ASPECT-ORIENTED DOMAIN SPECIFIC LANGUAGES FOR ADVANCED TRANSACTION MANAGEMENT

Johan Fabry
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium

Thomas Cleenewerck
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium

Keywords: Transaction Support, Software Engineering, Aspect-Oriented Programming, Domain-Specific Languages.

Abstract: Transaction management has some known drawbacks, which have been researched in the past, and many solutions in the form of advanced transaction models have been proposed. However, these models are too difficult to be used by the application programmer because of their complexity and their lack of separation of concerns. In this paper we address this by letting the application programmer specify advanced transactions at a much higher abstraction level. To achieve this, we use the software engineering techniques of Aspect Oriented Programming and Domain-Specific Languages. This allows the programmer to declare advanced transactions separately in a concise specification which is much more straightforward.

1 INTRODUCTION

In current-day module-based software development, software is built by decomposing the problem domain into different modules. However, given any decomposition of a sufficiently large problem domain, not all of its concerns can be modeled in separate modules. Some concerns will be scattered throughout the entire system, crosscutting different modules (Tarr et al., 1999). Transaction management is one of those concerns because to use transactions, the programmer must add transaction demarcation code: code which starts and ends transactions, and which indicates which data accesses occur within the scope of what transaction. Code for transaction demarcation is therefore not just contained in one module. Instead, transaction demarcation code is spread out into different modules.

Aspect-Oriented Programming (AOP) addresses the issue of such cross-cutting concerns (Kiczales et al., 1997) by introducing the concept of *aspects*. In AOP a concern is either implemented in a component, or in an aspect. A concern is called a component if it can be encapsulated cleanly into a module, and if it cannot be encapsulated cleanly into a module, it is called an aspect. (All concerns which are components are often called the base aspect or base code.) AOP allows the programmer not only to reason separately about the aspects, but also to specify them separate from the normal modules. This is achieved by specifying the code pertaining to an aspect in a separate aspect file, in a special aspect language. Once all modules and aspects are defined, a special tool, called an *aspect weaver*, combines these into executable code.

Proposed approaches that captured transaction management with aspects all elaborated in an object-oriented setting and start by assuming that transaction boundaries coincide with method boundaries, i.e. that transactions are started when a method starts executing and ended when the method ends. However, these approaches are limited, as they structurally restrict themselves to only implement classical transaction processing.

A known issue with classical transactions is that it has been developed to treat small units of work, which only access a few data items. As a result, as transaction time grows, and the number of data items accessed in one transaction becomes larger, the performance of the system will drop significantly (Gray and Reuter, 1993). To increase application performance, and to address some additional requirements, advanced transaction mechanisms (ATMS) have been developed, mostly between 1981 and 1997. An impressive number of alternate ATMS can be found in the literature, and two books have been published about the subject (Elmagarmid, 1992; Jajodia and Kershberg, 1997).

Our research focuses on bringing the known ad-

Fabry J. and Cleenewerck T. (2005).
ASPECT-ORIENTED DOMAIN SPECIFIC LANGUAGES FOR ADVANCED TRANSACTION MANAGEMENT.
In *Proceedings of the Seventh International Conference on Enterprise Information Systems*, pages 428-432
DOI: 10.5220/0002526404280432

vantages of separation of concerns through AOP to the field of ATMS. The goal is to be able to express the transactional properties of the application in a separate aspect, in domain-specific aspect languages. Such a separate specification then allows us to tailor the aspect-specific languages specially to the domain of the problem at hand, in this case the ATMS being used. As a result, the application programmer writes more concise and intentional aspect declarations, i.e. transactional specifications.

In this paper we start with an introduction to ATMS, then briefly discuss a published formal model for them. We then detail our contribution by first presenting the general aspect language we defined for ATMS: KALA, and second giving an overview of the different aspect languages for specific ATMS.

2 ADVANCED TRANSACTION MODELS

As said above, a large number of alternate ATMS can be found in the literature, therefore we do not provide a full overview here. Instead we briefly describe two models here: Nested Transactions and Sagas.

A first example is the best-known ATMS: *nested transactions* (Moss, 1981). Nested transactions allow for hierarchically structured transactions, where a child transaction also has access to the data used by its parent transaction, and this recursively to the root transaction. Furthermore, when a child transaction commits its data, this is not committed to the database, but instead to its parent, which is now responsible for committing this data.

A second example ATMS, *Sagas* (Garcia-Molina and Salem, 1987), is tailored towards long-lived transactions. Sagas split these into a sequence of atomic sub-transactions, which should either be executed completely or not at all. Splitting the long-term transactions releases locks earlier, which increases concurrency. However, to rollback the Saga extra work needs to be done: compensating actions must be executed to undo the effects of already committed sub-transactions. Hence, the application programmer should define a compensating transaction for each sub-transaction, which performs a semantical compensation action. To rollback a Saga, the TP Monitor aborts the currently running sub-transaction, and subsequently runs all required compensating transactions in reverse order.

Besides the models we discussed above, we have evaluated a large set of ATMS, and have concluded that each ATMS usually focuses on a small subset of the issues of classical transaction management. No overall system has been developed which treats a large number of the identified shortcomings of classical transactions. However, a formal model that covers many ATMS has been developed, that can serve as the basis for an overall system, and we discuss this next.

3 FORMALIZING ATMS WITH ACTA

The *ACTA* formalism (Elmagarmid, 1992) was created as a common framework in which it is possible to specify different ATMS. We will now give an overview of ACTA, and use nested transactions as a running example of an advanced model.

In ACTA, an ATMS is formally defined by stating three kinds of axioms that constrain and modify the transaction history. This transaction history, as defined in ACTA, consists of a sequence of operations of the different transactions on the database.

The first kind of axiom on transaction histories are *dependencies*: a dependency places a relationship between two transactions, defined in terms of the operations of these transactions. An 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 $(Tp\ \mathcal{CD}\ Tc)$.

The second kind of axioms defined by ACTA are *view* definitions, which allow different transactions to concurrently work on the same data as if they were the same transaction. In the example this is specified by defining the view of Tc to contain Tp.

The third kind of axioms are *delegation*: one transaction Ti delegates the responsibility for committing or aborting a specified number of its operations to another transaction Tj. 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, have been published. These descriptions are quite complicated and lengthy: for example, nested transactions is defined using nineteen axioms, which are not that straightforward, as can be seen in (Elmagarmid, 1992).

4 FROM ACTA TO ASPECT: THE LANGUAGE KALA

Having the ACTA formal model at our disposal is an important asset: we can use this model as a guideline for the implementation of a TP Monitor and for the definition of the aspect language for advanced transaction models. This will then allow us to support a wide variety of advanced transaction models. As a full discussion of the TP Monitor is outside of the

scope of this paper, we will instead focus here on the aspect language for advanced transaction models.

Our aspect language, called KALA (**K**ernel **A**spect **L**anguage for **A**TMS), which, together with the corresponding aspect weaver, allows for the specification of transactional properties of Java methods. These specifications are highly similar, but not identical to the ACTA specifications of transaction models. In KALA, a programmer declaratively annotates the transactional properties of a method, primarily using the concepts which are defined in ACTA: dependencies, view, and delegation. Additionally, two extra concepts are required in KALA: naming of transactions and life-cycle management. We discuss these before treating dependencies, view and delegation.

4.1 Naming

To be able to refer to transactions within a KALA program, two naming schemes are required: first a static scheme for compile-time naming and second a dynamic scheme for run-time naming.

In the static scheme, a transaction coincides with a method definition, and is therefore named by giving the full class name and the method signature, separated by a dot. This already allows us to write our first KALA program, below, in which we declare the method with signature `increment(int)` of the class `utility.Counter` transactional.

```
utility.Counter.increment(int) {}
```

Within one transaction specification, the programmer needs to be able to refer to other transactions that will run concurrently to be able to define dependencies, views and delegation relations between them (as defined in 3). To allow this, we also provide a dynamic naming scheme, at runtime. When a transactional method starts, it can register itself globally under a certain name, and all running transactional methods can use this name to refer to it. This is achieved in KALA by adding a `name` statement to the transactional properties, which binds a transaction identifier to a key (a Java expression enclosed between angular brackets). Lookup is performed, by using an `alias` statement. In such a statement, an expression referring to a dynamic name is resolved to an already running transactional method.

A reserved name is the pseudo-name `self`, which always refers to the current transaction.

As an example of naming, suppose our counter object has a unique identifier (contained in the instance variable `ident`) and we wish to register the increment transactional method using this name and the current count (contained in the instance variable `count`). This is performed by the following KALA program:

```
utility.Counter.increment(int) {
  name (self <this.ident.toString() + this.count>);}
```

4.2 Transaction Life-Cycle Management

Transactions are started and ended by the underlying application as it executes. However, ACTA, as a formal model, does not involve itself with this. ACTA restricts itself to verifying if the resulting transaction history complies to the axioms. In going from formal model to an implementation, we have to consider how we can ensure that transactions are started and ended when necessary.

As transactions coincide with methods, the main part of the responsibility for transaction life-cycle management lies in the control flow of the underlying application: at some points a transactional method will be called, resulting in the beginning of a transaction, and when the method ends, the transaction will also end. However, in addition to the transactions started by the application, some models, require *secondary transactions* to automatically run when their dependencies are satisfied. In Sagas, for example, these are compensating transactions which run at rollback, i.e. these methods are executed at that time.

To support this, we have added the option to start secondary transactions automatically through the use of the `autostart` statement. Combining such an autostart with the required dependency statements, which we will see below, will ensure that the automatic transactions run when required. With the `autostart` statement a transactional method will be called which will run in a new thread, with a set of given transactional properties.

A second life-cycle operation is the termination of transactions. This is required because due to the complex interactions of different transactions in ATMS, when a transaction ends its name and dependencies may still be required. When these are no longer needed, the transaction can be stopped and removed from the system by means of the `terminate` statement.

4.3 Dependencies, Views and Delegation

Dependencies, views and delegation are concepts of ACTA which are represented as statements in KALA. These statements can be performed at begin, commit or abort time and therefore are grouped in `begin`, `commit` or `abort` blocks, as can be seen in the example describing a nested transaction below:

```
AClass.nestedMethod() {
 alias(parent < [...] > )
 begin  { dep(self wd parent, parent cd self);
```

```
        view(self, parent) }
commit { del(self, parent) } }
```

In this example, we first use the `alias` statement to lookup the parent transaction and bind it to the variable `parent`. We have omitted the expression that determines the parents' name, for brevity.

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. Also in the `begin` block, the `view` statement declares that the child views the operations of the parent. Lastly, in the `commit` block, the `del` statement, performs delegation from child to parent transaction.

5 A FAMILY OF DOMAIN-SPECIFIC LANGUAGES

KALA is a powerful language consisting of a set of assembly-like language constructs, based on the minimal set of basic building blocks of the ACTA formal model. The programs in KALA describing ATMS specify the inner-workings of these ATMS, as we have seen in 4.3. Clearly this level of complexity is not suitable for the application engineer as he should in fact only reason in terms of the abstractions offered by an ATMS.

To raise this level of abstraction to the level of concepts present in the ATMS, we have built a number of Domain Specific Languages, each language specific to an ATMS, while obtaining a high degree of reuse between the different language implementations. We will now discuss these languages, before detailing their implementation.

5.1 Domain-specific languages

Domain-specific languages are little languages which are specifically designed to express applications in a particular domain. This entails that the language constructs of a DSL directly reflect the concepts of the domain and hide the DSL programmer from non-domain-specific technical issues.As a result, using a DSL shields the application engineer from the technical details involving the inner-workings of the ATMS he is currently using.

For each ATM we constructed a specific DSL that reflects the abstractions used in that model. We have currently implemented four DSLs, respectively for classical transactions, nested transactions, Sagas and RCS (which is an ATMS we did not discuss here). Below an example is given of a specification in the nested transaction language:

```
AClass.nestedMethod() extends < [...] >;
```

In this example, the transaction corresponding to the method `AClass.nestedMethod()` is declared to be a child of another transaction, by using the `extends` keyword. (As in 4.3 we have omitted the Java code that determines the parents' name)

At compile-time a DSL takes a domain-specific program, such as the one above, and generates the equivalent KALA program. In the above case, the result is the KALA program in section 4.3.

A more complicated example is the DSL which was constructed to describe Sagas. An example of such a Saga specification is given below. It is a money transfer method which is divided into three steps, each a method which is called by the transfer method. In this code we declare the Saga and its steps. Also, each step, except for the last, defines a compensation step. Compensation steps are defined through the use of the `comp` keyword, which takes as argument a static name, as defined in 4.1, and a list of actual parameters for that method.

```
Bank.moneyTransfer(Account, Account, int) {
  Bank.transfer(Account, Account, int)
    compensate Bank.transfer(Account, Account, int)
      <dest, source, amount>;
  Bank.printReceipt(Account, Account, int)
    compensate Bank.printRet(Account, Account, int)
      <source, dest, amount>;
  Bank.logTransfer(Account, Account, int); }
```

As an illustration of the advantage of using a DSL, we include the KALA program for the second method only. We can not include the full KALA program due to lack of space. Suffice it to say that the full program is over four times the size of the code shown here, and has the same complexity.

```
Bank.printReceipt(Account, Account, int) {
   alias (Saga <Thread.currentThread()>);
   alias (CompPrev <""+Saga+"Comp1">);
   name (self <""+Saga+"Step2>);
   autostart (Bank.printRet(Account, Account, int)
      <source, dest, amount> {
         name(self <""+Saga+"Comp2">); });
   begin { alias (Comp <""+Saga+"Comp2">);
      dep(Saga ad self, self wd Saga,
         Comp bcd self); }
   commit { alias (Comp <""+Saga+"Comp2">);
      dep(CompPrev wcd Comp, Comp cmd Saga,
         Comp bad Saga); }}
```

In this code, we see how this step obtains references to the saga and to other transactions and names itself, before defining a compensating transaction. At begin and commit time, dependencies are placed between these, to ensure correct compensation when required.

It is clear that the KALA code above is more complex than the definition in the saga language, while in effect describing the same thing. This clearly shows how the use of a DSL has raised the level of abstraction to the level of the concepts available in the ATMS, further reducing complexity of usage.

5.2 Open set of language features

Instead of implementing a DSL compiler for each ATM from scratch, we have chosen for a modular approach that can exploit the common language features of the different DSLs. The division of a ATM specific DSL into modular and reusable language features is hampered because the effects of these language features are scattered in the resulting KALA program. For example, take the four basic language features of the Saga DSL: `name` (denoting the step name), `saga`, `step` and `comp`, which should correspond with four modules. As we have seen in 5.1, each step results in a single KALA transaction.

When modularizing the saga language definition, the step and its compensation description are separated into two modules, each module yielding a partial transaction description. When the two language features are combined into single language, the two partial descriptions must be combined together to a single KALA transaction. However, the name of a compensating transaction is not confined to just one step, but it is present in different steps, i.e. each step is linked to other steps. This entails that the `comp` module needs to invasively change different `step`s with additional dependencies for the compensating transactions, which hampers the division of the DSL into reusable modules. In order to separate these different language features into reusable modules, a system supporting such invasive composition is required.

Keeping both the modularity and invasive composition requirements in mind, the Linglet Transformation System (LTS) (Cleenewerck, 2003) seemed the most appropriate system. In LTS a language implementation is divided into smaller reusable components, each defining a small language feature. The languages in LTS are constructed by composing these modular and reusable language components (called linglets) together. In LTS, for each of the four basic language features of the Saga DSL (`name`, `saga`, `step` and `comp`) a linglet is built. With these four language features the Saga DSL can be specified through composition. This composition is achieved by customizing the linglet definitions with other linglets. Also, these linglets have been reused when building the RCS DSL, which is an extension of the Saga DSL.

In total, we have built four DSLs: one for classical transactions, and three for the ATMS mentioned here, of which we have shown two, out of a set of seven different language components.

6 CONCLUSIONS

Advanced transaction models address shortcomings in the classical transaction model, but are difficult to use for the application programmer due to their complexity and the lack of separation of concerns. Application programmers need detailed knowledge of the advanced transaction models to use them in their software and must spread the configuration information for these models throughout the code.

In this paper, we reduced this complexity by providing a open family of domain-specific languages, each of which tailored for one specific ATMS. These languages allow the application programmers to declaratively state the transactional properties at a high level of abstraction, in a specification separate from the base application.

To reach this goal, we used Domain-specific languages and Aspect-Oriented Programming. As a result, two contributions have been made: the implementation of an aspect language, KALA, to express advanced transaction models and the development of an open set of language features to raise the abstraction level of KALA to concepts found in ATMS.

ACKNOWLEDGMENTS

We thank Kris Gijbels and Wolfgang De Meuter for their valuable feedback when proof-reading, and Theo D'Hondt for supporting this research.

REFERENCES

Cleenewerck, T. (2003). Component-based dsl development. In *Proceedings of GPCE'03 Conference, LNCS 2830*, pages 245–264. Springer-Verlag.

Elmagarmid, A. K., editor (1992). *Database Transaction Models For Advanced Applications*. Morgan Kaufmann.

Garcia-Molina, H. and Salem, K. (1987). Sagas. In *Proceedings of the ACM SIGMOD Annual Conference on Management of data*, pages 249 – 259.

Gray, J. and Reuter, A. (1993). *Transaction Processing, Concepts and Techniques*. Morgan Kaufmann.

Jajodia, S. and Kershberg, L., editors (1997). *Advanced Transaction Models and Architectures*. Kluwer.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *Proceedings of ECOOP 1997*. Springer Verlag.

Moss, E. B. (1981). Nested transactions: An approach to reliable distributed computing. Technical report.

Tarr, P. L., Ossher, H., Harrison, W. H., and Jr., S. M. S. (1999). N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119.