ReLAx: Implementing KALA over the Reflex AOP Kernel

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. This raises the necessity of an appropriate infrastructure for DSALs. This paper illustrates how the Reflex kernel for multi-language AOP allows for the definition of DSALs, by considering the implementation of a DSAL for advanced transaction management, KALA. We detail the implementation of KALA in Reflex, illustrating the ease of implementation of runtime semantics, syntax, and language translation.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques

Keywords Domain-Specific Aspect Languages, Reflex, KALA.

1. Introduction

Initial research on AOP focused on domain-specific aspect languages (DSALs), like RG [14] and AML [10]. 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 [12]. Important reasons for this trend are the following three impediments to the growth of DSALs: the amount of effort required to implement a new language; the difficulty to extend such an ad-hoc weaver; the inability to combine the weavers of different DSALs.

The above three impediments can however be addressed through the use of an appropriate infrastructure for DSALs. This is precisely the objective of Reflex, a kernel for multi-language AOP [21]. 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 [2].

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

Workshop DSAL '07 March 12, 2007 Vancouver, British Columbia, Canada.
Copyright © 2007 ACM 978-1-59593-659-8...$5.00

This paper shows the design and implementation of a non-trivial DSAL on top of Reflex. The language is KALA, a DSAL for advanced transaction management [6, 7], which is significant in size, complexity, and specific syntax and scoping rules. The comprehensive discussion of the implementation provided here furthermore demonstrates how the use of such infrastructure allows to create a compact yet accessible implementation.

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 concludes.

2. Multi-language AOP and Reflex

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 [20] as well as our current Java implementation, Reflex [21].

A versatile AOP kernel supports the core semantics of various AO languages through appropriate 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, such a kernel is a mediator between different coexisting AO approaches: it detects interactions between aspects, possibly written in different languages, and provides expressive means for their resolution.

The architecture of an AOP kernel hence consists of three layers: 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; and 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 [9]. 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 a versatile kernel for multi-language AOP. As such, it follows the architecture of an AOP kernel [20, 21]:

- 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 [22], and explained in more detail next.
- The **composition layer** ensures automatic detection of aspect interactions, and provides expressive means for their explicit resolution [18, 19].
- The **language layer** is based on the MetaBorg approach for unrestricted embedding and assimilation of domain-specific languages [2].

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, e.g., the control at which metaobjects act (before, after, around), and a dynamically-evaluated activation condition. The aforementioned links are called *behavioral links* to distinguish them from *structural links*, which are used to perform structural actions [19].

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 [3]. 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.

3. KALA in a Nutshell

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 (ATMS) [5, 11]. An overview of these advanced models is outside of the scope of this paper. The most well-known advanced models are *nested transactions* [15], which allow a hierarchically structured computation to be matched to a tree of transactions, and *sagas* [8], to split a long-lived transaction into a number of shorter steps. We only discuss nested transactions here.

Example: Nested transactions. This model is one of the oldest and arguably the best-known ATMS [15]. It enables a running transaction T to have a number of *child transactions* T_c. 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.

The ACTA Formalism for ATMS. In addition to a large number 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 [4]. 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 [13, 16, 17]. 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.

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 [7]; in this paper we solely provide a brief description of the language using one example program fragment for nested transactions.

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 usual in OO programs that use of transactions, the life-cycle of every transaction coincides with the life-cycle of a method [6]. The transaction begins when the method begins, commits when the method ends normally, and aborts if the method ends with a specific 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 [12].

Consider the KALA code shown in Fig. 1. Line (1) is the *KALA signature*, which identifies the transactional methods. As a result, all data accesses to shared data within these methods (and within methods called by these methods) 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.

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.

```
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 1. KALA code for children in nested transactions.

Transactional properties. Transactional properties of a method are declared to apply either at begin, commit or abort time. They 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 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` (5), 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. 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 [4, 7]. 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. In Fig. 1, 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 there.

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 do not discuss in details here. Such *secondary transactions* run outside of the main control flow of the application, and do not need to be run in order to have a successful completion of the advanced 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.

4. ReLAx: Implementing KALA in Reflex

4.1 Operational Description of KALA

In general, 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* [6]. 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 [6]).

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. 2, 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.

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. After dependencies have been set, the autostarts are allowed to proceed with their begin phase.

At this point, ATPMos is asked if, according to the dependencies currently placed on this transaction, it may begin; otherwise, this call blocks. The call to ATPMos may finally return with three possible values (Fig. 2): 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. 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.

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 set 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 due to a dependency, control flow does

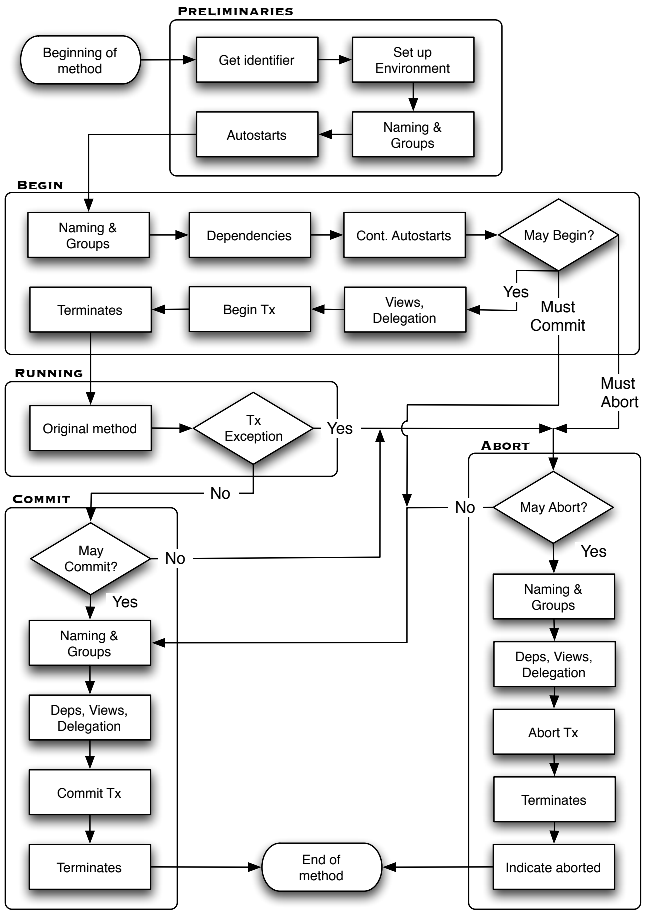

Figure 2. Flow chart of a KALA transaction.

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 [6]. 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. For sake of simplicity, we consider a KALA program that only contains one KALA declaration. This discussion can however straightforwardly be generalized to programs with multiple KALA declarations.

Declaration links. 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 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.

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. 2, in interaction 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 8 configuration objects, as well as with the list of formal parameters of the transactional method. A `Demarcator` is reentrant, *i.e.* 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 of a 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: a naming evaluator and autostarts for the preliminaries, and for begin, commit and abort blocks a naming evaluator and a transactional properties object. A Reflex link is created using the parameterized demarcator as metaobject, and with the appropriate attributes.

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 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 (a process called assimilation).

```
module Kala imports Java-15-Prefixed Pattern                        12
 exports context-free syntax
  KDecl* ->  CompilationUnit                                         13
  FQMPattern KBody              -> KDecl                             14
  "{" Prelim? BeginBlock?
     CommitBlock? AbortBlock? "}" -> KBody                           15
  PrelimStm*                   -> Prelim                             16
  "begin"  "{" BlockStm* "}" -> BeginBlock                           17
  "commit" "{" BlockStm* "}" -> CommitBlock                          18
  "abort"  "{" BlockStm* "}" -> AbortBlock                           19
  AStartStm     -> PrelimStm                                         20
  NamingStm     -> PrelimStm                                         21
  NamingStm     -> BlockStm                                          22
  DepStm        -> BlockStm                                          23
  ViewStm       -> BlockStm                                          24
  DelStm        -> BlockStm                                          25
  TermStm       -> BlockStm                                          26
  "autostart" "(" MethSig ASActuals
                         KBody ")" ";" -> AStartStm                  27
  "alias" "(" KBinding ")" ";"          -> NamingStm                 28
  "name"  "(" KBinding ")" ";"          -> NamingStm                 29
  "groupAdd" "(" KBinding ")" ";"       -> NamingStm                 30
  "dep" "(" JavaId JavaId JavaId ")" ";"-> DepStm                    31
  "view" "(" Min? JavaId JavaId ")" ";" -> ViewStm                   32
  "del" "(" JavaId JavaId  ")" ";"      -> DelStm                    33
  "terminate" "(" JavaExpr  ")" ";"     -> TermStm                   34
  JavaId JavaExpr                       -> KBinding                  35
```

Figure 3. Syntax definition of KALA in SDF.

5.1 Declarative Syntax Definition

The syntax definition of KALA is performed using SDF, a modular syntax definition formalism [24]. Fig. 3 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 [1]), shown in line (12). SDF productions are declared in the reverse manner from the traditional BNF notations, as illustrated in Fig. 3 where (13) 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 (14); a `KBody` is made up of 4 optional sections (15): preliminaries, begin block, commit block, and abort block. Preliminaries are a list of `PrelimStm` (16), while blocks are made up of `BlockStm` (17)-(19). A `PrelimStm` can be either an autostart (20) (defined on line (27)), or a naming statement (21). A naming statement is also valid as a `BlockStm` (22), along with dependency, view, delegation, and termination statements (23)-(26). A naming statement can either be an `alias` (28), a `name` (29), or a `groupAdd` (30). These statements include binding expressions, binding a Java identifier to an expression (35). Dependency, view, delegation, and termination statements also make use of the imported Java non-terminals `JavaId` and `JavaExpr` (31)-(34).

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 [23]. The actual AST nodes that are produced for

```
AssimKDecl :                                               36
KDecl(meth, KBody(prelim, begin, commit, abort)) ->
 |[ Hookset    ~hs      = ~<AssimMethSig> meth ;           37
    String[]   ~formals = ~<AssimFormals> meth;            38
    NamingEval ~pre-nts = ~<AssimNTs> prelim;              39
    AStart[]   ~as      = ~<AssimAStarts> prelim;          40
    NamingEval ~b-nts   = ~<AssimNTs> begin;               41
    KProps     ~b-props = ~<AssimProps> begin;             42
     // ...same for commit and abort blocks...
    this.install(~hs, ~formals, ~pre-nts, ~b-nts,          43
     ~b-props, ~c-nts, ~c-props, ~a-nts, ~a-props); ]|
 where <newname> "hs" => hs                                44
  ;  // etc. for all variable names
```

Figure 4. Rule for assimilating a KALA declaration.

```
AssimProps :
 stms ->
  |[ new KProps(new String[][]{~deps },             45
               new String[][]{~views},
               new String[][]{~dels }) ]|
  where <try(filter(AssimDep))>  stms => deps       46
      ; <try(filter(AssimView))> stms => views      47
      ; <try(filter(AssimDel))>  stms => dels       48

AssimDep :                                           49
 DepStm(Id(src), Id(dep), Id(dest)) ->
   var-init |[ { "~src", "~dep", "~dest" } ]|        50
 // similar for views and delegation
```

Figure 5. Rule for assimilating transactional properties.

```
AssimNTs :
stms ->
 |[ new NamingEval(){                                       51
     void evalNaming(KALAEnv e,TxManager t){~n_stms}        52
     void evalTerm(KALAEnv e,TxManager t){~t_stms}} ]|      53
  where <try(filter(AssimNaming))> stms => n_stms           54
      ; <try(filter(AssimTerm))> stms   => t_stms           55
AssimNaming :
 NameStm(KBinding(Id(id), expr)) ->
   |[ this.nameOp(e, t, "~id", ~expr_ok); ]|                56
 where <topdown(try(LookupId))> expr => expr_ok             57
 // similar for alias and groupAdd statements
AssimTerm :
 TermStm(Term(expr)) ->                                     58
   |[ this.terminateOp(e, t, ~expr_ok); ]|
 where <topdown(try(LookupId))> expr => expr_ok
LookupId :
 ExprName(Id(id)) -> |[ e.lookup("~id") ]|                  59
```

Figure 6. Rules for assimilating naming and termination.

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 [25]. By defining assimilation rules, KALA declarations are converted into Reflex configuration code, in plain Java.

Assimilating declarations. Fig. 4 shows the main assimilation rule, which deals with KALA declarations. An assimilation rule has a name (36), 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 [2]. Within this block, *metavariables* are referred to using the `~` escape. A `where` clause (44) 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. An `install` method is then called, creating and installing the link (43). These statements are inserted in the initialization method of a generated configuration class.

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. 4 are denoted by an identifier in order to refer to them when calling the `install` method (43). To ensure hygiene, identifiers are automatically generated by Stratego. This is why the hookset variable in (37) is the metavariable `~hs`, which is determined in the `where` clause of the rule, which applies the `<newname>` utility rule to the `"hs"` symbol (44). 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 (37) 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 applying a rule to this same term (38). Following this, the eight configuration objects (Sect. 4.2) needed are obtained via application of dedicated rules: the preliminary naming statements (39), the autostart objects (40), the naming and termination statements of the begin block (41), its transactional properties (42), etc.

Assimilating properties. Statements that deal with transactional properties *–i.e.* dependencies, views and delegation– are assimilated into a configuration object `KProps` (Fig. 5). A `KProps` object is a bi-dimensional array of strings, as explained in Sect. 4.2. The creation of this array is shown in (45); the content of each column in this configuration object is obtained via applying other assimilation rules, one for each type of property: dependencies (46), views (47), and delegation (48). 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. 5 shows the assimilation rule for dependencies (49): if a statement is a dependency, it is assimilated into a variable initializer with the three corresponding values (source, dependency, and destination) (50).

Assimilating naming and termination. Naming and termination are more complex statements to assimilate (Fig. 6), 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) (51). A naming evaluator has two methods, `evalNaming` and `evalTerm`, which are filled in with statements generated by the assimilation of naming (52) and termination (53), respectively. The application of these assimilations is defined in the `where` clause of the main assimilation rule (54)(55).

As an example, Fig. 6 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 (56). 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 (57). 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 (59). The assimilation of a termination statement is very similar (58).

6. Conclusions

In this paper we have detailed the implementation of KALA, a DSAL for advanced transaction management in the Reflex kernel for multi-language AOP. 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. 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.

Contrast this with the original proof-of-concept implementation of KALA, based on source-code transformation. 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. 3). 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. This illustrates the advantages we gained with respect to amount of code that needs to be written and maintained.

Furthermore this paper demonstrates that, while being compact, the implementation is accessible enough that it can be explained in reasonable detail in the scope of a few pages. This is thanks to the use of an appropriate infrastructure, in this case Reflex.

References

[1] 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.

[2] 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).

[3] 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.

[4] 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.

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

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

[7] 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.

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

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

[10] 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.

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

[12] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of AspectJ. *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.

[13] Jörg Kienzle and Rachid Guerraoui. AOP: Does it make sense? - the case of concurrency and failures. *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.

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

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

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

[17] 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).

[18] 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.

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

[20] 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, Sept 2004.

[21] 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.

[22] 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).

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

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

[25] 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.