Available online at www.sciencedirect.com

ScienceDirect

Science of Computer Programming 71 (2008) 165–180

Science of Computer Programming

www.elsevier.com/locate/scico

KALA: Kernel aspect language for advanced transactions

Johan Fabry[a,*], Éric Tanter[a], Theo D'Hondt[b]

[a] *PLEIAD Lab, Computer Science Department (DCC), University of Chile, Blanco Encalada 2120, Santiago, Chile*
[b] *Vrije Universiteit Brussel, Programming Technology Lab, Pleinlaan 2, 1050 Brussel, Belgium*

Received 31 July 2006; received in revised form 29 April 2007; accepted 24 October 2007
Available online 31 January 2008

Abstract

Transaction management is a known crosscutting concern. Previous research has been conducted to express this concern as an aspect. However, such work has used general-purpose aspect languages which lack a formal foundation, and most importantly are unable to express *advanced* models of transaction management. In this paper, we propose a domain-specific aspect language for advanced transaction management, called KALA, that overcomes these limitations. First, KALA is based on a recognized formalism for the domain of advanced transaction management, called ACTA. Second, as a consequence of being based on the ACTA formalism, KALA covers a wide variety of models for transaction management. Finally, being a domain-specific aspect language, KALA allows programmers to express their needs at a higher level of abstraction than what is achieved with general-purpose aspect languages. This paper reports on the design of KALA and its implementation over Java, based on the Reflex AOP kernel for domain-specific aspect languages.

Keywords: Advanced transaction management; Domain-specific languages; ACTA; Reflex

1. Introduction

The de facto standard for managing concurrency in large-scale distributed systems are *transactions*. These systems, however, are confronted with the limitations of transaction management, because transactions were originally designed to manage only one specific kind of data access: short, unstructured accesses, working in isolation, that only touch a few data items. A large body of work in the transaction management community recognizes this problem and many *advanced transaction models* (ATMS, sometimes also called extended transaction models) have been developed [1,2] that address these limitations. A prime example, and arguably the best-known ATMS, is nested transactions [3]. The research in this area also includes a general formalism, *ACTA* [4,1], which allows a wide variety of ATMS to be described. The list of ATMS is open-ended as each ATMS addresses a particular subset of the limitations of transaction management (which, hereafter we refer to as classical transactions), and no single ATMS has been created which addresses all the known shortcomings of transaction management systems.

The use of such an ATMS in an application, however, poses a significant hurdle for the programmer, because code must be added to the application to mark the start and the end of transactions, as well as for indicating which data

* Corresponding author.
E-mail addresses: jfabry@dcc.uchile.cl (J. Fabry), etanter@dcc.uchile.cl (É. Tanter), tjdhondt@vub.ac.be (T. D'Hondt).

0167-6423/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2007.10.004

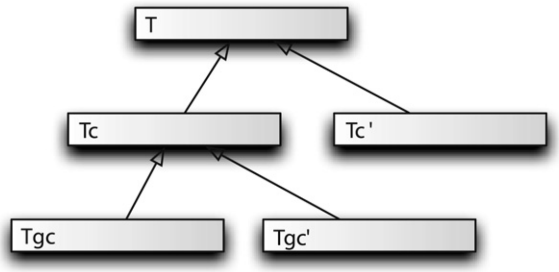

Fig. 1. The nested transactions ATMS.

accesses are part of what transaction. A known problem of this *demarcation code* is that it cannot be modularized through classical software engineering techniques, because it *crosscuts* the application structure [5–7]. As a result, the code for the transaction management concern is scattered throughout the application, where it is tangled with the core logic of the application. To address these problems, research has been performed which renders classical transactions into an aspect [5–7]. These proposals allow the programmer to define the transactional behavior of an application separately from the application logic, yielding the known benefits of separation of concerns.

However, there are two main issues with current proposals of aspectizing transactions: (i) they are limited to classical transactions; (ii) they do not provide any formal foundation, which would allow them to be extended to address ATMS in general. Our proposal overcomes these limitations by building upon the formal foundation of ACTA [4,1]. We transform this formalism into an aspect language that covers a wide variety of ATMS, including classical transactions. This allows an application programmer to modularize *advanced* transaction management into an aspect.

The aspect language we propose is called KALA, which stands for **K**ernel **A**spect **L**anguage for **A**TMS. KALA is a Domain-Specific Language [8,9] (DSL) created specifically for the domain of ATMS as an aspect, i.e., KALA is a *Domain-Specific Aspect Language* (DSAL). As KALA is specific to a particular domain, KALA programs are written at a much higher level of abstraction than the corresponding demarcation code. Even compared to the research on classical transactions as an aspect, KALA offers a higher level of abstraction, because these proposals use general-purpose aspect languages. Therefore, the use of KALA results in specifications which are not only easier to grasp, but also are more concise. This drastically lowers the amount of code that needs to be written.

In this paper we detail the creation process of KALA. We first give a brief overview of the ACTA formalism, before considering what was needed to turn this formalism into an aspect language. We then introduce the KALA language, highlighting its most important features and providing an overview of its syntax. Next, we overview the implementation of KALA over the Reflex kernel for domain-specific aspect languages. Finally, we discuss the advantages of using KALA and conclude.

2. The ACTA formalism

In contrast to the earlier work on aspect languages for transaction management, which are not based on a formal model, we have chosen to base our aspect language on a generally accepted formalism for the domain of ATMS. This not only yields a strong formal foundation for our aspect language, but also ensures wide applicability of the language.

The formalism we have chosen to base our aspect language on is *ACTA* [4,1], which is a well-known and accepted formalism for ATMS. Furthermore, ACTA descriptions are purely concerned with the transactional behavior of the system, and do not include any information whatsoever on the base concern of the application. Therefore, ACTA is a good base for the construction of an aspect language for ATMS, as this aspect language will only treat the transactional behavior of the system [10].

We now give a brief overview of ACTA, focusing on the most important elements of the formalism and using the nested transactions ATMS as a running example. In Nested Transactions, illustrated in Fig. 1, a tree of transactions is formed where children Tc have access to the intermediate results of the parent T. Also, the work of children is not written to the database as they commit, but becomes part of the work of the parent. This implies that a child has to commit before its parent commits and that if a parent aborts, that its children also abort. Lastly, if a child aborts, its parent is not required to abort.

ACTA formal definitions are based on the *transaction history*: the sequence of operations performed by the transactions. These operations are either reads and writes of shared data or the life-cycle operations of the transactions themselves: *begin*, *commit* and *abort*. In ACTA these life-cycle operations are called *significant events*. Each ATMS may define additional significant events which are added to this list. Nested transactions, for example, adds the *spawn* significant event, indicating the start of a nested transaction.

In ACTA, an ATMS is formally defined by stating three kinds of axioms that constrain and modify the transaction history: dependencies, views and delegation.

The first kind of axiom on transaction histories is *dependency*: a dependency places a relationship between two transactions, defined in terms of the significant events of these transactions. Dependencies are restrictions placed on the execution of certain transactions, depending on the result of execution of other transactions. A first example dependency is the commit dependency ($Tj\ \mathcal{CD}\ Ti$): if transactions Ti and Tj commit, Ti must commit before Tj. In nested transactions, if Tp is a parent transaction and Tc a child transaction, then ($Tp\ \mathcal{CD}\ Tc$), i.e., the child commits before the parent commits. A second example is the weak-abort dependency ($Tj\ \mathcal{WD}\ Ti$): If Ti aborts and Tj has not yet committed, then Tj must abort. For nested transactions, this is expressed as ($Tc\ \mathcal{WD}\ Tp$). The dependencies we will use in this paper are the following:

commit dependency ($Tj\ \mathcal{CD}\ Ti$): if transactions Ti and Tj commit, Ti must commit before Tj.
weak-abort dependency ($Tj\ \mathcal{WD}\ Ti$): if Ti aborts and Tj has not yet committed, then Tj must abort.
abort dependency ($Tj\ \mathcal{AD}\ Ti$): if Ti aborts then Tj must also abort.
begin-on-commit dependency ($Tj\ \mathcal{BCD}\ Ti$): Tj is only allowed to begin executing after Ti has committed.
begin-on-abort dependency ($Tj\ \mathcal{BAD}\ Ti$): Tj is only allowed to begin executing until Ti has aborted.
compensation dependency ($Tj\ \mathcal{CMD}\ Ti$): Tj is required to commit if Ti aborts.

The second kind of axiom defined by ACTA is *view*: a view definition allows one transaction to view the intermediate results of another transaction. The use of views between two transactions remove the isolation barrier between these two transactions, letting them concurrently work on the same data as if they were the same transaction. For example, in nested transactions, the child transaction Tc sees the intermediate results of the parent Tp. This is specified by defining the view of Tc to contain Tp.

The third kind of axiom is *delegation*: one transaction Ti delegates the responsibility for a specified number of its operations to another transaction Tj. Conceptually, the transaction history is modified such that the operations which were performed by Ti are now recorded as being performed by Tj. For example, in nested transactions, if a child transaction Tc commits, its effects are delegated to its parent Tp.

A wide variety of ATMS have been formally described in ACTA [4,1], including classical transactions. This indicates that ACTA is suitable to use as a foundation to build our aspect language for ATMS.

3. From a formalism to an aspect language

As ACTA allows the formal specification of a wide variety of ATMS, it is beneficial to use it as a basis for the implementation of an aspect language specific for the domain of ATMS. This domain-specific aspect language can, in turn, support a wide variety of ATMS.

ACTA, however, is just that: a formalism. It was not conceived with an implementation in mind. This is evident from the fundamental premise of ACTA: *"The correctness properties of different transaction models can be expressed in terms of the properties of the histories produced by these models"* [4]. In other words, ACTA verifies *a posteriori* if the produced transaction history is correct. For a running application, however, verification of transaction histories, i.e., adherence to the ATMS, must be performed *while the application is running*.

In classical transaction management correctness is provided by a *transaction processing monitor* [11] (usually abbreviated as TP Monitor). The TP Monitor is a component of the distributed system, built specifically for this purpose and its working is steered by the transaction demarcation code present in the application. To support the use of ATMS we have created a TP Monitor, called ATPMos, that provides an implementation of the three kinds of axioms present in ACTA: dependencies, views and delegation, in addition to the tasks performed by a TP Monitor for classical transactions. We will briefly touch on the implementation of ATPMos later, in Section 5. For now, suffice it to say that ATPMos provides for runtime verification and enforcement of the ACTA axioms for a given application, provided that this application includes the corresponding transaction demarcation code.

Returning to the *a-posteriori* verification of transaction histories, as defined in ACTA, this implies that the record of this history is kept for ever, leading to a potentially infinitely large transaction history. It is clear that an implementation of a TP Monitor based on ACTA needs to restrict memory usage to a reasonable amount. Therefore, it is best to keep as little information about the transaction history as possible and remove it from the system when no longer needed. A common technique to enforce correct transaction histories for classical transactions, without needing to record them, is the use of locks [11]. Using locks allows for verification while the transactions are running, in contrast to the *a posteriori* approach of ACTA. If we wish to use locks in a TP Monitor based on ACTA, the demarcation code needs to perform additional work, while the transaction is running, to ensure that the resulting history is correct. There are four main topics to consider in this work: performance; dependency checking; transaction life-cycle; and naming. We outline these next.

3.1. Performance

The first implementation topic to consider is performance: the expressive power of ACTA is a possible performance hurdle for the implementation. There are two areas which need to be addressed here. We have already indicated the need to keep the transaction history as minimal as possible. Second is the issue of limiting the expressiveness of ACTA: In a number of places in the axioms an arbitrary logic expression can be used as a specification, which can take an arbitrarily large amount of time to evaluate. A complete implementation of ACTA, therefore, requires the TP Monitor to be able to not only reflect on the transaction history, but also to evaluate logic expressions given by the ATMS specification. Not only is this a significant implementation task, but it would also have, in general, an undefinably large impact on transaction performance. Such a performance impact runs contrary to the expectations for TP Monitors, which is to process transactions in a short time-frame. It is therefore necessary to consider limiting the expressiveness of the framework somewhat in favor of a faster TP Monitor. Consequently, we take a representative subset of ACTA and consider expanding this as required later.

Reviewing the ATMS specifications published in [4,1], we can consider the following simplifications to limit the transaction history kept by the TP Monitor:

Locks: As we have stated above, enforcement of correct transaction histories is already performed in TP Monitors through the use of locks. Therefore, we do not need access to the transaction histories to determine conflicts.

Delegation: Modifying the transaction history is only performed simultaneously with a significant event, and we have established that most frequently all the work of a transaction is delegated to another transaction. Combined with the use of locks, as above, where owning a lock implies responsibility of an operation, this means that delegation boils down to changing the ownership of all the locks of the delegator to the delegatee.

The two above simplifications mean that we can remove the explicit transaction history, while still guaranteeing that a significant number of ATMS are covered. Furthermore, investigation of the ATMS published in [4,1] has shown that no use is made of full logic expressions. Therefore, we have opted to remove the expression evaluation. To assess, in general, the full impact of such removal is a hard task, as there are an unlimited number of ATMS which can be specified in ACTA. If we would need to achieve full compliance to the formalism, we can however add such support to the TP Monitor later.

3.2. Dependency checking

The second topic to consider are dependencies, which also will indirectly impact performance. As an example, consider $Tj\ \mathcal{CD}\ Ti$: If Ti and Tj commit, Ti must commit before Tj. This is very easy to verify *a posteriori*, but this verification is more intricate if performed at runtime. While running, the transactional aspect of the application will build a dynamically evolving web of related dependencies, depending on the execution of the application. Owing to this dynamic nature, we cannot statically determine if dependencies are satisfied. Instead, we need to ensure that we do not violate these dependencies while the application is executing. To achieve this, each transaction will have to wait before a significant event until its dependencies are satisfied. Furthermore, it may be forced to commit or abort at that time in order to satisfy a dependency.

For example, taking $Tj\ \mathcal{CD}\ Ti$ as above, if Tj wants to commit while Ti is running, this has to be postponed until Ti has committed. Also, if Ti aborts, postponing Tj will turn out to be unnecessary. This is because the abort of Ti means that there is no commit restriction on Tj. As this is runtime behavior of the system it is impossible to

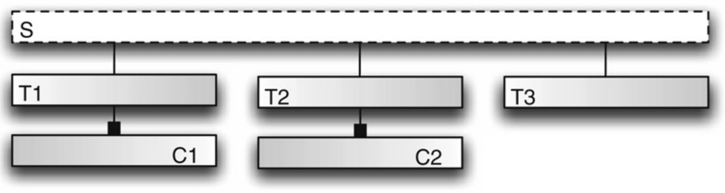

Fig. 2. The Sagas ATMS.

determine *a priori* the behavior of Ti with regard to commits or aborts, i.e., whether Tj needs to wait before a commit or not. As a result, Tj is always required to wait for Ti at commit time in the case of $Tj\ \mathcal{CD}\ Ti$. This entails that the performance of the TP Monitor will not be optimal in all cases. Instead, we trade off performance for guaranteed correctness in all possible cases.

As said above, dependencies are specified with a link to a significant event of a transaction: begin, commit or abort. Recall that the list of significant events is open and that each ATMS may add its own significant events to the list consisting of begin, commit and abort. Such an open list of significant events requires a certain flexibility of the dependency mechanism, as it may need to be invoked at any point of the execution of a transaction. It is clear that having a fixed set of significant events eases implementation of the dependency mechanism, which will probably also open avenues for performance optimization. Investigating the published ATMS in [4,1], we see that this is indeed possible. In these ATMS, we can map new significant events to begin, commit or abort. For example, consider the spawn event in nested transactions. This event can be mapped to the begin event, as spawn denotes the beginning of a nested transaction. Therefore, we limit the significant events to begin, commit, and abort, and implement the dependency checking mechanism such that it only intervenes at these moments in the execution of a transaction.

3.3. Transaction life-cycle

Third, we need to concern ourselves with the underlying forces that steer the life-cycle of a transaction: Transactions are started and ended by the underlying application as it executes. However, ACTA does not concern itself with the application's execution, but solely with the resulting history. Therefore, to ensure that the resulting history is correct, we need to consider how we can make certain that transactions start and end when necessary. It is clear that the main part of the responsibility for this lies with the control flow of the underlying application: at some points a transactional unit of code will run, resulting in the beginning of a transaction, and when the unit ends, the transaction will also end. We have chosen to use methods as transactional units of code, an approach which is common in the field. What this means is that a transaction begins when method execution begins, and the transaction ends as method execution ends.

A major assumption that we need to make is that the control flow of the methods does not run contrary to the control flow mandated by the ATMS. We need to make this assumption because, in general, this is impossible to verify statically or fully enforce dynamically. For example, in nested transactions, the method of a sub-transaction has to end before the method of its parent has ended. If these methods run in different threads, and the child transaction depends on user input to terminate, we cannot, at compile-time, verify termination of the child prior to the parent. While this example can be enforced dynamically others cannot. For example, consider an ATMS that requires some transaction Ti to run before another transaction Tj. If in a single threaded program the method of Tj is called before Ti, we cannot, at runtime, ensure that Tj runs before Ti, as this would imply that we modify the order of the statements in the code.

Secondary transactions

In addition to transactions running in the main control flow of the application, some ATMS require certain *secondary* transactions to run. These are conceptually separate from the main control flow of the application. Instead, they run automatically when their dependencies are satisfied. Usually secondary transactions are present to ensure the overall consistency of the shared data. For example, consider the Sagas ATMS [12], one of the best-known ATMS in the domain, illustrated in Fig. 2. In Sagas, a transaction S is split into a sequence of steps $T1$ to Tn, where each step is a classical transaction. Also, for each step Ti where $i : 1, \ldots, n-1$, a semantical 'undo' action Ci, called a *compensating transaction*, is defined. These compensating transactions are secondary transactions that are used to rollback the Saga when this is required. To perform such a rollback, the currently running step is rolled back and the compensating transaction for each step which has already committed is fired, in the inverse order of the steps. This illustrates that such secondary transactions are not related to the application logic *per se* but are a result of using the ATMS. Therefore, we should not require the application code to verify these dependencies and run the appropriate code, but place this extra responsibility at the conceptual level of the ATMS, and not of the application.

Transaction termination

The end of a transaction Ti does not immediately imply that the TP Monitor may remove all traces of this transaction from memory. There may still be dependencies placed on Ti that constrain other transactions still running in the system. For example, consider a dependency that forces a transaction Tj to abort if Ti has committed. When Ti commits, this dependency may not be removed until Tj has also ended. Owing to the dynamic nature of dependencies, automatically deciding when dependencies can be removed without impacting the consistency of the ACTA specification is impossible. As a result, we need the programmer of the application to write demarcation code that explicitly *terminates* the transaction, notifying the TP Monitor that the transaction can safely be removed completely from the system.

3.4. Naming

The fourth and last topic to consider is the naming of transactions: while formally representing transactions as Ti and Tj, such identifications are not straightforwardly translatable to a transaction at runtime. What we need is to be able to identify unambiguously a transactional unit of code at runtime. Hence, we need some form of naming support.

Our solution for this is inspired by the concept of name servers in distributed systems: we have added a name service to the TP Monitor. Transactions can register themselves in this name service under a name of their own choosing. Other transactions can obtain a reference to a registered transaction by querying the name service for the transaction with that name.

Also, as we are creating an aspect language, we need to consider how crosscuts are specified. This will allow the aspect, i.e., a collection of ACTA properties for a transaction, to be associated with the method for that transaction.

This concludes our discussion of the issues we need to take into consideration when using the ACTA formal model as the basis for an aspect language for ATMS. We first talked about the required runtime component that enforces the constraints of the model: the TP Monitor. We then touched upon four specific topics that require special attention: the impact on performance, performing dependency checking, the life-cycle of transactions, and naming and grouping. Having considered these topics, we can now continue with introducing the aspect language for ATMS which we have defined: KALA.

4. Introduction to KALA

The KALA aspect language was specifically designed to allow for the separate specification of transactional properties for ATMS of Java methods. KALA is a domain-specific aspect language based on the ACTA formalism, thereby ensuring that a wide variety of ATMS can be implemented. In KALA, the programmer declaratively states the transactional properties of a Java method in a block of statements, using the constructs provided by the ACTA formalism. In other words, in KALA, dependencies, views, and delegations of a given transaction are defined through declarations made for the corresponding method.

This section informally discusses the KALA language, without referring to its actual implementation. This topic is deferred to the next section.

4.1. Naming

To be able to refer to transactions within a KALA program, two naming schemes are required: a scheme for load-time naming and a scheme for runtime naming and grouping.

Load-time naming

Recall that in Section 3.3, we have chosen to let the transactional units of code be methods. Consequently, to identify to which transactional unit of code a set of transactional properties belongs, we need to uniquely identify methods. The identification phase is performed at load time, i.e., when Java class files are loaded into the virtual machine. Hence this naming is termed load-time naming.

Load-time naming is performed in KALA by giving the full class name and the method signature, separated by a dot. This allows us to write our first KALA program, below, containing one KALA declaration. This declaration states that the method with signature `root(int foo)` of the class `util.strategy.Hierarchical` is transactional.

```
util.strategy.Hierarchical.root(int foo) {}
```

The block after the signature, known as the body of the KALA declaration, is empty. This means that the declaration does not specify any additional transactional properties.

We provide support for the use of wildcard expressions in the method name of the signature. These wildcard expressions have the same semantics as in the well-known wildcard expression mechanism of AspectJ [13] pointcuts. This allows a KALA declaration to apply to multiple methods of the same class, effectively gathering a crosscutting specification into one module. This naming can indeed be seen as the pointcut specification of the advanced transaction management aspect.

In AOP terminology, the join-point model [10] of KALA contains method descriptions. The pointcut expressions for this model are the load-time naming expressions. They define at which points of the base program execution the code for the aspect, called the advice, will be executed.[1] In KALA, the advice code is the code contained in the body of the KALA program, i.e., the specification of transactional properties. We are aware that the load-time naming scheme we use for defining pointcuts is simple and that other aspect languages provide much more expressive forms of defining pointcuts. We have chosen to keep our naming scheme simple, as the use of more expressive pointcuts is outside of the scope of this research.

Note that, contrary to Java method signature definitions and AspectJ-like crosscut specifications [13], we require the list of formal parameters to also contain a name for each parameter. These names can be used in the runtime naming scheme, which we discuss next.

Runtime naming

In addition to the static scheme, we also need a runtime, dynamic, naming scheme, as we discussed in Section 3.4. This is required to allow a transaction specification to define dependencies, views and delegations between different transactions at runtime. Our TP Monitor provides a name service for this, and the use of this service is integrated in KALA by the following keywords: `name` and `alias`. Registration of a transaction is performed by adding a `name` statement to the transactional properties. This statement takes two parameters: an identifier for a transaction, and a Java expression that may evaluate to a value of any type. This expression value is used as a key to store the identifier in the global naming service. In other words, the transaction is given a globally accessible name. Lookup of identifiers is performed using an `alias` statement, which takes the same type of arguments. This statement binds the result of the lookup using that key to a local alias, i.e., the alias is made to refer to a transaction identifier. In both these statements, the expression for the key may have sub-expressions that refer to an alias or to the name of a formal parameter as specified in the load-time naming. At runtime, when the expression is evaluated, formal parameters are bound to the actuals of the method call and aliases are resolved to the unique identifier of an already-running transactional method. As a result, the sequence of naming statements is of importance; names are accessible only after they have been defined. A reserved alias is the pseudo-alias `self`, which always designates the current transaction.

As an example of dynamic naming, consider the above method `root`. In Section 4.2 it will become the root of a tree of nested transactions. Methods called by `root` will be child transactions, and these will need to be able to obtain a reference to the root transaction. We show here how to provide the required naming infrastructure for this. First the root transaction adds itself to the name service, i.e., it names itself, using as key the current thread (which is obtained through the Java expression `Thread.currentThread()`). If we assume that called methods that match the `child*`

[1] Conceptually, join-points only exist at runtime, and therefore advice code execution can only be determined at runtime. However, due to the simplicity of our join-point model, this can be determined at load time, which amounts to an optimization.

method name wildcard expression will run in the same thread, these can use the current thread as a key to obtain an alias to the root transaction. The code for this is shown below:

```
util.strategy.Hierarchical.root(int foo) {
   name(self Thread.currentThread());}
util.strategy.Hierarchical.child*() {
   alias(root Thread.currentThread());}
```

Note that the above specification yields a runtime tree structure with maximum depth of one. Any `child*` method called from the `root` method, directly or indirectly, will refer to this root as its parent. This is even so if a method, say `childG`, has actually been indirectly called from the root, through another child method, say `childP`. In this case, it can be argued that `childG` should behave as a grandchild of the root and a child of `childP`. Such an implementation is possible in KALA, as we show next.

Within the top-level block, shown in the example above, nested blocks can be placed. These contain statements to be executed at begin, commit or abort time, and are therefore named `begin`, `commit` and `abort` blocks. We use a `commit` and an `abort` block to define nested transactions with a dynamically determined tree structure as follows:

```
util.strategy.Hierarchical.child*() {
   alias(parent Thread.currentThread());
   name(self Thread.currentThread());
   commit { name(parent Thread.currentThread());}
   abort { name(parent Thread.currentThread());} }
```

The above code ensures that the `Thread.currentThread()` key is always associated with the currently active nested transaction. To achieve this, it overwrites the name of the parent transaction in the naming service on line 3 with the identifier of this transaction. As a result, any other child method, say `childG`, that gets called from this method, say `childP`, will regard `childP` as its parent. When `childP` ends either in a commit or abort it restores the original value of the parent in the naming service.

We have lexical scope of names with regard to nested blocks. Within an inner block names of the enclosing block are visible. Furthermore, we also allow variable overriding: within a nested block variables are allowed to be locally redefined.

Runtime grouping

KALA also provides for transactions to be placed in named groups. This is performed using the `groupAdd` statement, which takes an alias and a key expression as arguments, as in the `name` statement. If the group with that key does not exist, it is automatically created. Only transactions can be added to groups, i.e., groups cannot contain other groups. Groups can also be registered in the name service through a `name` statement and looked up using an `alias` statement. The expressions in these statements may also refer to names of groups in addition to names of transactions and formal parameters of the method. We do not provide example code using groups here, an illustration if this is given in Section 4.3.

The `alias` statement is used to lookup both groups and transactions. To enable this, the lookup algorithm is structured as follows:

(1) The key is looked up in the namespace for group names. If it is found, the group is returned;
(2) Otherwise, the key is looked up in the namespace for transaction names. If it is found, the transaction is returned;
(3) If the key is not found (lookup failure), the *null transaction* is returned.

The null transaction is a dummy transaction: all KALA statements that have the null transaction as one of their arguments fail, emitting a warning. It is clear that such failure handling is simplistic, and appropriate treatment of KALA failures is a topic of current research.

The type of the resulting variable binding, i.e., the alias is to a transaction or to a group, is kept as metainformation of the binding. This allows all KALA statements to have overloaded behavior, depending on the runtime type of the alias. For example, for the `name` statement, this means that the new name is either registered in the name space for transactions or the name space for groups, depending on the type of the alias.

4.2. Dependencies, views and delegations

Dependencies, views and delegations are concepts of ACTA which are directly represented as statements in KALA. These declarations apply at begin, commit or abort time of a transaction and are therefore placed in `begin`, `commit` and `abort` blocks. This can be seen in the example (further expanding the description of a child transaction for nested transactions) shown below:

```
util.strategy.Hierarchical.child*() {
   alias(parent Thread.currentThread() );
   name(self Thread.currentThread());
   begin { dep(self wd parent, parent cd self);
           view(self, parent); }
   commit { del(self, parent);
            name(parent Thread.currentThread()); }
   abort { name(parent Thread.currentThread());} }
```

In this example, the `alias` statement looks up the parent transaction and binds it to the variable `parent`, before actually overwriting it as discussed previously. In the `begin` block, the `dep` statement declares a dependency between two transactions, in this case the $\mathcal{WD}$ and $\mathcal{CD}$ dependencies between parent and child. This ensures that the child does not commit after the parent has committed and that if the parent aborts the child will not commit later on. Also in the `begin` block, the `view` statement declares that the child sees the operations of the parent. In the `commit` block, the `del` statement performs delegation from child to parent transaction. Note that, within a given block, the sequence of dependency, view and delegation statements is unimportant. This is because these axioms are said to apply simultaneously at that instant in time.

Recall that KALA does not implement the dependency checking mechanism. This functionality is delegated to the TP Monitor, in this case ATPMos. As a result, the types of dependencies that can be set in KALA are only limited by the support of the TP Monitor. In ATPMos this list is open-ended: programmers can add support for new dependencies. We will touch on this in Section 5.

The `dep`, `view` and `del` statements have overloaded behavior, depending on the type of arguments. If any argument of these statements is a group, the behavior is to perform the action for each member of the group. For example, if a view is set from a group to a transaction, this view is set from all members of the group to this transaction. If a view is set from a group to another group, this view is set between each member of the source group and each member of the destination group. The only exception is the `del` statement: as delegation to multiple transactions cannot be performed, if a group is given as destination the `del` statement fails.[2]

An important advantage of the use of KALA is that it hides the process of checking and enforcing dependencies. Dependencies are verified at begin, commit and abort time. At these times, the transaction communicates with the TP monitor, and waits either until all dependencies are satisfied and it may proceed, or until it is forced to commit or abort to satisfy a dependency. Without the use of KALA, the programmer needs to implement this decision logic by hand. The KALA programmer is relieved from this burden, so that he can concentrate solely on declaring the transactional properties of the method.

4.3. Transaction life-cycle management

Specifying secondary transactions

Support for secondary transactions, introduced in Section 3.3, is provided by KALA through a specific `autostart` statement. The use of this statement, in the top-level block of KALA statements for a method, fully automates starting a secondary transaction in a separate thread. The `autostart` statement starts a new thread in which the method of the secondary transaction, given as the first argument, is immediately called. The second argument to the `autostart` statement is the list of actual parameters for that method. Each of these parameters can be any Java expression. As

[2] In the current implementation, the failure just emits a warning.

in `name` and `alias`, these expressions may also refer to the formal parameters in the load-time naming signature and to aliases. The third argument is a full KALA specification for the transactional properties of the method *when it runs in the autostart*. This argument will typically be used to allow dependencies to be set that restrict the secondary transaction to only begin when these are satisfied.

For an example of autostarts, we consider the Sagas ATMS that we briefly discussed in Section 3.3 and its use of compensating transactions. Recall that a compensating transaction implements a semantic 'undo' action for the transaction it is associated with. In this example, we consider a bank transfer saga which has been split into different steps. One of these steps performs the actual money transfer, implemented in a method `Cashier.transfer(BankAccount source, BankAccount dest, int amount)`, the KALA code of which is shown below. The compensating transaction for a bank transfer is the inverse transfer operation, i.e., calling the `transfer` method with `source` and `dest` swapped. This can be seen in the first and second argument of the `autostart` statement below. The third argument solely specifies that the autostart transaction names itself.

```
Cashier.transfer
 (BankAccount source, BankAccount dest, int amount) {
  alias (saga Thread.currentThread());
  autostart (Cashier.transfer
             (BankAccount source, BankAccount dest, int amount)
    <dest, source, amount> {
      name(self "CompOf"+saga);
    });
  begin { alias (comp "CompOf"+saga);
          dep(saga ad self, self wd saga, comp bcd self); }
  commit { alias (comp "CompOf"+saga);
           dep(comp cmd saga, comp bad saga);} }
```

We assume that the `transfer` method is called from a method that represents the saga. That method implements the sequencing of the saga by calling the different steps in succession and names itself in a similar way as the root transaction in Section 4.1. The block of KALA statements in the autostart here use the alias obtained to the saga to give the compensating transaction a unique name. This name is looked up in the begin and commit blocks of the top-level KALA specification, which allows dependencies to be placed between the compensating transaction and the transfer operation, and between the compensating transaction and the saga. These dependencies are in place to ensure that the compensating transaction for this step only runs if the saga rolls back after this step has already committed. We do not discuss these dependencies in detail here, as such a description is outside of the scope of this paper. Instead we refer to the definition of these dependencies in the ACTA formal model [4,1].

Transaction termination

A second life-cycle operation available in KALA is the termination of transactions. As stated in Section 3.3, this is required because even when a transaction ends, its dependencies may still need to be kept in the system. When these are no longer needed, the transaction can be stopped and removed from the system by means of the `terminate` statement. This statement takes as argument an expression that evaluates to the name of a transaction, as in the `alias` statement. Termination can be specified to be performed at any of the significant events of a transaction. Therefore, these statements are also grouped in `begin`, `commit` or `abort` blocks. Furthermore, `terminate` has overloaded behavior similar to `dep`, `view` and `del`: applied to a transaction, this transaction is terminated, and applied to a group all transactions of this group are terminated.

Termination is illustrated by the example below:

```
util.strategy.Hierarchical.root(int foo) {
   name( self Thread.currentThread());
   commit{ terminate("ChildrenOf" + self);
           terminate(self); }
   abort{ terminate("ChildrenOf" + self);
          terminate(self); } }
```

```
util.strategy.Hierarchical.child*() {
   alias(parent Thread.currentThread() );
   name(self Thread.currentThread());
   groupAdd(self "ChildrenOf" + parent);
   begin { dep(self wd parent, parent cd self);
           view(self, parent); }
   commit { del(self, parent);
            name(parent Thread.currentThread());
            terminate("ChildrenOf" + self); }
   abort { name(parent Thread.currentThread());
           terminate("ChildrenOf" + self);} }
```

This example completes the previous description of nested transactions: the children now add themselves to a group uniquely identified by the parent alias. When this parent ends, either by commit or abort, that group is terminated. As the root is not part of a parent group, it also needs to terminate itself when finished.

4.4. *Defining advanced models versus using advanced models*

KALA code defines an ATMS by combining the fundamental ACTA building blocks and linking these combinations to a part of the application being developed. As a result, the transactional behavior of that part of the application is defined. Consequently, everywhere the same ATMS is used this tedious and complex process is repeated. This will lead to code duplication, which is better avoided. Furthermore, this process entails that the application programmer should know the complex technical implementation of the ATMS, which requires intricate work.

It would be better to avoid the need for an application programmer to write low-level code. This programmer, simply using an ATMS, should not be exposed to the internals of this ATMS. As the application programmer will not want to modify the ATMS behavior such internals are irrelevant. More importantly, this is dangerous as it needlessly exposes the internals of the ATMS, breaking encapsulation and more easily allowing errors to be written.[3]

We have proposed a solution to this problem in [14], of which we provide a brief summary here. Our solution is based on KALA as a generally applicable programming language for declaring the transactional properties of code for an ATMS. In other words, KALA can be used as a *kernel* for other programming languages. These then further simplify specifying the transactional properties of code for a given ATMS, trading off generality. Such further simplification is possible by using an extra level of abstraction. These languages abstract from the internals of the ATMS, instead letting the application programmer reason about the concepts present in that ATMS. As a result, programmers simply use the ATMS as a black box component. At compile time the DSL code is translated to equivalent KALA code. This KALA code is then woven into the base concern.

Using KALA as a kernel allows for relatively straightforward definitions of new DSLs once their implementation in KALA is known. All that is required is a translation from the DSL program to the equivalent KALA program.

We briefly treat two of the DSLs we created here: a DSL for classical transactions, and one for nested transactions. For a more detailed discussion of these DSLs, and a DSL definition of Sagas we refer to [14].

DSL for classical transactions

In classical transactions only one concept is present, which is the indication that a method is a transaction. Therefore, the DSL for classical transactions is simple: programs consist of a list of declarations declaring a method to be transactional. Such a declaration takes the form of a `trans` keyword, followed by the full signature of a method, as in load-time naming (see Section 4.1).

For example, consider the method with name `methodName` with parameter list `parameterList`, of the class `className`, contained in the package `packageName`. To declare this method transactional, the following line of code in the transactions DSL suffices:

[3] Following the simple rule that the more code is written, the higher chance for errors in that code.

```
trans packageName.className.methodName(paramList);
```

The equivalent KALA code for this program is given below:

```
packageName.className.methodName(paramList) {
   commit {terminate(self); }
   abort {terminate(self); } }
```

This code is quite straightforward, only specifying termination of the transaction at commit and abort time.

Nested transactions DSL

The nested transactions ATMS adds the concept of a transaction hierarchy to classical transactions. In general, this hierarchy can be arbitrarily structured. An arguably convenient structure is the hierarchy of callers, which we have used in our running example of nested transactions. We therefore also have defined a DSL for this specific case of nested transactions.

In this DSL, the new concept is therefore that this method is a sub-transaction of the (possibly indirectly) calling transaction. This is declared by reusing the naming from the DSL for classical transactions and extending it with the `extends caller` statement. Root transactions can be stated explicitly in the DSL as well, and as they do not extend any other transaction, they are therefore specified by omitting `extends caller`.

Both kinds of transactions can be seen in the example below, which at compile time translates to the example code for nested transactions we have given in Section 4.3

```
trans util.strategy.Hierarchical.root(int foo);
trans util.strategy.Hierarchical.child*() extends caller;
```

To summarize, our solution to the problem of needing to know the internals of an ATMS and the inherent code duplication is to add an extra layer of abstraction. This is done through the use of a domain-specific language per ATMS. The use of such model-specific languages allows us to isolate the aspect programmer from the implementation details of the transaction. Instead, the programmer now focuses solely on the concepts defined in the ATMS being used.

5. KALA implementation

An important facet of any programming language is how it is implemented. In this section we give a brief overview of the implementation of KALA.

KALA declares, at a high conceptual level, the transactional properties of a method, but these high-level properties need to be translated to lower-level transaction demarcation code. At runtime, an application that has a KALA specification needs to run such demarcation code, interfacing with our TP Monitor, ATPMos, which supports the ACTA axioms. The interplay of the behavior encoded within the demarcation code and the runtime enforcement of ATPMos yields the overall application behavior that conforms to the ATMS specified in KALA.

ATPMos is a lock-based TP Monitor, of which the standard transactional functionalities have been extended to provide support for dependencies, views and delegation. Views are implemented by extending the lock conflict checking to also take into account view relationships. Delegation is straightforwardly implemented by transferring the locks from the delegator to the delegatee. Dependency support is realized through the use of a constraint engine, as discussed in Section 3.2. Dependencies are implemented as constraint specifications for this engine. As we have already mentioned, the list of dependencies supported by ATPMos is open. Support for new dependencies is achieved by creating a new constraint specification, which can even be added while ATPMos is running. Lastly, ATPMos provides a basic implementation of a naming and grouping service, as required to enable runtime naming and grouping. We do not discuss the implementation of ATPMos further in detail here, as it is outside of the scope of this text. For a more detailed discussion we refer to [15].

In the remainder of this section, we give a brief overview of how the high-level KALA declarations are translated to demarcation code that properly interacts with ATPMos. First of all, we introduce the infrastructure used to support KALA in the Java language, and then we discuss some elements of the implementation.

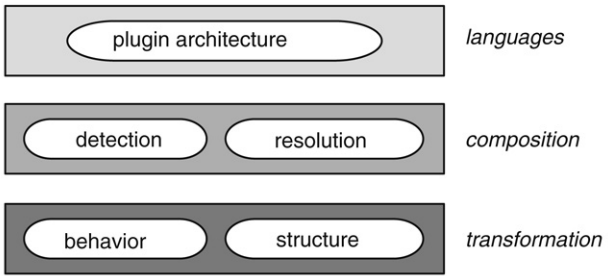

Fig. 3. Architecture of a versatile kernel for multi-language AOP.

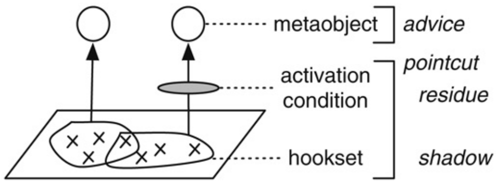

Fig. 4. The link model and correspondence to AOP concepts.

5.1. Multi-language AOP and reflex

The KALA domain-specific aspect language is implemented over Reflex,[4] a versatile kernel for multi-language AOP in Java [16]. As such, Reflex is specifically designed as a powerful backend for implementing (domain-specific) aspect languages. Its three-layer architecture (Fig. 3) consists of: first a general-purpose transformation layer at the bottom; on top of this transformation layer, a composition layer that takes care of both detection and resolution of aspect interactions [17]; a plugin architecture for the modular definition of (domain-specific) aspect languages completes the picture. This last layer supports the translation of aspects expressed in an aspect language into lower-level kernel constructs.

Of interest to us in this paper are the top and bottom layers. The transformation layer is indeed the weaver. It relies on a model of partial reflection, both behavioral and structural. The behavioral part is based on a model of explicit *links* presented in [18], pictured in Fig. 4. A link binds a *hookset* (i.e., a set of program points) to a *metaobject* [19] implementing the corresponding behavioral extension. The actual communication protocol between the base code denoted by the hookset and the metaobject is fully customizable.

The language layer is implemented with the MetaBorg approach to language embedding and assimilation [20]. MetaBorg consists of a combination of the SDF syntax definition formalism [21], which is particularly well suited for parsing languages with complex syntax (such as aspect languages like AspectJ [13]) in a modular fashion [22], and the rule-based transformation system Stratego/XT [23] for compiling the parse tree produced by the SDF-generated parser into Java code using the Reflex API for composition and transformation.

5.2. Overview of the KALA implementation

In the above setting, the implementation of KALA is quite straightforward, since weaving is supported by the Reflex backend, and generation of Java code is handled declaratively in Stratego rules. This section does not enter into the details of the implementation, as this is extensively discussed in [24]. Here we restrict our discussion to an overview of how the main elements of KALA are supported.

Runtime behavior. The runtime behavior of KALA is implemented by a reusable metaobject implementing the core demarcation logic. This metaobject is parameterized by the KALA specification, which in Reflex is expressed as a number of parameters passed to the KALA metaobject.

The demarcation logic, as in demarcation code for classical transactions, starts and ends the transaction, and includes data accesses of the method in the transaction. Furthermore, it performs naming and grouping, sets

[4] http://reflex.dcc.uchile.cl/.

dependencies and views, and performs delegations at the different phases in the life-cycle of the transaction. Lastly, it ensures, in cooperation with ATPMos, that dependencies are adhered to. As we discussed in Sections 3.2 and 4.2, at each phase in the life-cycle it is determined if the transaction may proceed as planned, needs to wait, or is required to commit immediately or abort immediately to satisfy a dependency.

Load-time naming. The identification of methods in KALA code is straightforwardly expressed in Reflex as a hookset: declaring in KALA that a method is transactional implies configuring Reflex so that executions of this method are intercepted and subsequent control is given to the KALA metaobject.

Runtime Naming, Grouping and Termination. ATPMos provides the required naming service for runtime naming and grouping. KALA code is translated to executable Java code that calls this service. This is implemented by generating anonymous inner classes with the corresponding code. Similar code is generated for termination, first calling the name service and second calling ATPMos to perform termination on the resulting transaction or group.

Alias bindings, i.e., local references to transactions, and bindings between formal parameters and their values are managed using an environment object. This object also takes care of scoping of `begin` `commit` and `abort` blocks and for `autostart` statements.

Dependencies, Views, Delegations. The ACTA axioms of dependencies, views and delegations are now represented simply as objects passed to the KALA metaobject. At begin, commit and abort time, the metaobject first executes the naming and grouping code generated for that life-cycle phase. Second, dependencies and views are set and delegations are performed, by looking up the aliases in the environment object and calling ATPMos using the result of the lookup.

Autostarts. Autostarts are implemented by generating anonymous inner classes as a subclass of `Runneable`, which allows instances of them to run in a separate thread. These classes contain one `run` method, which simply calls the method of the secondary transaction, as specified in the KALA program. The metaobject for this method call is given access to the environment in which the `autostart` statement is nested, and configured with the transactional properties as specified in the `autostart` statement. To run the autostart a new thread is started that calls the `run` method on the instance, effectively running the method in a separate thread.

6. Benefits of using KALA

To validate the use of KALA, we have implemented the use of five different ATMS: two variations of Nested Transactions [3], Sagas [12], Relatively Consistent Schedules [25], and Cooperating Nested Transactions [15]. Of these five, the last two ATMS did not have any formal specification published in ACTA. This not only illustrates that we can indeed tackle a wide variety of ATMS, but also that we can provide support beyond the ATMS for which a formal description has been published.

In these implementations, we have seen that using KALA allows us to achieve obliviousness [26] at the level of the code, i.e., the code for the base concern does not include any transaction-related code. All the code for the transaction management concern is completely contained in KALA modules. As a result, we achieve a high degree of separation of concerns for the transaction management concern. Given the known benefits of separation of concerns, this significantly increases ease of development and maintenance of these applications.

An arguably important form of transaction management in Java that also achieves a degree of obliviousness is declarative transaction management in the Enterprise JavaBeans (EJB) middleware standard [27]. In this setting, the use of transactions is specified by defining transaction attributes in a separate file, the deployment descriptor. This deployment descriptor contains various types of metadata, such as the name of the bean, its type, security properties for its methods and transaction attributes for its methods. The list of possible transaction attributes is however fixed, and none of the attributes allow for ATMS. Therefore only classical transactions can be used. Research has been performed by Procházaka [28] on an extension of this list, as well as an implementation of the standard that supports this extension. This work is known as Bourgogne Transactions. It allows for a number of ATMS to also be used, by extending the list of possible transaction attributes to allow the specification of delegation and dependencies. The types of dependencies are however fixed, in contrast to KALA, where this is open-ended. Furthermore, Bourgogne transactions provides no support for views, which is required, e.g., for nested transactions.

In addition to yielding the benefits of separation of concerns, KALA raises the level of abstraction of the specification of the transactional properties involved. In KALA, the programmer solely needs to state the transactional properties of the code in terms of the ACTA axioms, as well as performing naming and life-cycle management declaratively. This has to be contrasted to writing demarcation code directly in Java. Transactions also are set to begin and end on method boundaries, but the programmer has to perform additional tasks: verifying and enforcing dependencies, explicitly starting and ending transactions and including operations on shared data in the transaction. Therefore, as a result of raising the level of abstraction using KALA, we obtain much more concise code, where fewer tasks have to be performed by the programmer.

As an illustration of the degree of conciseness, consider a simple bank transfer operation for which we use the Sagas ATMS [12], which we have briefly described in Section 4.3. We do not include the full example here, but we report on the amount of code required to implement it. The Java code for the bank transfer is four methods, totaling 37 lines of code. If we manually add demarcation code this bloats to 267 lines of code, i.e., over seven times as much code. If we use KALA instead, we only need to add 52 lines of demarcation code to use the Sagas ATMS.

In contrast, if we were to use a general-purpose aspect language, such as in [5–7], where AspectJ [13] is used, we cannot achieve this large amount of code reduction. We can implement the core logic of the demarcation object as a reusable part of the aspect, but configuring this metaobject requires some work. Firstly, for each cluster of naming and grouping operations, i.e., top-level, begin, commit and abort time, an anonymous inner class needs to be written by hand that evaluates the key expressions. Furthermore, in this expression, sub-expressions which are references to aliases and parameters of the method need to be replaced with lookups in the environment. Secondly, for termination a similar anonymous inner class needs to be written by hand. Thirdly, for each autostart an anonymous inner class needs to be written, also by hand, that calls the method for the secondary transaction.

We have not performed the experiment of implementing the use of Sagas in a general-purpose aspect language. Therefore, we do not have the data to compare lines of code required. It is however clear that more code is needed than in KALA as all the support code for parameterizing the metaobject, especially considering the anonymous inner classes, is more verbose than the equivalent KALA code.

7. Conclusion

In this paper we have detailed the creation process of KALA: a domain-specific aspect language for using advanced transaction management in a distributed system. In major contrast to other work on treating transactions as an aspect, KALA is built upon a formal foundation, which is the ACTA formalism. In this paper we first discussed how we transformed the ACTA formalism into the KALA aspect language. We then proceeded with giving an introduction to KALA, discussing its implementation with the Reflex AOP kernel, and finally talking about the benefits of using KALA.

We conclude that, because KALA is built upon a formalism that covers a wide variety of ATMS, KALA also is able to cover a wide variety of ATMS. Its expressiveness therefore goes much farther than other aspect-oriented proposals for transaction management. Furthermore, we have a domain-specific aspect language that exposes the concepts of the ATMS domain, while automating as much of the implementation of these concepts as possible. As a result, this domain-specific aspect language significantly raises the level of abstraction and makes the specification of the transactional properties of an application much more concise.

Acknowledgments

We thank the proof-readers Pascal Constanza, Andy Kellens, Coen De Roover and Thomas Cleenewerck for their invaluable insights and helpful comments. É. Tanter is partially financed by the Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile, and by FONDECYT Project 11060493.

Availability. The implementation of KALA on top of Reflex, called ReLAx is available on the PLEIAD website: http://pleiad.dcc.uchile.cl/relax.

References

[1] A.K. Elmagarmid (Ed.), Database Transaction Models For Advanced Applications, Morgan Kaufmann, 1992.
[2] S. Jajodia, L. Kershberg (Eds.), Advanced Transaction Models and Architectures, Kluwer, 1997.
[3] J.E.B. Moss, Nested transactions: An approach to reliable distributed computing, Ph.D. Thesis, Massachusetts Institute of Technology, 1981.
[4] P.K. Chrysanthis, K. Ramamritham, A formalism for extended transaction models, in: Proceedings of the 17th International Conference on Very Large Data Bases, 1991, pp. 103–112.

[5] J. Kienzle, R. Guerraoui, AOP: Does it make sense? — the case of concurrency and failures, in: B. Magnusson (Ed.), Proceedings of the 16th European Conference on Object-Oriented Programming, ECOOP 2002, in: Lecture Notes in Computer Science, vol. 2374, Springer-Verlag, Málaga, Spain, 2002.
[6] A. Rashid, R. Chitchyan, Persistence as an aspect, in: M. Akşit (Ed.), Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, AOSD 2003, ACM Press, Boston, MA, USA, 2003, pp. 120–129.
[7] S. Soares, E. Laureano, P. 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, ACM Press, Seattle, Washington, USA, 2002, pp. 174–190. ACM SIGPLAN Notices, 37 (11).
[8] A. van Deursen, P. Klint, J. Visser, Domain-specific languages: An annotated bibliography, SIGPLAN Notices 35 (6) (2000) 26–36.
[9] M. Mernik, J. Heering, A.M. Sloane, When and how to develop domain-specific languages, ACM Computing Surveys 37 (4) (2005) 316–344.
[10] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: M. Akşit, S. Matsuoka (Eds.), Proceedings European Conference on Object-Oriented Programming, vol. 1241, Springer-Verlag, Berlin, Heidelberg, New York, 1997, pp. 220–242.
[11] J. Gray, A. Reuter, Transaction Processing, Concepts and Techniques, Morgan Kaufmann, 1993.
[12] H. Garcia-Molina, K. Salem, Sagas, Proceedings of the ACM SIGMOD Annual Conference on Management of Data, 1987, pp. 249–259.
[13] The AspectJ project. http://eclipse.org/aspectj/, 2006.
[14] J. Fabry, T. D'Hondt, A family of domain-specific aspect languages on top of kala, in: Open and Dynamic Aspect Languages Workshop at AOSD06, March 2006.
[15] J. Fabry, Modularizing advanced transaction management - tackling tangled aspect code, Ph.D. Thesis, Vrije Universiteit Brussel, Vakgroep Informatica, Laboratorium voor Programmeerkunde, PROG, July 2005.
[16] É. Tanter, J. Noyé, A versatile kernel for multi-language AOP, in: R. Glück, M. Lowry (Eds.), Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering, GPCE 2005, in: Lecture Notes in Computer Science, vol. 3676, Springer-Verlag, Tallinn, Estonia, 2005, pp. 173–188.
[17] É. Tanter, Aspects of composition in the reflex AOP kernel, in: W. Löwe, M. Südholt (Eds.), Proceedings of the 5th International Symposium on Software Composition, SC 2006, in: Lecture Notes in Computer Science, vol. 4089, Springer-Verlag, Vienna, Austria, 2006, pp. 98–113.
[18] É. Tanter, J. Noyé, D. Caromel, P. Cointe, Partial behavioral reflection: Spatial and temporal selection of reification, in: R. Crocker, G.L. Steele Jr. (Eds.), Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003, ACM Press, Anaheim, CA, USA, 2003, pp. 27–46. ACM SIGPLAN Notices, 38 (11).
[19] P. Maes, Concepts and experiments in computational reflection, in: OOPSLA '87: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications, ACM Press, New York, NY, USA, 1987, pp. 147–155.
[20] M. Bravenboer, E. Visser, Concrete syntax for objects, in: Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2004, ACM Press, Vancouver, British Columbia, Canada, 2004, pp. 365–383. ACM SIGPLAN Notices, 39 (11).
[21] E. Visser, Syntax definition for language prototyping, Ph.D. Thesis, University of Amsterdam, Sep. 1997.
[22] M. Bravenboer, E. Tanter, E. 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, ACM Press, Portland, Oregon, USA, 2006, pp. 209–228.
[23] E. Visser, Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9, in: Domain-Specific Program Generation, in: Lecture Notes in Computer Science, vol. 3016, Springer-Verlag, 2004, pp. 216–238.
[24] J. Fabry, E. Tanter, T. D'Hondt, ReLAx: implementing KALA over the reflex AOP kernel, in: Proceedings of the 2nd Workshop on Domain Specific Aspect Languages (Vancouver, British Columbia, Canada, March 12–12, 2007), DSAL '07, ACM, New York, NY, doi:http://doi.acm.org/10.1145/1255400.1255403.
[25] A. Aziz Farrag, M. Tamer Özsu, Using semantic knowledge of transactions to increase concurrency, ACM Transactions on Database Systems 14 (4) (1989) 503–525.
[26] R.E. Filman, D.P. Friedman, Aspect-oriented programming is quantification and obliviousness, in: OOPSLA 2000 Workshop on Advanced Separation of Concerns, Minneapolis, MN, 2000.
[27] R. Monson-Haefel, Enterprise JavaBeans, 3rd ed., O'Reilly, 2001.
[28] M. Prochazka, Advanced transactions in enterprise javabeans, in: EDO '00: Revised Papers from the Second International Workshop on Engineering Distributed Objects, Springer-Verlag, London, UK, 2001, pp. 215–230.