KALA: A Domain-Specific Solution to Tangled Aspect Code

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

Nicolas Pessemier
INRIA Futurs - LIFL, Projet Jacquard/GOAL
Bâtiment M3
59655 Villeneuve d'Ascq, France
nicolas.pessemier@lifl.fr

1. Introduction

In multi-tiered distributed systems transaction management has long been a mainstay of concurrency management. Transactions were however originally conceived only for brief and unstructured database accesses. Because of this they are a poor match for applications that wish to access data in a more structured way, or for a relatively long time. Negative consequences of this mismatch are, for example, that transaction throughput is only optimal when each transaction has a very short life-time. The multiple shortcomings of classical transactions are recognized by an important body of work in the transaction management community. To address them, many advanced transaction models (ATMS) have been developed, including a formalism called ACTA [CR91]. Each of these advanced models addresses a subset of the known shortcomings of classical transactions.

As with classical transactions, advanced transaction management is a cross-cutting concern. We have therefore investigated how it can be modularized into an aspect and developed the domain-specific aspect language KALA [FD06]. KALA is based on the ACTA formalism, and KALA programs declare how a particular application uses an ATMS, as expressed in ACTA.

When performing this research, we encountered a problem in the aspect code itself. We found that because the ATMS concern is a complex concern it can be subdivided in multiple sub-concerns, and that the code for these sub-concerns cross-cut the aspect itself. This yields aspect code which itself tangles multiple concerns. We therefore termed this phenomenon *Tangled Aspect Code*.

In this paper we describe how KALA is able to address the problem of Tangled Aspect Code through the use of domain information. KALA was developed solely for the domain of ATMS, and with the intent to tackle this problem. As a result the modularization for sub-concerns offered by KALA is straightforward for the programmer and the composition of sub-concerns requires no programmer intervention.

2. KALA in a Nutshell

2.1 Advanced Transaction Management

As we have said above, a number of ATMS have been developed, each addressing a specific set of shortcomings of classical transactions. We do not give an overview of these models here, as this is outside of the scope of this paper. Instead we briefly discuss two well-known models: Nested Transactions [Mos81] and Sagas [GMS87]. These are illustrated in Figures 1 and 2, and we provide a short description of these models next.

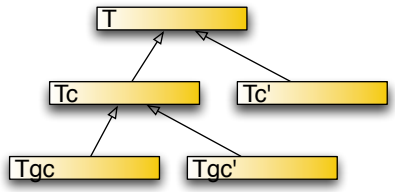

Figure 1. The Nested Transactions ATMS

Nested transactions [Mos81] is one of the oldest and easily the most well-known ATMS. It enables a running transaction T to have a number of child transactions Tc. Each Tc 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. Tc may itself also have a number of children Tgc, forming a tree of transactions. When a child transaction Tc commits its data, this data is not written to the database, but instead *delegated* to its parent T, where it becomes part of the data of T. If a transaction Tx is the root of a transaction tree, *i.e.* it has no parent, Tx's data will be committed to the database when T commits. Lastly, if a child transaction Tc aborts, the parent T is unaffected. T is not required to also abort, *i.e.* when it ends it may choose freely to either commit or abort.

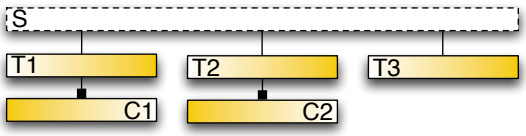

Figure 2. The Sagas ATMS

Sagas [GMS87] is, next to Nested Transactions, one of the oldest ATMS and also arguably one of the most referenced ATMS in the community. Sagas is tailored towards long-lived transactions. Instead of one long transaction T, a saga S splits T into a sequence of sub-transactions $T1$ to Tn. Each sub-transaction is a normal classical transaction and this sequence is executed completely before the saga commits. To abort or rollback a running saga S, the currently running sub-transaction Ti is aborted and the work of already committed transactions $T1$ to $Ti - 1$ has to be undone,

as their results have already been committed to the database. To allow this, the application programmer has to define for each subtransaction Ti a *compensating transaction* Ci that performs a semantical compensation action. To undo the work of $T1$ to $Ti-1$, $C1$ to $Ci-1$ are run by the runtime transaction monitor in inverse sequence, *i.e.* starting with $Ci-1$.

As said above, and illustrated by these two examples, different ATMS exhibit different concurrency management properties. This allows a given application to choose the advanced model that provides the best match to the concurrency management properties it requires. Also, if no matching model exists, it is possible to create a new model that provides the properties required by the application. However, as we argue in [Fab05] when using traditional software engineering approaches, there is only a small degree of separation of concerns between the ATMS concern and the other concerns present in the application.

2.2 ACTA and KALA

In general, to use transactions in an application, the developer needs to add *transaction demarcation code*, which is spread throughout the entire application. Using aspects, however, previous work has successfully achieved the modularization of the concern of classical transaction management [KG02, RC03, SLB02].

Advanced transaction models also suffer from the problem of cross-cutting demarcation code [Fab05]. Therefore, we created the domain-specific aspect language called KALA [FD06] to modularize advanced transaction management as an aspect for Java applications. KALA is based on the ACTA formalism for ATMS [CR91], which is accepted in the community as covering a wide field of advanced transaction models. In ACTA, extra properties are given to classical transactions, or properties of such transactions are modified, resulting in a collection of transactions that exhibits the behavior of an advanced model. The formalism declares three kinds of properties: *dependencies*, *views* and *delegation* which are declared between two transactions. The views and delegation properties correspond to viewing and delegation between them, which we have mentioned above when discussing nested transactions. Dependencies set relationships between two transactions and can be used to, for instance, sequence multiple transactions or trigger the beginning of a compensating transaction, which will be illustrated in Sections 4.2 and 6.1.1. KALA reifies the ACTA constructs of dependencies, views and delegation as the `dep`, `view` and `del` statements in the language. A full discussion of KALA is outside of the scope of this paper, instead we give a brief overview here. For a full description we refer to [FD06, Fab05].

A KALA program specifies what dependencies, views and delegations apply at the begin, commit and abort time of a transaction. As is the norm in multi-tier transactional systems, the life-cycle of a transaction coincides with the life-cycle of a method. The transaction begins when the method begins, commits when the method ends normally and aborts if the method ends with a (given type of) exception. All data accesses within this method (and within the methods called by this method) are included in the transaction. To identify this method, the signature of the method is used, possibly using wildcards, similar to AspectJ [asp06]. This yields the following overall structure of the KALA declarations for a method (square brackets indicate placeholders for actual KALA statements):

```
1 MethodSignature(ArgumentList){
2    [ preliminaries ]
3    begin { [ begin time properties ] }
4    commit { [ commit time properties ] }
5    abort { [ abort time properties ] }
6 }
```

Note that the depencency, view and delegation specifications inside a `begin`, `commit` and `abort` block are considered to happen in the same atomic action. Therefore the sequence of these statements within such a block is of no importance.

In order for dependencies, views and delegation to be applied to two transactions the KALA code needs to be able to refer to these transactions. This is performed through the use of a global naming service. Within KALA code, a local reference to such a name, *i.e.* an alias, is obtained through the `alias` statement. This statement takes the alias for that transaction, and a Java expression that evaluates to the key that is looked up in the name service. This expression has access to the actual parameters of the method and to aliases which have already been resolved. The alias `self` is always bound to the currently executing transaction. An alias placed in preliminaries is looked up immediately before the transaction starts, and is accessible thoughout the remainder of the KALA code for that method. Aliases placed in `begin`, `commit` and `abort` blocks are looked up at that moment in the life-cycle of the transaction, and are only accessible at that time. A transaction can be added to the naming service, *i.e.* given a global name, using the `name` statement. This statement takes an alias (which may be `self`), and a Java expression that evaluates to the key for the naming service. Note that, contrary to dependencies, views and delegation, the sequence of `name` and `alias` statements is important, as the expressions used in these statements have access to already resolved aliases.

In addition to naming, KALA also provides support for groups. Transactions can be added to a named group using the `groupAdd` statements. KALA makes no distinctions between transactions and groups of transactions, *i.e.* all statements can take groups or transactions as arguments[1].

KALA requires the programmer to perform manual memory deallocation for transactions (equivalent to the `free` statement in C++). This is performed through the `terminate` statement, which takes as argument the alias of the transaction or group to be freed. Termination can be performed at begin, commit or abort time of a transaction. If the transaction being terminated has not yet committed, it will be immediately forced to rollback.

Lastly, the preliminaries may contain an `autostart` statement. This statement specifies that a separate transaction needs to be started, in parallel to this transaction. The autostart specifies the signature of the method corresponding to this transaction, a list of actual parameters, and a KALA specification for this transaction. Autostarts are used, for example, within the KALA specification of a transaction Ti of a Saga S to specify the compensating transaction Ci. This specification then also contains dependencies at begin, commit and abort time of Ti that restrict Ci to only run if the Saga is aborted.

3. Tangled Aspect Code

3.1 Sub-concerns in the Aspect

If we consider various ATMS from a conceptual point of view, we find that these ATMS are not one monolithic block, but incorporate different design decisions. For example, consider how rollback is handled in the Sagas ATMS: compensating transactions are executed in the inverse sequence of the steps of the saga. Translated to application code, *i.e.* methods, each step corresponds to a method, as is each counterstep. If we consider conceptually the tasks that need to be performed by the demarcation code for such a step, we can infer that some parts of this code treat managing rollback of the saga. This code performs the work of defining and starting up compensating transactions, ensuring that these only begin when the

[1] The only exception being that a group cannot be the destination of a delegation operation.

saga aborts, and that they run in the right sequence. All of these low-level tasks comprise the code for one concern, which is managing rollback of the saga.

We can indeed consider management of rollbacks a true concern in this demarcation code, as it is a design decision of the ATMS that lies conceptually at a higher level of abstraction than the implementation details of the code, *i.e.* the various tasks of the code we identified above. This corresponds to the original consideration of a concern by Parnas [Par72] where he states that a module, *i.e.* a concern, corresponds to the implementation of a design decision. We claim that management of rollbacks is part of the design of the ATMS, as different implementations of this concern can be easily envisioned. For example, we could specify that compensating transactions run in the same sequence as the steps of the saga, or even let the compensating transactions run in parallel, to attempt to speed up saga rollback.

Sagas demarcation code will however contain more than code for rollbacks. In addition to this, the general structure of the Saga as a sequence of steps, where each step is itself a transaction, also needs to be defined. In other words, if we reflect on the various tasks performed by saga demarcation code, we find that this code treats two different sub-concerns: first the management of the *structure* of the overall transaction, and second the management of how *rollback* is performed.

This conceptual decomposition of an ATMS into different sub-concerns is not unique to the Sagas ATMS. We have performed a similar analysis of various ATMS, and found that these are also composed of multiple sub-concerns [Fab05]. In addition to the two sub-concerns of structure and rollback handling identified above, we have encountered the sub-concerns of *view management* and *delegation management*. Note that the list of four sub-concerns of ATMS is open-ended. Although we have identified these sub-concerns in many ATMS, it is possible that a new ATMS contains a sub-concern which we have not yet encountered.

To summarize, we should not consider an ATMS conceptually as one monolithic block, but rather as a composition of a number of sub-concerns. We have identified four such sub-concerns so far: the structure of the advanced transaction, how rollback is handled, the management of views and the management of delegation.

3.2 Tangled Aspect Code

If we wish to modify sub-concerns of an ATMS, or add implementations for new sub-concerns to an existing ATMS, the aspect that modularizes the ATMS must take into account this requirement. The aspect must be structured in such a way that modification of sub-concerns is easy, enabling easy creation of new ATMS through changes in the implementation of these sub-concerns. In other words, such a conceptual separation into modules should therefore ideally also be present in the KALA code. This would bring the well-known advantages of Separation of Concerns to the level of the aspect.

We find, however, that such a separation into multiple modules is absent from the KALA code. The reason for this is the primary decomposition inherent in KALA code. KALA code reflects the life-cycle of a transaction, and is therefore subdivided into three different phases: a begin phase, a commit phase and an abort phase. In contrast to this, the implementation of a sub-concern can affect multiple phases in the life-cycle, and in one phase the code for multiple sub-concerns can be present. For example, in Sagas the code for the rollback concern is contained in the begin and commit blocks of various steps, and in the commit and abort blocks of the top-level Saga, as we will see in Section 4.2. As a result, the aspect code for the ATMS concern tangles the multiple sub-concerns present in the ATMS being implemented.

We can see this tangling as a case of the tyranny of the dominant decomposition [TOHJ99]. The dominant decomposition in KALA is the life-cycle of the transaction in begin, commit and abort phases. The modularization of sub-concerns of an ATMS, however, is orthogonal to time. One sub-concern can act at multiple points in the life-cycle of a transaction. As a result, the sub-concerns cross-cut the dominant decomposition, leading to code which is scattered and tangled. In other words, a KALA program is a combination of different sub-concerns and we see that the code of these sub-concerns is tangled. We call this phenomenon, where the aspect itself is a tangled mess of sub-concerns, *tangled aspect code*.

4. Separate Definition of Concern Code

Instead of having an ATMS as a monolithic block, we want to apply the known benefits of separation of concerns [HVL95] to the process of creating and modifying an ATMS. Applying separation of concerns here, *i.e.* programming an ATMS in multiple modules, will greatly ease implementation and modification of this ATMS. This enables a new ATMS to be built, or an existing ATMS to be adapted. This in effect tailors an ATMS to best fit the transactional properties required by the application being developed.

We have seen above that these concerns cut across the dominant decomposition of KALA code, which is the life-cycle of a transaction. Therefore a separate modularization mechanism is required for these concerns. KALA contains such a mechanism, which allows separating the specification of the different concerns in a straightforward manner by writing them as separate KALA files. The composition of these modules into a complete specification is fully automatic, and is discussed in the next section. The straightforward modularization and automatic composition is possible because we used the properties of the domain when creating KALA.

In this section, we show how, applied to a given application, KALA can be used to define the different concerns of an ATMS separately. We assume here that an analysis has first been made of the different concerns present in the ATMS being used, as we have performed in Section 3. The different concerns identified in such an analysis, applied to an application, can then be written down separately in multiple KALA files, *i.e.* one file per concern. We show this by taking the Sagas ATMS we analyzed in Section 3.1, and writing KALA code for this ATMS.

As a concrete example of KALA code for the use of Sagas, we use the example of a bank transfer operation we introduced in [Fab05]. This is part of an application for bank cashiers, servicing customer at the teller window. The transfer operation is split in three steps: a `transfer`, a `printReceipt` and a `logTransfer` method, all called in sequence from a `moneyTransfer` method. The first step performs the actual money transfer, the second step prints out a receipt for the customer, and the third step updates the global log of the bank. Note that we do not include the Java code of the bank transfer operation here, as it is not relevant to this discussion.

We identified in Section 3.1 that the Sagas ATMS is comprised of two concerns: first the management of the structure of the overall transaction, and second the management of how rollback is performed. To have an implementation of these transactional concerns for the bank transfer operation, we now write KALA declarations for all four methods first for the structure concern, and second for the rollback concern.

4.1 Sagas: Structure

The first concern we implement here, is the structure of the saga. Recall that we identified this concern as the management of the overall structure of the saga, in which the steps perform their work. The structure concern codifies the subdivision of the saga into

multiple steps, allowing each step to obtain a reference to the top-level saga, and ensures that after the saga has ended cleanup work is performed.

Note that although we subdivide the discussion of the implementation of this concern into two parts, all the code for the structure concern is implemented in one file, as is indicated by the continuity in line number counting.

4.1.1 Saga Top-level

The first KALA declarations we show are for the top-level `moneyTransfer` method and are given below. This code registers itself in the naming service, such that the steps in the saga, shown later, can obtain a reference to the saga. At commit and abort time, the unique identifier of this saga is used to refer to a group name which is therefore guaranteed to be unique for this saga. In this group, the various steps of the saga will have registered themselves. As a result, termination of this group implies termination of all the steps of the saga, and together with termination of the saga itself ensures proper cleanup is performed.

```
1  Cashier.moneyTransfer
2     (Account src, Account dest , int amt) {
3     name(self Thread.currentThread());
4     commit { terminate("ID" + self + "Step");
5              terminate(self); }
6     abort {  terminate("ID" + self + "Step");
7              terminate(self); }
8  }
```

4.1.2 Saga Steps

The code of all the steps of the saga is virtually identical, the only difference being the identification of the method corresponding to each step. We therefore only show the code for the `logTransfer` step. Each of these steps first require a reference to the top-level transaction so as to, second, add itself to the group of steps. By adding itself to the group of steps, it ensures that it will be terminated when the saga ends, by the code either in line 5 or 7.

```
9  Cashier.logTransfer
10    (Account src, Account dest , int amt) {
11    alias (Saga Thread.currentThread());
12    groupAdd(self "ID" + Saga + "Step");
13 }
```

This concludes the code for the structure concern of the sagas ATMS, applied to the bank transfer example. This code implements the structure of the saga in multiple steps, with termination of the steps when the saga ends. The following concern will add the handling of rollbacks of this structure, yielding the behavior of the Sagas ATMS.

4.2 Sagas: Rollback Handling

The second concern which we implement here, is rollback handling for the saga. Recall that in order to rollback a saga, the currently executing step is aborted, and that all committed steps are compensated for by executing compensating steps in the reverse sequence of step execution.

The KALA code below is an implementation of the above concern, and is defined in a separate KALA file. Again, we subdivide the discussion of the implementation in multiple parts, and the line numbers show this code all belongs to one file.

4.2.1 Saga Top-level

The top level of the saga registers itself, as in the structure concern, because the steps and compensating steps will place dependencies on the sagas, as we see later. At commit and abort time, the group of compensating steps is aborted, similar to what is performed in the structure concern.

```
1  Cashier.moneyTransfer
2     (Account src, Account dest , int amt) {
3     name(self Thread.currentThread());
4     commit { terminate("ID" + self + "Comp");
5              terminate(self); }
6     abort {  terminate("ID" + self + "Comp");
7              terminate(self); }
8  }
```

4.2.2 Last Step

In the last step of the saga, to implement the rollback concern, a number of dependencies have to be set between the step and the saga when the step begins. To set these dependencies, in lines 12 and 13, a reference to the saga has to be obtained, which is performed in line 11. `Saga ad self` forces the Saga to abort if this transaction aborts. `self wd Saga` states that if the Saga aborts before this transaction ends, it is also forced to abort. `Saga scd self` ensures that the Saga does not commit before this transaction has committed.

```
9  Cashier.logTransfer
10    (Account src, Account dest , int amt) {
11    alias (Saga Thread.currentThread());
12    begin { dep(Saga ad self); dep(self wd Saga);
13            dep(Saga scd self); }
14 }
```

4.2.3 First and Second Step

The first step of the saga needs to declare the compensating transaction used when the saga rollbacks. It achieves this by using an `autostart` statement in lines 18 thru 22, which compensates a bank transfer simply by performing the inverse transfer operation. The secondary transaction registers itself under a unique name in line 21, so that in lines 23 and 26 a reference can be obtained to this transaction to set the required dependencies. Also, the compensating transaction adds itself to the group of compensating transactions in line 22, ensuring it is properly terminated in line 4 or 6, when the saga ends. The dependencies on the compensating transaction ensure that it does not begin unless this transaction has committed (`Comp bcd self`), only begins if the saga aborts (`Comp bad Saga`) and disallow it to abort in that case (`Comp cmd Saga`)

```
15 Cashier.transfer
16    (Account src, Account dest , int amt) {
17    alias (Saga Thread.currentThread());
18    autostart (transfer
19       (Account src, Account dest, int amt)
20       (dest, src, amt) {
21          name(self "ID" + Saga + "Comp");
22          groupAdd(self "ID" + Saga + "Comp"); });
23    begin {  alias (Comp "ID" + Saga + "Comp");
24             dep(Saga ad self); dep(self wd Saga);
25             dep(Comp bcd self); }
26    commit { alias (Comp "ID" + Saga + "Comp");
27             dep(Comp cmd Saga);dep(Comp bad Saga);}
28 }
```

The second step of the saga is highly similar to the first step of the saga, the only differences being a different `autostart`, and dependencies being placed on the previous compensating transaction, as can be seen below. A reference to the compensating transaction of the first step of the saga is obtained in line 32. This allows the `CompPrev wcd Comp` dependency to be placed in line 42, ensuring that the previous compensating transaction begins after this has

ended. In other words, this determines the sequence in which the compensating transactions will run.

```
Cashier.printReceipt
   (Account src, Account dest, int amt) {
   alias (Saga Thread.currentThread());
   alias (CompPrev "ID"+Saga+"Comp");
   autostart (printTransferCancel
      (Account src, Account dest, int amt)
      (src, dest, amt) {
          name(self "ID" + Saga + "Comp");
          groupAdd(self "ID" + Saga + "Comp"); });
   begin {  alias (Comp "ID" + Saga + "Comp");
            dep(Saga ad self); dep(self wd Saga);
            dep(Comp bcd self); }
   commit { alias (Comp "ID" + Saga + "Comp");
            dep(CompPrev wcd Comp);
            dep(Comp cmd Saga);dep(Comp bad Saga);}
}
```

This completes the KALA code of the concern of rollback handling for the bank transfer operation using the sagas ATMS. As there are no more concerns in this ATMS, this concludes the KALA code for this example.

4.3 Conclusion

In this section we have shown how the implementation of a chosen ATMS for a given application is modularized using KALA code. In KALA, each module is implemented in a separate file. As an example we have shown a bank transfer operation that uses the Sagas ATMS. We have taken the decomposition of the Sagas ATMS, perfomed in Section 3.1, which identified two concerns, and given the KALA code for each of these concerns.

This modularization frees us from having to write tangled aspect code, which brings the benefits of separation of concerns to the process of defining an ATMS as an aspect. Instead of having to write aspect code which itself is tangled with multiple concerns, with all the impediments this entails, we now cleanly separate each concern in a separate KALA module.

5. Composing KALA Code

Given a definition of an ATMS concern in multiple modules, these need to be composed to form the complete KALA program. Conceptually, the concerns are combined before the ATMS aspect is woven, because it is the combination of these concerns that forms the complete definition of the ATMS. The KALA weaver, therefore, is not built to weave each concern of an ATMS separately into the base code. Instead all KALA modules are combined into one KALA file, describing how the ATMS is used, and this full description is woven into the base code[2].

Because of the domain-specific nature of KALA we were able to fully take into account the properties of the domain, yielding a composition mechanism that requires no programmer intervention.This is mainly due to the inherent composition properties of the ACTA model, which were taken into account when designing the KALA language. As a result, composition of multiple KALA modules is straightforward. In fact, composing multiple specifications in essence boils down to a simple merge, as we show here.

Conceptually, different KALA specifications declare that different actions need to take place at a given time in the life-cycle of a transaction: before the transaction begins, at begin time, at commit time or at abort time. In the composed file, therefore, for each of these moments in the life cycle all the actions defined for that point need to be performed. In other words, all the declarations that pertain to one moment in the life-cycle of the transaction have to be gathered into one block of the resulting specification.

The sequence of statements for naming and grouping within this composition matters, however, as an alias referred in a KALA statement needs to have been previously looked up. Therefore, when composing multiple KALA specifications, the partial ordering of naming and grouping statements within each KALA file needs to be preserved in the global file.

Considering in more detail the `begin`, `commit` and `abort` blocks of KALA code, we can state that the sequence of the code for setting dependencies, placing views, performing delegation and termination, however, is irrelevant. This is because, as said in Section 2.2, these are considered to happen in the same atomic action of begin, commit or abort. Therefore, when composing a number of `begin`, `commit` or `abort` blocks for the same method, their dependency, view, delegation and terminate statements can be simply joined into one sequence which respects the partial ordering of names and groups. The same observation holds for `autostart` statements, as their sequence in the KALA code also is of no importance. All `autostart` statements for one method are placed before the begin block of the composed KALA specification.

We can implement the above composition by a simple merge, the implementation of which is outlined next. Given that we have a number of KALA specifications for one method and we need to generate an output file:

1. Start the output file with the method signature suffixed with {.
2. For each specification, take the sequence of top-level declarations and add them to the output file.
3. Write the start of a `begin` block to the output file.
4. For each specification take the sequence of begin declarations and add them to the output file.
5. Write the close of the `begin` block, and the start of the `commit` block to the output file.
6. For each specification take the sequence of commit declarations and add them to the output file.
7. Write the close of the `commit` block, and the start of the `abort` block to the output file.
8. For each specification take the sequence of abort declarations and add them to the output file.
9. Write the close of the `abort` block and the closing } to the output file.

There is one downside, however, to this simple merging, which is name clashes: multiple modules should not define the same names. If these modules redefine the name with the same target, as in line 3 of both modules in the sagas example in Section 4, this is not an issue. But if multiple modules define the same name for a different target this will lead to wrongly placed dependencies, views, delegation, and so on, yielding faulty code. Conceptually, this issue can, however, be easily solved through a renaming or a merge of names. Therefore we do not provide an outline of such an implementation here.

The above is all which is required to compose multiple KALA modules into one full program. Thanks to the properties of the domain, which were taken into account when designing KALA, we have a straightforward composition mechanism that requires no programmer intervention.

[2] Although this aspect code will be tangled aspect code, this is not an issue since this code is but an intermediate representation which is not presented to a programmer.

6. Building a New ATMS: Cooperating Nested Transactions

In this section we show how we can use KALA to define a new ATMS to fit a given application or class of applications. The goal is to achieve an ATMS in which the transactional properties better align with the transactional properties of the (class of) application(s). We show this by creating a new ATMS, which we call Cooperating Nested Transactions, that aims to achieve the highest possible performance for computations that are hierarchically structured.

Before we introduce Cooperating Nested Transactions, we first give a definition of the Nested Transactions ATMS in multiple KALA modules. Second, we show how we can easily modify this definition to yield the Cooperating Nested Transactions ATMS.

In this section, we do not provide example applications to which the KALA code is applied. This is because as we solely wish to concentrate on the implementation of the ATMS, without considering how this ATMS is used by an application. We will use placeholder code, which is marked like this, when referring to base-level entities, such as method signatures. When these ATMS are used for a given application, this placeholder code needs to be replaced by the appropriate code for that application.

6.1 Nested Transactions

In Section 3.1 we established that Nested Transactions is composed out of four different concerns: structure, handling of rollbacks, view management and delegation of operations. We now write KALA code for each of these concerns separately.

6.1.1 Structure

The structure of Nested Transactions is not fixed statically as in Sagas, instead of this, at runtime a tree structure of transactions is built. Each transaction that forms a part of the tree structure is solely responsible for itself. Given such a tree structure, built at runtime, there is however one restriction: a parent may not commit before all its children have ended. Therefore a commit dependency `cd` needs to be placed between a parent and each of its children. This requires that each child obtain a reference to its parent before placing this dependency, as shown in line 4 of the code below. We achieve this by first letting each transaction name itself (line 2), so that it can be referred to by its children, and second letting each transaction obtain a reference to its parent by performing a lookup in line 3.

```
1 packageName.className.methodName(parameterList) {
2    name(self name expression);
3    alias(parent parent expression);
4    begin {  dep(parent cd self); }
5    commit { terminate(self); }
6    abort {  terminate(self); }
7 }
```

6.1.2 Rollback Handling

When rolling back a transaction which is a part of a tree of nested transactions we need to ensure that if this transaction aborts, all its children also abort. This is implemented first by letting each child add itself to a group associated with the parent in line 4 of the code below, and second by letting each transaction terminate its children when aborting, in line 6 of the code below. Having each child add itself to the group associated with the parent, however, also implies that each parent needs to also clean up this group when committing, which is performed in line 5.

```
1 packageName.className.methodName(parameterList) {
2    name(self name expression);
3    alias(parent parent expression);
4    groupAdd(self "ID" + parent + "Children");
5    commit { terminate("ID" + self + "Children"); }
6    abort {  terminate("ID" + self + "Children"); }
7 }
```

6.1.3 Delegation

Upon commit of a child its work is delegated to the parent, which is performed in line 4 of the KALA code below. Again this requires a reference to the parent, which in turn requires that each transaction register itself.

```
1 packageName.className.methodName(parameterList) {
2    name(self name expression);
3    alias(parent parent expression);
4    commit { del(self parent); }
5 }
```

6.1.4 View Management

Thirdly, in Nested Transactions, a child has a view on the intermediate results of its parent, which is achieved by setting the view at begin time in line 4 of the code below.

```
1 packageName.className.methodName(parameterList) {
2    name(self name expression);
3    alias(parent parent expression);
4    begin { view(self parent); }
5 }
```

This concludes the definition of the Nested Transactions ATMS. We now show how we can straightforwardly modify this ATMS to better fit a particular class of applications.

6.2 Cooperating Nested Transactions

One of the advantages of using the multi-tiered architecture in a large-scale distributed system is the ability of this architecture to provide a faster response time of the middle tier through load balancing. We can use parallelization on multiple servers to perform sub-computations of a given algorithm in parallel, but we want the entire computation to be performed as a single transaction to prevent data inconsistency. In a hierarchically structured computation, we can have sub-computations as nested sub-transactions of the main algorithm, and distribute sub-transactions over multiple servers, to be performed in parallel.

We can consider using Nested Transactions as an ATMS for this application: as sub-computations are sub-transactions they will preserve data consistency, and can access the data of the parent. Also, a failure in the sub-computation will not necessarily imply that the entire computation is lost. This allows graceful recovery of errors in the computation, without needlessly losing work. Having sub-computations performed in parallel, however, may entail that each of these sub-computations needs to be able to access the other computations' intermediate results, as they are supposed to cooperate, in parallel, to achieve the overall goal. This is not possible when using nested transactions and therefore, we have adapted the Nested Transactions ATMS to allow sharing between multiple sub-transactions, yielding a new ATMS: *Cooperating Nested Transactions* (CNT).

In CNT, all siblings of a parent transaction have access to each other's intermediate results though a view relationship. This, however, has an impact when aborting a child transaction. The siblings which have seen the inconsistent data of this child and have not committed are also considered to be inconsistent and should abort. This only applies to the sibling transactions that run simultaneously, in parallel, with the aborting sub-transaction.

Siblings that have committed before the aborter are not aborted, and siblings that start after the abortion need not abort. This limits the lost work in such cases to only include sibling sub-transactions which run at the same time as the aborting sub-transactions. This is an advantage of using CNT over running the entire computation in one transaction. If we would do this and a sub-computation aborts, automatically all of the work of the entire computation would be lost. With CNT, only the work of the sub-computations simultaneously running is lost.

We have implemented CNT in KALA by taking the implementation of Nested Transactions and modifying the concerns of view management and rollback handling. This illustrates one of the benefits of applying separation of concerns at the level of the ATMS definition, easing modification of an ATMS as only the code for the changing concerns needs to be considered, as we show next.

6.2.1 View Management

In CNT, all children of a given transaction can see each other's intermediate results. To implement this, in the code below, views are set from this transaction to all siblings, and the reverse, in line 7. This, however, requires each child of a transaction to add itself to the group of children of the parent, performed in line 4, and that a reference to be obtained to this group, in line 5. Also, when a transaction ends, the group of children of this transaction has to be removed by the system, which is performed in lines 8 and 9 of the code below.

```
1  packageName.className.methodName(parameterList) {
2     name(self name expression);
3     alias(parent extends expression);
4     groupAdd(self "ID" + parent + "Children");
5     alias(siblings "ID" + parent + "Children");
6     begin {  view(self parent);
7        view(self siblings); view(siblings self); }
8     commit { terminate("ID" + self + "Children"); }
9     abort {  terminate("ID" + self + "Children"); }
10 }
```

6.2.2 Rollback Handling

When performing rollback, siblings of the erroneous transaction should also abort, as they have seen the intermediate state of the aborting transaction. We cannot modify transactions that have already committed, but we can abort all currently running siblings of the aborting transaction, which is performed by the `terminate` statement in line 7.

```
1  packageName.className.methodName(parameterList) {
2     name(self name expression);
3     alias(parent parent expression);
4     groupAdd(self "ID" + parent + "Children");
5     commit { terminate("ID" + self +  "Children"); }
6     abort {  terminate("ID" + self + "Children");
7              terminate("ID" + parent +"Children"); }
8  }
```

7. Conclusion

KALA was designed to enable the modular specification of ATMS, avoiding the need to write tangled aspect code. In this paper we introduced how KALA enables the application of separation of concerns in the process of defining an ATMS.

In KALA, each concern can straightforwardly be written in a separate module and the composition does not require any programmer intervention. This is thanks to the domain-specific nature of KALA, where the properties of the domain were extensively taken into account when defining the modularization and composition mechanism.

We have shown how two existing ATMS can be programmed as KALA modules, namely Nested Transactions and Sagas. Furthermore, we described how a new ATMS: Cooperating Nested Transactions was created by modifying a number of modules from an existing ATMS, in this case Nested Transactions. This shows the benefit of modularization in KALA code, *i.e.* applying separation of concerns when defining an ATMS.

Acknowledgments

Thanks to Denis Conan for fruitful discussions when considering the topic of Tangled Aspect Code and thanks to Theo D'Hondt for supporting this research.

References

[asp06] The AspectJ project, 2006. http://eclipse.org/aspectj/.

[CR91] 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.

[Fab05] Johan Fabry. *Modularizing Advanced Transaction Management - Tackling Tangled Aspect Code*. PhD thesis, Vrije Universiteit Brussel, Vakgroep Informatica, Laboratorium voor Programmeerkunde (PROG), July 2005.

[FD06] 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.

[GMS87] Hector Garcia-Molina and Kenneth Salem. Sagas. In *Proceedings of the ACM SIGMOD Annual Conference on Management of data*, pages 249 – 259, 1987.

[HVL95] Walter L. Hürsh and Cristina Videira Lopes. Separation of concerns. Technical report, College of Computer Science, Northeastern University, 1995.

[KG02] Jörg Kienzle and Rachid Guerraoui. AOP: Does it make sense? - the case of concurrency and failures. In *Proceedings of ECOOP 2002*. Springer Verlag, 2002.

[Mos81] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, Massachusetts Institute of Technology, 1981.

[Par72] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[RC03] Awais Rashid and Ruzanna Chitchyan. Persistence as an aspect. In *2nd International Conference on Aspect-Oriented Software Development*. ACM, 2003.

[SLB02] Sérgio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of OOPSLA 02*. ACM, 2002.

[TOHJ99] Peri L. Tarr, Harold Ossher, William H. Harrison, and Stanley M. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, 1999.