` requests on the buffer).

Both KALA and SOM can be used in a single application, if there is no interaction. Here by interaction we mean a shared join point. Basically, this refers to the fact that it is erroneous to have an object being both a transactional object from the KALA point of view, and an object monitor from the SOM point of view. This is because both aspect languages, although with a different focus and scope, deal with concurrency. Note that a similar observation has also been made by Kienzle and Guerraoui in [19].

Suppose the above SOM aspect is defined in a file `buf.som`, and a KALA program including the transactional declaration of (70) is defined in a file `data.kala`. When run, since `BufferData` actually matches both the SOM aspect and the type pattern of the transactional declaration of KALA, Reflex *detects and reports* this interaction to the user when weaving on the `get` and `put` methods:

```
[WARNING] don't know how to compose on BufferData.get():
     - SOM (buf.som, line 1)
     - KALA (data.kala, line 1).
[WARNING] composing arbitrarily (sequence).
...weaving goes on...
```

The programmer is therefore *informed* of the interaction, and can decide which measure to take. In the case of SOM and KALA, as we said, it is semantically incorrect to have *both* apply to the same objects, although they can co-exist in an application. Reflex makes it possible to declare a *mutual exclusion* between links [26], that can result either in a warning or in an error. In the case of an interaction between any link coming from KALA and any link coming from SOM, an error should occur. This declaration can be put *e.g.* in a separate configuration class, in order to avoid modifying both KALA and SOM aspect definitions.

The code for this is shown in Fig. 17. It first creates a selector matching all links resulting from the assimilation of a KALA aspect (81). This is done by introspecting the `LANG` property of a link (82)[1]. The same is done for matching all links resulting from a SOM declaration (83). Finally, a composition rule is declared (84)-(85): it ensures that any interaction between a SOM link and a KALA link is interpreted as an *error*, hence stopping the weaving process with the given error message, in addition to the information related to the interaction:

```
[ERROR] forbidden interaction on BufferData.get():
     - SOM (buf.som, line 1)
     - KALA (data.kala, line 1).
-> SOM and KALA cannot affect the same objects
```

In this scenario, the programmer is therefore left with the alternative of modifying the cut of the aspects in order to ensure that they do not interact.

[1] Links can have arbitrary properties. A usage of these properties is precisely to tag links with information related to the DSAL program that generated them, such as language name and source code location. We have chosen not to present these features here for the sake of clarity and brevity of the assimilation code.

6.3 Benefits of the Infrastructure

This section highlights the main benefits of our infrastructure for domain-specific aspect languages:

- **Versatile weaving facilities** – The reflective core of Reflex offers generic facilities for both structural and behavioral aspects, with both expressive cuts and actions. The fact that all constituents of an aspect are first-class entities (objects) brings a lot of benefits [9], some of which have been used in the implementation of KALA: a KALA aspect definition is parameterized by both its cut and the parameters used for the explicit instantiation of the demarcator; the demarcator controlling an autostart method is dynamically set to the demarcator of the enclosing definition; and the resourceable aspect is programmatically deployed upon the first definition of a KALA aspect.
- **Extensible language definition** – The language facilities of MetaBorg combine very well with the versatility of Reflex, since it is possible to define concisely, declaratively, and modularly, both domain-specific aspect languages and extensions to them. In addition, although we have not illustrated this feature here, MetaBorg supports actual *embedding* of languages within a host language [3]. This means that for example KALA code can be embedded within Java code.
- **Composition support** – The composition facilities of Reflex represent a major motivation for using this platform to implement domain-specific aspect languages, because it ensures that DSALs can be used in conjunction with other aspect languages, domain-specific or not. The fact that Reflex automatically detects interactions, reports on them, and offers expressive means for their resolution is crucial for AOP in general, and multi-language AOP in particular.

It is true that our infrastructure is certainly not as efficient as agressively-optimized compilers like *abc* [1]. Nevertheless, recent performance evaluations of AOP systems [14] show that Reflex performs really well compared to other dynamic aspect systems, which makes it a reasonable platform from this point of view. Also, the conciness of language definitions and extensions makes it more suitable for rapid language prototyping and validation of ideas, because less burden is placed on the programmer than that of extending a full compiler infrastructure.

7. Discussion

7.1 Previous Work

The motivation for a versatile AOP kernel was first presented in [29], and the first account of Reflex as a kernel for multi-language AOP was reported in [30]. Although a first attempt at the language layer of the kernel was included, the only languages supported in Reflex were AspectJ and SOM [5]. SOM indeed features a very limited DSAL because it is only used to configure bindings of schedulers implemented in Java, and the implementation of AspectJ was not extensible [23].

Since then, a major shift has been taken with respect to the language layer by working on the integration of MetaBorg, as reported here. An extensible kernel language for Reflex, *i.e.* concrete syntax for the Reflex kernel API using SDF plus the corresponding assimilation in Stratego, was proposed in [28]. Here, we have pushed the experimentation on the kernel a step further by studying the support of a full-fledged domain-specific aspect language. KALA is an interesting DSAL because it has its own specific syntax and scoping rules and, if considering the extension we have presented, requires both behavioral and structural abilities of the kernel.

With respect to KALA, the original proof-of-concept implementation was based on source-code transformation and required a variety of tools to be combined to weave KALA and Java source into one executable. The implementation in Reflex integrates these tools, making the development process using KALA much easier. Considering the time taken, the implementation in Reflex was completed in approximately half a man-month, whereas the original implementation took almost two. In terms of the implementation, we can compare both the syntax definition and the semantics.

In the original implementation the concrete grammar was defined using a yacc-like parser generator, and required 130 lines of code. The SDF definition is 4 times more compact (32 lines, Fig. 10). Also, the original version suffered from context-sensitive parsing issues for the Java expressions embedded in KALA statements, which required the use of special escape characters. ReLAx does not have this drawback, due to to the many advantages of SDF for parsing "composite" languages [2]. With respect to semantics, the original source code transformation engine consists of approximately 1200 lines of code, not counting the Java parser used. The ReLAx implementation consists of 150 lines of Stratego rules, and 500 lines of Java code. It not only is half the size of the original implementation but also presents several modularity advantages. First, transformation rules are specified declaratively in Stratego, separately from the actual semantics implementation in Java (Sect. 5.2). Second, the Java implementation consists of only 25 lines of weaving specification (the Reflex configuration class of Fig. 9), and 4 classes implementing the whole of KALA as explained in Sect. 4.2. The demarcator class is the most involved; about 230 lines of code. In contrast to this, the original implementation is an ad-hoc engine handling weaving- and semantics-related concerns in a mixed manner, making it much less evolvable and understandable.

The current implementation furthermore has several advantages over the previous one. First of all, because Reflex is based on bytecode weaving, the source code of the base program is no longer required, potentially allowing a larger amount of software to be treated, and we are somewhat more protected from changes in the Java language grammar. Second, the current implementation is, as we have shown, modularly extensible, both in terms of syntax and semantics. This is important for future work on KALA. Finally, the fact that Reflex manages composition of aspects written in different languages is a definite plus compared to the previous version. Managing interactions such as between SOM and KALA would have been impossible with the previous version, because it "blindly" transforms base application code.

7.2 Related Work

Transaction management as an aspect. Although transaction management is generally accepted by the AOSD community as being an aspect, only a few papers have been published that treat this subject [19, 22, 25]. Furthermore, in two of these papers transaction management is but a minor topic because the main focus of the paper is on persistence [22, 25]. Common to the three papers is that they use AspectJ and aspectize classical transactions only and do not address issues like what happens if a transactional method calls another transactional method. KALA however addresses not only classical transactions, but also a wide variety of advanced transaction models. This allows the above issue to be easily addressed, *e.g.* using nested transactions it can straightforwardly be mapped to spawning a child transaction.

The first of the three above papers, by Kienzle and Gerraoui [19], is arguably the best known, using transaction management to evaluate AOP. The same topic is addressed by Soares et al. [25]: they build a number of persistence and distribution aspects, which include transaction management and implement a health watcher system first without and afterwards with these aspects. Similarly, Rashid and Chitchyan implement a persistence aspect, which includes a transaction management part [22]. The

paper explores whether persistence can be aspectized and if this aspect is reusable for other applications. Remarkably, the conclusions of the last two papers contradict the first: Kienzle and Gerraoui are pessimistic, while Soares et al. and Rashid and Chitchyan are optimistic about the use of AOSD for transactions.

Also treating the topics of aspects and transaction management is the recent work of Kienzle and Gélineau [18]. It is however not directly related to our work and the above three papers. This is because it discusses the design and implementation of a TP Monitor using aspects, raising interesting composition issues and not how (advanced) transactions are used by an application.

Extensible aspect languages. Work on extensible aspect languages somehow relates to ours. Josh [7] is an open AspectJ-like language, which makes it possible to experiment with new means of describing pointcuts and advices. However it does not fit the purpose of multi-language AOP mainly because of its lack of support for aspect composition, as well as the fact that it is not possible to experiment with DSALs whose syntax does not fit the AspectJ feel.

The AspectBench Compiler [1] is an extensible framework for experimenting with new language features in AspectJ. The spirit of *abc* is similar to Josh, but since *abc* is a full compiler, it provides a powerful framework for static analysis. By sticking to AspectJ as the basic language, *abc* presents the inconvenience that both the complexity of AspectJ and that of a full compiler infrastructure may be overkill for simple extensions. Also, issues in the extensibility of the syntax definition mechanism presently used in *abc* have been reported in [2]. The focus of *abc* is on efficient extensions to AspectJ, rather than on combination of different aspect languages and treatment of aspect composition.

Platforms for DSALs. The above projects do not explicitly target multi-language AOP as such, and are therefore only loosely related to our work. As a matter of fact, only a few proposals of platforms for domain-specific aspect languages have been made.

XAspects [24] is a plugin mechanism for domain-specific aspect languages, based on AspectJ [17]. An aspect language is implemented as a plugin generating AspectJ code, while the global compilation process is managed by the XAspects compiler. XAspects suffers a number of limitations, among which the most important are that the XAspects compiler provides no higher-level intermediate abstractions to DSAL implementors, and that composition of aspects is not tackled at all. Furthermore, because AspectJ is a mature and production-quality aspect language with a strong practitioner perspective, it has a limited versatility for the purposes of multi-language AOP: no detection of aspect interactions, restricted means for resolution, it is impossible to define algorithmic cuts, to parameterize aspects, pointcuts and advices, etc.

Brichau *et al.* [4] present an approach to building composable aspect-specific languages with logic metaprogramming. Using the same logic language, aspects and aspect languages can be composed. The underlying logic facilities are very good for expressing advanced and parameterized aspects, however, the aspect languages do not really shield the programmer from the inherent power of the logic metaprogramming approach: no aspect-specific syntax is provided, aspects are defined in the same logic framework as languages. Issues such as detection of interactions and support for structural aspects are not considered.

7.3 Future Work

Although research on advanced transactions has been dormant for some years, these concepts are becoming more and more relevant with the growth of Web Services and the requirements for composition of Web Services. We are therefore performing research on how KALA can be adapted to best fit this domain.

With respect to the support of domain-specific aspect languages in Reflex, we are in the process of fine-tuning a plugin architecture for the seamless integration of Reflex and MetaBorg, addressing issues such as packaging of language implementations, traceability, and dynamic combination of aspect languages. We are also designing and implementing a number of domain-specific and general-purpose aspect languages on top of this infrastructure.

8. Conclusions

In this paper we have shown how an infrastructure for domain-specific aspect languages (DSALs) aids in their development. In particular, we have detailed the implementation of KALA, a DSAL for advanced transaction management in the Reflex kernel for multi-language AOP. KALA is a good candidate for this case study because it has its own specific syntax and scoping rules and, in the extended version, requires both behavioral and structural abilities of the kernel in which it is implemented.

We first provided an operational description of KALA and gave an overview of its implementation over Reflex as a generic object parameterized by a collection of configuration objects. This illustrates an important benefit of using an infrastructure like Reflex, which is the use of generic facilities for aspects, where all constituents of an aspect are first-class objects. We then described how KALA programs are translated into the required configuration objects, giving the full KALA syntax definition and an overview of how code generation is performed, using SDF and Stratego. We have finally shown some additional benefits of using Reflex over writing a domain-specific weaver from scratch: *(a)* ease of making modular extensions to the language, and *(b)* built-in support for the automatic detection and explicit resolution of interactions between aspects written in different languages.

This brings us to the conclusion that the use of a appropriate infrastructure for domain-specific aspect languages, such as Reflex, gives a number of invaluable assets to the aspect language developer: ease of implementation, ease of extension and effortless support for composition. Using such an infrastructure enables a faster development cycle of DSALs and allows them to coexist within one application, thereby removing the most important impediments to the re-emergence of DSALs in the aspect community.

Availability. ReLAx is available on the Reflex website:
`http://reflex.dcc.uchile.cl/`

Acknowledgments

Thanks to Martin Bravenboer for his support with MetaBorg, and to Carlos Noguera, Nicolas Pessemier, and Guillaume Pothier for their invaluable feedback on drafts of this paper.

References

[1] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc: an extensible AspectJ compiler. In *Transactions on Aspect-Oriented Software Development*, volume 3880 of *Lecture Notes in Computer Science*, pages 293–334. Springer-Verlag, 2006.

[2] Martin Bravenboer, Éric Tanter, and Eelco Visser. Declarative, formal, and extensible syntax definition for AspectJ – a case for scannerless generalized-LR parsing. In *Proceedings of the 21th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2006)*, Portland, Oregon, USA, October 2006. ACM Press. ACM SIGPLAN Notices, 41(11). To Appear.

[3] Martin Bravenboer and Eelco Visser. Concrete syntax for objects. In *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*

(OOPSLA 2004), Vancouver, British Columbia, Canada, October 2004. ACM Press. ACM SIGPLAN Notices, 39(11).

[4] Johan Brichau, Kim Mens, and Kris De Volder. Building composable aspect-specific languages with logic metaprogramming. In Don Batory, Charles Consel, and Walid Taha, editors, *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2002)*, volume 2487 of *Lecture Notes in Computer Science*, pages 110–127, Pittsburgh, PA, USA, October 2002. Springer-Verlag.

[5] Denis Caromel, Luis Mateu, and Éric Tanter. Sequential object monitors. In Martin Odersky, editor, *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP 2004)*, number 3086 in Lecture Notes in Computer Science, pages 316–340, Oslo, Norway, June 2004. Springer-Verlag.

[6] Shigeru Chiba. Load-time structural reflection in Java. In Elisa Bertino, editor, *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP 2000)*, number 1850 in Lecture Notes in Computer Science, pages 313–336, Sophia Antipolis and Cannes, France, June 2000. Springer-Verlag.

[7] Shigeru Chiba and Kiyoshi Nakagawa. Josh: An open AspectJ-like language. In Karl Lieberherr, editor, *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pages 102–111, Lancaster, UK, March 2004. ACM Press.

[8] Panos K. Chrysanthis and Krithi Ramamritham. A formalism for extended transaction models. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 103–112, 1991.

[9] Bruno De Fraine, Wim Vanderperren, and Davy Suvée. Motivations for framework-based AOP. In *Proceedings of AOSD Workshop on Open and Dynamic Aspect Languages*, Bonn, Germany, 2006.

[10] Ahmed K. Elmagarmid, editor. *Database Transaction Models For Advanced Applications*. Morgan Kaufmann, 1992.

[11] Johan Fabry. *Modularizing Advanced Transaction Management - Tackling Tangled Aspect Code*. PhD thesis, Vrije Universiteit Brussel, Vakgroep Informatica, Laboratorium voor Programmeerkunde (PROG), July 2005.

[12] Johan Fabry and Theo D'Hondt. KALA: Kernel aspect language for advanced transactions. In *Proceedings of the 2006 ACM Symposium on Applied Computing Conference*, 2006.

[13] Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proceedings of the ACM SIGMOD Annual Conference on Management of data*, pages 249 – 259, 1987.

[14] Michael Haupt. *Virtual Machine Support for Aspect-Oriented Programming Languages*. PhD thesis, Software Technology Group, Darmstadt University of Technology, 2006.

[15] John Irwin, Jean-Marc Loingtier, John R. Gilbert, Gregor Kiczales, John Lamping, Anurag Mendhekar, and Tatiana Shpeisman. Aspect-oriented programming of sparse matrix code. In *ISCOPE*, volume 1343 of *Lecture Notes in Computer Science*, pages 249–256. Springer-Verlag, 1997.

[16] Shushil Jajodia and Larry Kershberg, editors. *Advanced Transaction Models and Architectures*. Kluwer, 1997.

[17] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of AspectJ. In Jorgen L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001)*, number 2072 in Lecture Notes in Computer Science, pages 327–353, Budapest, Hungary, June 2001. Springer-Verlag.

[18] Jörg Kienzle and Samuel Gélineau. Ao challenge - implementing the acid properties for transactional objects. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 202–213, New York, NY, USA, 2006. ACM Press.

[19] Jörg Kienzle and Rachid Guerraoui. AOP: Does it make sense? - the case of concurrency and failures. In Boris Magnusson, editor, *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002)*, number 2374 in Lecture Notes in Computer Science, pages 37–61, Málaga, Spain, June 2002. Springer-Verlag.

[20] Anurag Mendhekar, Gregor Kiczales, and Jim Lamping. RG: A case-study for aspect-oriented programming. Technical Report SPL97-009P9710044, Xerox PARC, February 1997.

[21] E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, Massachusetts Institute of Technology, 1981.

[22] Awais Rashid and Ruzanna Chitchyan. Persistence as an aspect. In *2nd International Conference on Aspect-Oriented Software Development*. ACM, 2003.

[23] Leonardo Rodríguez, Éric Tanter, and Jacques Noyé. Supporting dynamic crosscutting with partial behavioral reflection: a case study. In *Proceedings of the XXIV International Conference of the Chilean Computer Science Society (SCCC 2004)*, Arica, Chile, November 2004. IEEE Computer Society Press.

[24] Macneil Shonle, Karl Lieberherr, and Ankit Shah. XAspects: An extensible system for domain-specific aspect languages. In *OOPSLA 2003 Domain-Driven Development Track*, October 2003.

[25] Sergio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of the 17th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2002)*, pages 174–190, Seattle, Washington, USA, November 2002. ACM Press. ACM SIGPLAN Notices, 37(11).

[26] Éric Tanter. Aspects of composition in the Reflex AOP kernel. In Welf Löwe and Mario Südholt, editors, *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.

[27] Éric Tanter. Declarative composition of structural aspects. Technical Report TR/DCC-2006-11, University of Chile, 2006. Submitted to TAOSD.

[28] Éric Tanter. An extensible kernel language for AOP. In *Proceedings of AOSD Workshop on Open and Dynamic Aspect Languages*, Bonn, Germany, 2006.

[29] Éric Tanter and Jacques Noyé. Motivation and requirements for a versatile AOP kernel. In *1st European Interactive Workshop on Aspects in Software (EIWAS 2004)*, Berlin, Germany, September 2004.

[30] Éric Tanter and Jacques Noyé. A versatile kernel for multi-language AOP. In Robert Glück and Mike Lowry, editors, *Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005)*, volume 3676 of *Lecture Notes in Computer Science*, pages 173–188, Tallinn, Estonia, September/October 2005. Springer-Verlag.

[31] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In Ron Crocker and Guy L. Steele, Jr., editors, *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2003)*, pages 27–46, Anaheim, CA, USA, October 2003. ACM Press. ACM SIGPLAN Notices, 38(11).

[32] Mark van den Brand, Hayco de Jong, Paul Klint, and Pieter Olivier. Efficient annotated terms. *Software–Practice and Experience*, 30:259–291, 2000.

[33] Eelco Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997.

[34] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Springer-Verlag, June 2004.