ASPECT-ORIENTATION IN MODELLING: LESSONS LEARNED

Ella Roubtsova
Open University of the Netherlands, the Netherlands
Munich University of Applied Science, Germany
ella.roubtsova@ou.nl
ella.roubtsova@hm.edu

Keywords: Separation of concerns, Behaviour models, Semantics, Composition techniques.

Abstract: Model-driven software engineering community faces the problems related to the growing complexity of system models and their rapid evolution. These problems are similar to the problems of programming. Driven by the ideas of Aspect-Oriented Programming many modelling techniques were revised in attempts to find their ways to deal with model complexity and evolution. This paper presents an analysis of existing semantic groups of modelling approaches, their goals, pros and cons, advantages and restrictions. It is aimed to show the deep reason of the fact that particular composition semantics combined with the elegant idea of aspect weaving leads to heavy verification procedures after any change in the AOM model. This analysis reveals the elements of modelling semantics that simplify the verification procedures in AOM.

1 INTRODUCTION

Model-driven software engineering community faces the problems related to the growing complexity of system models and their rapid evolution. The analysts (Chwif et al., 2000) name many reasons for model complexity starting from the “show off” factor, the “possibility” to execute complex model on powerful computers and ending with the “lack of understanding of the real system” and “unclear modelling objectives”. Although it is universally acknowledged that the simplest model is the most beautiful one (Robinson, 1994), there are inescapable reasons for model complexity. The modelled system covers several interfering and non-localizable problem frames (Jackson, 2005) and the validity and completeness of the model may depend on the coverage of relevant problem frames.

The non-localizable problem frames in models are often called aspects following the terminology of Aspect-Oriented Programming (AOP). The modelling approaches that have the goal to separate aspects in models and then compose them into the complete model or produce the behaviour of the complete model, form the research area called Aspect-Oriented Modelling (AOM).

This paper presents a survey of existing semantic groups of aspect-oriented approaches to identify and explain their goals, abilities and restrictions. There are many AOM surveys, for example (Reina et al., 2004), (Chitchyan et al., 2005), (Op de beeck et al., 2006), (Reddy et al., 2006) and (Schauerhuber et al., 2007). These surveys mostly analyze the UML-based approaches. The deviations of semantics is not the point of attention. The criteria of comparison are often of practical nature such as the position of the approach within the full life cycle, evaluation of granularity and scalability. The aim of this paper is different. It is aimed to show the deep reason of the fact that particular composition semantics combined with the elegant idea of aspect weaving leads to heavy verification procedures after any change in the AOM model. This analysis reveals the elements of modelling semantics that simplify the verification procedures in AOM.

The reminder of the paper is the following. Section 2 briefly presents the achievements of AOP. Section 3 observes the semantic groups of Aspect-Oriented Modelling approaches on the basis of the used composition techniques and explains the applicability and restrictions, PROS and CONS of those groups of AOM approaches. Section 4 draws conclusions and identifies the semantic elements that provide the necessary flexibility and may result in a breakthrough in the line of heavy design approaches.

Roubtsova E.
ASPECT-ORIENTATION IN MODELLING: LESSONS LEARNED.
DOI: 10.5220/0004458100130024
In *Proceedings of the First International Symposium on Business Modeling and Software Design (BMSD 2011)*, pages 13-24
ISBN: 978-989-8425-68-3

2 ACHIEVEMENTS OF AOP

It was recognized by (Cardone and Lin, 2004) that the difficulties with separation of concerns in AOP were caused by the composition support of the mainstream object-oriented programming languages that “restricted to single inheritance and subtype polymorphism”. For example, there are three possible ways to organize two features *A* and *B*. into classes: (1) put them in the same class, (2) make *A* a subclass of *B*, (3) make *B* a subclass of *A*. The first two choices force *B* to be included whenever *A* is included. By forcing to choose a single fixed hierarchy orthogonal composition of features is restricted. “As the number of features grows, this problem becomes more severe, because the number of possible feature combinations grows rapidly, but the number of feature combinations that can be practically supported does not”.

In order to solve this problem, Aspect-Oriented Programming (AOP) invented a modular unit called *aspect* to implement a crosscutting concern. This unit contains an *advice* in form of a code presenting a concern and *pointcut designators* being the instructions on where, when and how to invoke the advice. The well defined places in the structure of a program where an advice should be attached were named *join points*. The join point model, using method calls as join points, were generally accepted (Filman et al., 2004) and implemented as extensions of programming languages. Such extensions gave a new task for compilers to produce the code with hard woven aspects. Another branch of AOP developed middleware able to fulfil run-time aspect composition without producing the code of the complete program.

The base programs used for weaving aspects are designed to be oblivious (Filman and Friedman, 2004) to aspects. The advantage of this feature is that ”Concerns are separated not only in the structure of the program but also in heads of their creators”. The disadvantage is that the systems ”melded from separate minds” can only function if the properties of aspects are preserved in the composed system. This preservation has not been achieved in AOP, although this is an active research area (Kiczales and Mezini, 2005). Current AOP techniques allow producing spectative, regulative and invasive categories of aspect (Katz, 2006). Spectative aspect can change the values of variables local to the aspect, but can’t change the value of any variable of other aspects and the flow of method calls. Regulative aspects affect the flow of control by restricting operations, delaying, or preventing the continuation of a computation. Invasive aspects change the values of variables in the underlying program. In case of invasive aspects no guaranty can be given about preserving of dynamic properties of the underlying program.

3 AOM AND MODELLING SEMANTICS

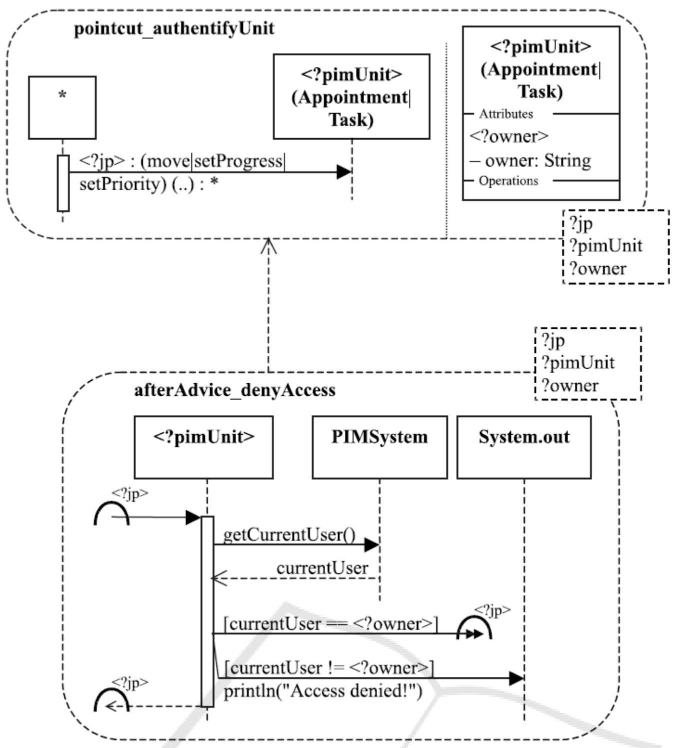
The complexity problems caused by the non-localizable overlapping concerns was identified in AOP and AOM was trying to map the findings of AOP on models and understand the borders of application of the AOP solutions. The AOM started with modelling and analysis of AOP programs. The next step was to evaluate the ability of existing modelling techniques to capture aspects and compose them. This research activity resulted in two streams of approaches. The dominant stream localizes concerns at the modelling level but do not simplify model analysis. There are approaches that also localize reasoning on concern models about the behaviour of the whole model.

3.1 Revision of Sequence Diagrams

The sequence diagrams were the first group of modelling notations revised for the AOM purposes.

A set of sequence diagrams with conventional semantics is aimed to present only a part of possible sequences of system behaviour. Sequence diagrams are mostly used to illustrate behaviour of programs and therefore use operation calls and returns as elements of behaviour. The composition techniques used in sequence diagrams are restricted to sequential composition, alternatives, cycles and insertion of sequences.

For the AOM needs the conventional sequence diagrams were extended with Join Point Designation Diagrams (JPDD) (Stein et al., 2005). An advice behaviour and the join points of this advice are specified with different sequence diagrams. JPDDs are modelling means to graphically represent join point queries on sequence diagrams. Figure 1 borrowed from (Stein and Hanenberg, 2006) shows the sequence diagrams presenting a join point and an advice-sequence of an aspect “Access control policy” that prescribes that the owner management data are used to ensure that particular units (i.e. tasks, contacts, or appointments) are modified or viewed by their proper owner. The advice presented with the diagram $afterAdvice\_DenyAccess$ is executed when one of the join points specified with wildcards $move|setProgress|setPriority$ is called. The information about the current user is obtained $getCurrentUser()$ and compared with the information about the owner of the item. If the current



```
aspect Authorization(){
    pointcut restrictAccess(PIMUnit pimUnit):
        (call(* Appointment.move(..)) ||
         call(* Task.setProgress(..)) ||
         call(* Task.setPriority(..))) && target(pimUnit) &&
          if(!pimUnit.getOwner().equals(PIMSystem.getCurrentUser())
                    || (PIMSystem.getCurrentUser()== null));

      void around(PIMUnit pimUnit) : restrictAccess(pimUnit) {
         System.out.println("Access Denied!") ;
}
```

Figure 1: JPDD model.

user is not the owner $currentUser! - <?owner>$ then the access is denied.

The sequence diagrams with JPDDs imitate aspect-oriented programs. The composition of sequences presenting the base behaviour, advice and join points is implemented as a compiler task. As the sequence diagrams are close to the notation of aspect-oriented languages, the code is generated from them. As sequence diagrams present partial behaviour of the system, then result of their composition is a partial code.

- PROS: The AOM version of sequence diagrams with JPDDs supports more compact presentation of aspects than the conventional sequence diagrams. Models of join points become reusable artefacts. Ability to present aspects with sequence diagrams eases evolution of sequence diagrams.
- CONS: Understanding of models and reasoning on models presented with AOM sequence diagrams with JPDDs become more difficult than understanding and reasoning with plain sequences diagrams. It is mostly because the weaving of aspects should be done in human heads. However, with tool support, the aspects are woven into base sequences and presented to the designers for understanding and linear reasoning.

Although sequence diagrams are found to be very intuitive means for behaviour modelling, their main issues still remain. Namely, even in combination with class diagrams sequence diagrams are not able to present complete behaviour of complex business systems having infinite set of traces. Moreover, sequence diagrams with JPDDs use the same composition techniques as the AOP programs and cannot prevent specification of invasive aspects, i.e. they cannot prevent changes of attribute values of one object or aspect by another aspect.

3.2 Reuse and Revision of Workflows

Activity and workflow based approaches are aimed to specify complete system behaviour that can be analyzed and verified against required properties. The workflows are often used in AOM approaches as integration means to combine specified aspects. There are also AOM approaches that define fragments of workflows and compose workflow fragments into the complete workflow.

Workflows for Integration. The Theme (Clarke and Baniassad, 2005) approach is a good example of the first type of workflow use. At the level of requirements a designer identifies actions as verbs in textual requirements. Each action potentially becomes a *theme* depicted as a rhomb (Figure 2). The actions have relations via concepts depicted as parallelograms with rounded corners. Actions found in different themes become potential aspects. An action view of each theme is a graph that contains actions and entities. The size of this view grows with the number of entities and actions in the model. The scalability is achieved by the separation of major and minor actions. “Major actions become themes, while minor actions are slotted to become methods within a theme” (Baniassad and Clarke, 2004). This means that the major actions in the Theme approach are not elementary, they are activities. The elements of behaviour are method calls and returns.

The action view is analyzed to produce the Theme/UML specification of actions-aspects as combination of class and sequence diagrams. The Design level Theme/UML supports modelling of aspects in UML. Theme/Doc views are mapped onto the Theme/UML model allowing traceability of requirements. If a Theme/action is reused in the model, the join points are specified in the Theme/UML view. Figure 2 shows that Themes *register* and *logged* identified in the workflow are specified as combinations of class and sequence diagrams called <*theme*>*Register* and <*theme*>*Logger*. In this case the methods of themes *register*, *unregister* and *give* should be specified as join points of the <*theme*>*Logger*.

The action view “drives composition semantics for design in Theme/UML”. The composition techniques are concatenation and hierarchy of *Major* and *Minor* actions. From the Theme/UML and Theme/Doc specifications the corresponding computation tree can be generated and may be compared with the workflow built at the level of requirements. For example, such approach as GrACE (Graph-based Adaptation, Configuration and Evolution) (Ciraci et al., 2009) uses different graph transformation techniques in order to compose behaviours presented by class, sequence diagrams and the action view in the corresponding computation tree of the designed system and apply model checking and verification algorithms to analyse the properties of the system.

- PROS: In addition to conventional activity diagrams Workflow diagrams in Theme/UML separate aspects and minor actions to increase the abstraction level and decrease the size of workflows.
- CONS: The used composition techniques, namely concatenation and hierarchy of activities, make it impossible to add new aspect models at the requirements level without verification of the complete model. The evolution of the model is handled by re-generating the views for any new set of requirements. Reasoning on models does not become easier as the minor actions are specified by sequence diagrams, but the composition of them as a workflow. Reasoning demands relatively heavy procedures of building the computation tree, reachability analysis and verification of dynamic properties.

Composed workflows. Another direction in the use of workflow semantics for aspect-oriented modelling is specifying fragments of a workflow and then composing them into the complete workflow. The composition techniques used in workflows are restricted to concatenation and hierarchical inclusion of fragments.

The ADORE approach (Mosser et al., 2010) (Activity moDel supOrting oRchestration Evolution) re-examines the Business Process Execution Language (BPEL) to enable separation of concerns.

An orchestration of services is defined in ADORE as a partially ordered set of activities. The types of activities that can be defined in ADORE include service invocation (denoted by *invoke-inv*), variable assignment (*assign-a*), fault reporting (*throw*-t), message receiving (*receive- rcv*), response sending (*reply-rp*), and the null activity(*nop*), which is used for synchronization purpose (Figure 3).

Each process (fragment) corresponds to a specific concern and uses a partial point of view on its target. A fragment contains special activities, called predecessors *P*, successors *S* and a hook (assimilated as a proceed in AspectJ) that represents a part of a trace where the fragment is connected into an existing or-

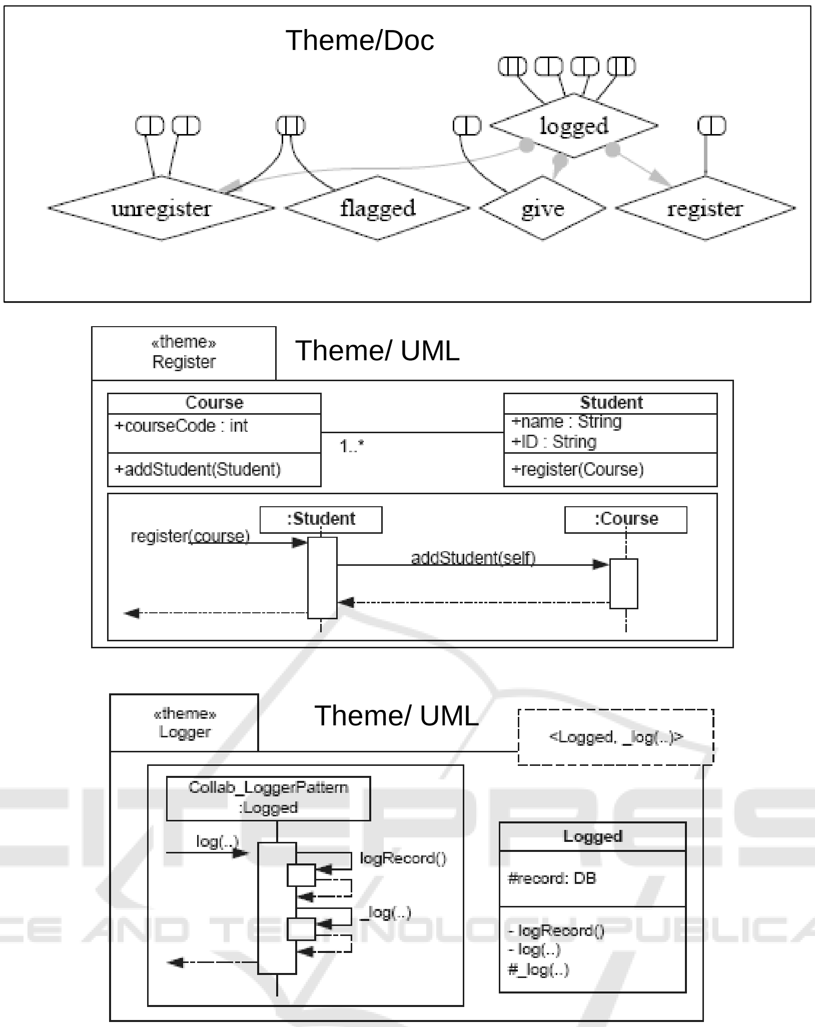


Figure 2: Theme model.

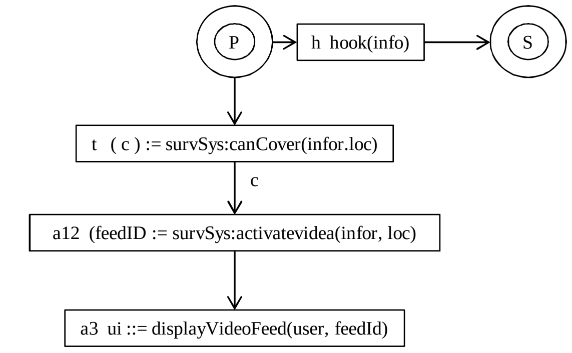

P
h hook(info)
S
t (c) := survSys:canCover(infor.loc)
c
a12 (feedID := survSys:activatevidea(infor, loc)
a3 ui ::= displayVideoFeed(user, feedId)

Figure 3: ADORE model. Fragment requestVideo<user>.

chestration. The hook predecessors *(P)* are the immediate predecessors of the first activity in the target block, and the hook successors *(S)* are the immediate successors of the last activity in the block.

Fig. 3 describes the following behaviour. After the execution of the hook predecessors *(P)*, perform the activity block referred to by *(h)* and then continue with the hook successors *(S)*. The vertical branch of the fragment represents the additional behaviour. In parallel to the behaviour described by the hook *(h)* the system first determines if the surveillance system can cover the crisis area *(t)*. If this functionality is available for this location, the process requests a video feed (single ADORE activity *a12*) and then broadcasts it to the Coordinator interface *(a3)*.

The binding instructions are specified in a separate file. For example, the fragment from Fig. 3 can be bound on the block of activities

$$a3 : cms : buildchecklist(i)$$

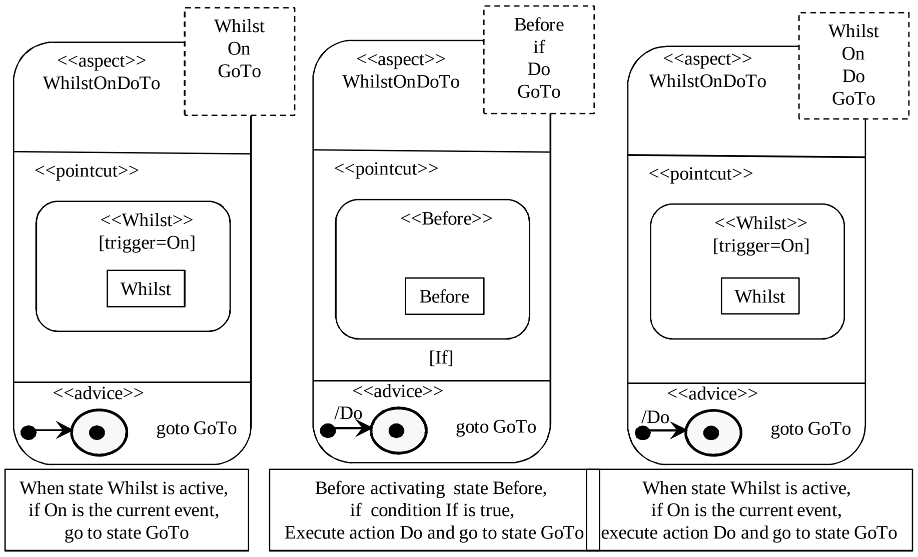


Figure 4: HiLA patterns.

and

$$a4 : (ci)ui := promptchecklist(coord, id, cc1)$$

of the base orchestration. The predecessor of $a3$ is assigned to P, the successor of $a4$ is assigned to S and the *hook* is assigned to $\{a3, a4\}$. The complete orchestration is generated from fragments according to the binding instructions.

As with such an approach there is no guarantee that any particular local properties of aspects are propagated to the complete orchestration, the side effects of separating of concerns and composition in ADORE are formulated as rules. The violating of rules not always signals about a mistake, this may indicate a "bad-smell", like, for example, the non-determinism caused by two conditions evaluated to false at the same time. The "bad-smells" are analyzed by the designer of the orchestration.

- PROS: ADORE extends BPEL with means for modelling of aspects. Models become closer to requirements.
- CONS: The composition techniques in workflows cause the need of re-generating and "bad-smells" re-analyzing of the complete orchestration after any change or adding any new fragment. Reasoning on models is not localized and demands reachability analysis and theorem proving.

3.3 Revision of State Machines

A UML Behaviour State Machine (BSM) (OMG, 2003) usually presents behaviour of one classifier.

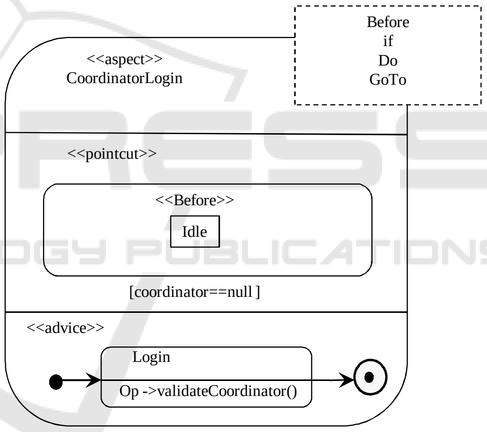


Figure 5: HiLA model of the Login Concern.

The execution semantics of BSM (McNeile and Roubtsova, 2009) is complex, involves a queue or a stack of active events and rules of their handling or keeping in the queue until the state is appropriate for their handling.

There are several approaches revising Behaviour State Machines (BSM) trying to extend behaviour specified by BSM for one classifier. The approach by (Mahoney et al., 2004) exploits the *AND-composition* of several independent (orthogonal) statecharts defined by D.Harel (Harel and Gery, 1997). "The key feature of orthogonal statecharts is that events from every composed statechart are broadcast to all others. Therefore an event can cause transitions in two or more orthogonal statecharts simultaneously" (Mahoney et al., 2004).

The ideas proposed by Mahoney et al. were further developed in the approach called High-Level Aspects (HiLA) (Hölzl et al., 2010). HiLA modifies the semantics of BSM allowing classifiers to apply additional or alternative behaviour. Aspects extend the behaviour specified for classifiers.

The basic static structure usually contains one or more classes. Each base state machine is attached to one of these classes and specifies its behaviour. HiLA offers two kinds of aspects to modify such a model. Static aspects directly specify a model transformation of the basic static structure of the model. Dynamic (high-level) aspects only apply to state machines and specify additional or alternative behaviour to be executed at certain "appropriate" points in time in the base machines execution.

HiLA introduces patterns for specification of dynamic aspects (Figure 4). A pattern *Whilst* always has an annotation $Trigger = e$. "Conceptually it selects the compound transition from $State^{*}$ to $State^{**}$ with trigger *e*, but if this transition does not exist, it is created." This means that an aspect is added while an action is in the stack of active actions.

A pattern labelled with *Before* selects each transition *T* entering any state contained in $State^{*}$, but only in active state configurations where after taking *T* all states in $State^{**}$, will become active"(Hölzl et al., 2010). An aspect may extend the behaviour of a class or introduce a new class to the specification.

Figure 5 shows the aspect validating if the *Coordinator* is logged in. The aspect is taken from the model of a crisis management system (Holzl1 et al., 2010). Before the *Idle* state of system becomes active, the system checks whether the coordinator is not logged in *(coordinator == null)*. If this is the case, the *Login* sub-state machine is triggered and operation *validateCoordinator()* becomes possible. The aspect is regulative, it changes the flow of control in the base state machine.

- PROS: HiLA patterns support specification of aspects in BSM.
- CONS: Evolution of models and reasoning on them do not become easier. Active objects represented with classifies with aspects and new aspect classes execute asynchronously resulting in non-deterministic system behaviour. HiLA uses formal methods of model validation. The application of aspects to BSM results in another UML state machine which is analyzed using the model checking component of Hugo/RTmodel checking tools. Hugo/RT translates the state machine and the assertions into the input language of a back-end model checker SPIN. SPIN then verifies the given properties presented in Linear Temporal Logic.

3.4 Revision of Design by Contract

The search of composition techniques suitable for AOM led to the new look at the set-theory composition. Such approaches as Package Merge (Ammour and Desfray, 2006) and Visual Contract Language (VCL) (Amálio and Kelsen, 2010) explore the declarative way of aspect specification based on composition of sets.

VCL revises the Design By Contract (DbC) principle. An element of behaviour is an operation. According to the DbC (Meyer, 1991) the pre-conditions and post-conditions have to be defined for any specified behaviour.
- If the precondition is satisfied then the result of the corresponding behaviour is defined. It satisfies the post-condition.
- If a precondition is violated, the effect of the corresponding behaviour becomes undefined.
This means that only the non-interactive part of behaviour that takes place, when the preconditions are not violated, can be composed and analyzed. VCL inherits this property of DbC.

A VCL model is organized around packages, which are reusable units encapsulating structure and behaviour. Packages represent either a traditional module or an aspect. VCL's package composition mechanisms allow larger package to be built from smaller ones. A package is denoted by a cloud. In Figure 6 package *Authentication Ops* extends package *Authentication*. A package encapsulates state and behaviour in form of attributes and operations. Package *Authentication Ops* contains operations *Login* and *Logout*.

Ordinary packages define global state. Abstract packages do not define global state, they act as containers for state structures and their local behaviour to be used in other packages.

VCL behavioural diagrams (BDs) identify the operations of a package. There are two types of operations: update and observe (or query). Update operations perform changes of state in the system; they involve a pair of states: before-state (described by pre-condition) and an after-state (described by post-condition). They are defined in VCL contract diagrams. Figure 6 shows examples of VCL contract diagrams for *LogIn* and *LogOut* operations that belong to the package *Authentication Ops*.

Aspects are composed using join extensions, which is illustrated in Figure 7. In a join extension, there is a contract that describes the joining behaviour of an aspect (a join contract) that is composed with a group of operations placed in a join-box. All operations of package *CrisisWithJI* are conjoined with

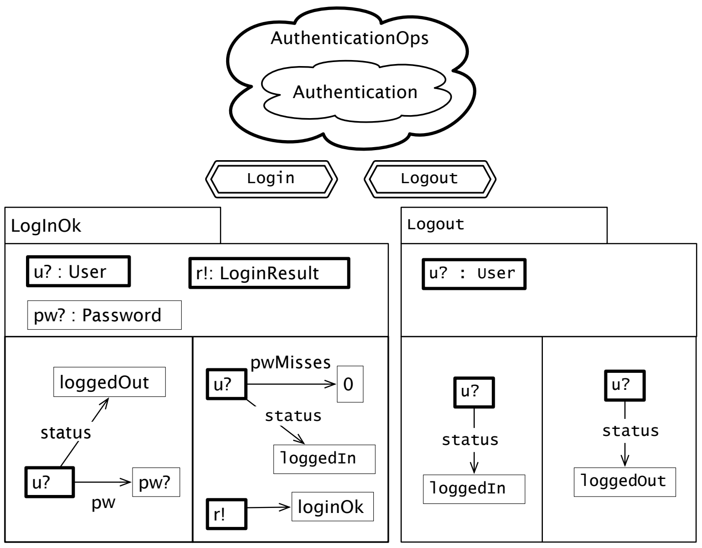


Figure 6: Packages in VCL.

join contracts *LoggingOp*, *SessionMgmtOp* and *AuthorisationOp*. Join contract *AuthorisationOp* specifies the extra behaviour of the *Authorisation* concern by adding an extra pre-condition to all operations of package *CrisisWithJI*; this specifies that the users executing operations of package *CrisisWithJI* must be logged-in and have the required permissions to execute that task.

VCL is designed with a formal Z semantics and so it has the potential for verification and global reasoning using theorem proving.

- PROS: The VCL extends the DbC approach by visual means to capture aspects. Modelling of packages is simple. It demands only knowledge of the set theory and predicate logic.

- CONS: The use of the DbC does not allow localizing of reasoning on aspects about behaviour of the whole model. Verification of the complete model is needed after every modification in the number and the content of aspects.

3.5 Revision of Mixins

Mixins are applied in the Protocol Modelling approach defined by McNeile and Simons (McNeile and Simons, 2003). A protocol model of a system is a composition of protocol machines. Protocol machines are partial descriptions of class behaviours. The composition operator for protocol machines is a variant of the parallel composition operator defined by Hoare (Hoare, 1985) in his process algebra Communication of Sequential Processes (CSP). This operator was extended by McNeile and Simons (McNeile and Simons, 2003) for machines and events with data.

A protocol machine has its own alphabet of recognized events and a local state. An event type is specified as a data structure (Figure 8). An event instance contains values of the attributes.

The local state of a machine is presented as a set of own attributes and attributes derived from the state of other machines. A machine can read the state of other machines but cannot alter it. The derived states should not be topologically connected. They are already spectative aspects inside of protocol machines.

Moreover, a machine has a set of transitions being triples *(state, event, state).* Events are presented to the model by the environment. Being in a suitable state, a protocol machine accepts the presented event, otherwise it refuses the event. A protocol model handles one event at a time and reaches a well defined quiescent state between each event. An element of behaviour is an accepted event. The behaviour of a protocol model is a set of sequences of accepted events.

Protocol machines use the CSP parallel composition algorithm to form more complex protocol machines. This is the description of the CSP parallel composition algorithm:
- If all machines of the protocol model having an event in their alphabet accept the event, the protocol model accepts it.
- If at least one of protocol machines having this event in its alphabet refuses the event, the composition of machines refuses it.
So, the semantics of refusal which is absent in other modelling semantics (BSM, workflows, sequence diagrams and contracts) is the key to synchronization and CSP parallel composition.

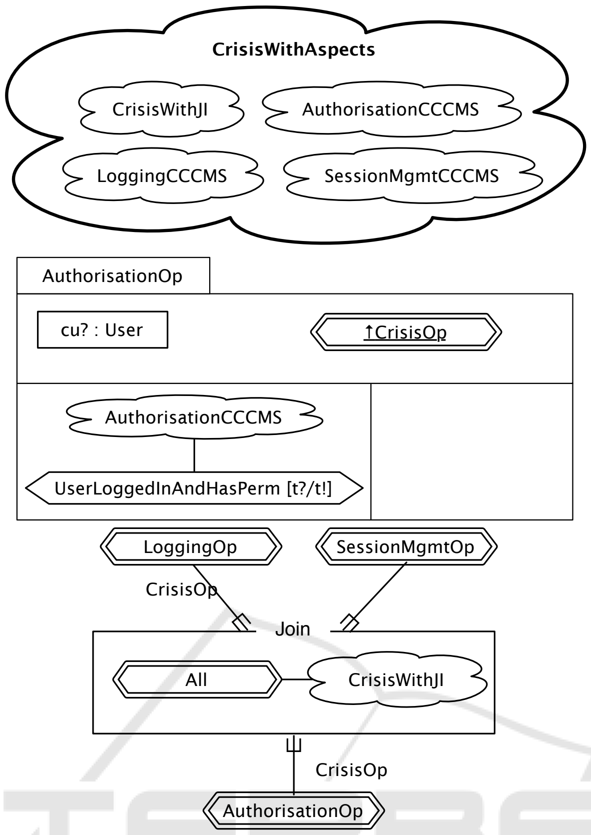


Figure 7: Aspect Composition in VCL.

It is recognized in (McNeile and Roubtsova, 2008) that protocol machines are natural abstractions to specify aspects. For example, a protocol model of a bank *Account* and a *Customer* is shown in Figure 8. Behaviours of both objects include and equally compose behaviours of aspects *Freezing* and *Freeze Control*. These aspects model the possibility to freeze the behaviour of an account or a customer. The INCLUDE-relation shown as a half-dashed triangle gives to Protocol Models the expressiveness of multiple inheritance. Behaviours of aspects are instantiated with instantiation of objects, each of which has its object identifier.

All behaviours are equally composed on the basis of the CSP parallel composition as shown in Figure 9. A new registered *Customer* and a new open *Account* are *not-frozen*. A bank security service may freeze both an *Account* and(or) a *Customer* submitting events *Freeze.Customer* and(or) *Freeze.Account*. Then such an *Account* or *Customer* transits to state *Frozen*. A *Frozen* behaviour cannot accept any event. Its state is derived from the instances of the behaviour *Freezing* included into the objects *Account* and *Customer* correspondingly.

A join point in protocol model is a set of events or states that can be seen identical in the context a particular protocol machine.
For example, the join point *Generic Operate* matches events *Deposit, Withdraw, Transfer, Leave, and Close* and any of these events can be accepted only in state *Freeze Not Active* which is derived from state *not Frozen.*

The proof presented in (McNeile and Roubtsova, 2008) shows that the CSP parallel composition of Protocol Machines guarantees preservation of ordering of traces of aspects in the whole specification. This property is called *local reasoning* or *observational consistency* (Ebert and Engels., 1994). It simplifies reasoning on models as the traces can become longer or shorter but the order of events will be never changed. For example, trace *Open, Withdraw, Freeze Account* is a trace of the composition of *Freezing* and *Account*, but the sub-trace *Open, Withdraw* is a valid trace of *Account*. Small and deterministic protocol machines are verified or tested by direct execution. Then the syntactic checks of specifications of join point are sufficient for verification of the whole model.

- PROS: Protocol machines with generic events and

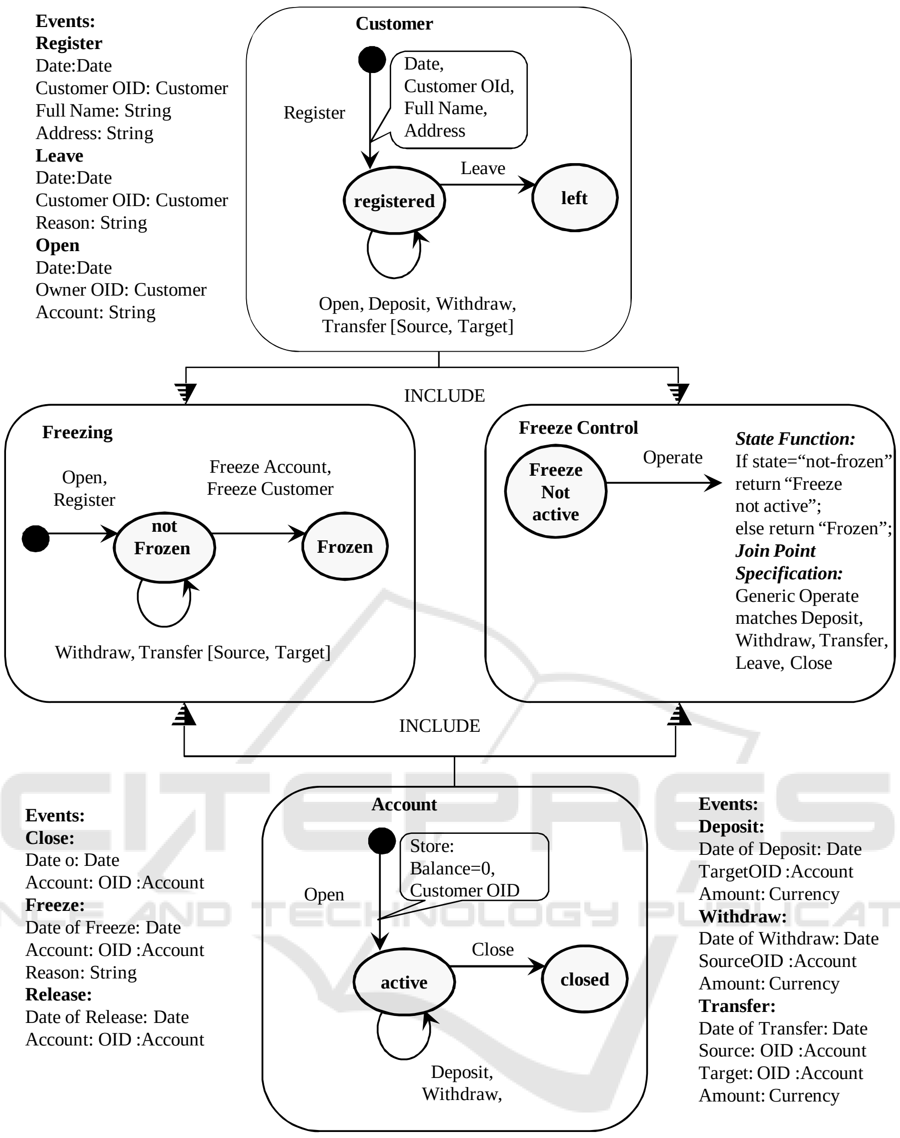


Figure 8: Protocol Model.

states and the synchronous composition are natural means for modelling of aspects. The CSP parallel composition provides flexible support for model evolution. Protocol machines are not hierarchical, they are equally composed. The composition operator guarantees preservation of behaviour and properties of aspects in the behaviour of the whole system.
- CONS: The understanding of a protocol model is not very intuitive. It takes some time to begin freely apply the CSP parallel composition. However, the tool Modelscope[1] supports execution and understanding of protocol models.

[1]*http://www.metamaxim.com/pages/download.htm*

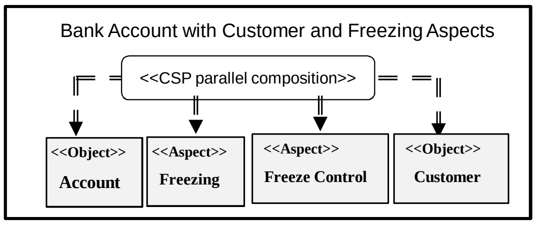

Bank Account with Customer and Freezing Aspects
<<CSP parallel composition>>
<<Object>> Account
<<Aspect>> Freezing
<<Aspect>> Freeze Control
<<Object>> Customer

Figure 9: CSP composition of Protocol Machines.

4 CONCLUSIONS

Models should have higher abstraction level than programs and may more freely experiment with composition techniques unavailable in the main stream modelling languages. However, the composition techniques of many modelling techniques nowadays does not deviate from composition forms available in dominant programming languages. These languages are designed to cover asynchronous communication and have restricted means to deal with shared data. That is why it is not a surprise that the extension of models with the aspect building constructions causes the same problems as in aspect-oriented programs. Namely, such AOM models do not prevent specification of invasive aspects, do not guarantee local reasoning and have to be globally verified after every modification. Using AOM approaches with the same composition techniques as in programs we just move the complexity from models to analysis techniques and tools and rely heavily on results of this analysis. As the models are subject of frequent changes, after any new change the analysis should be repeated. This dependence on analysis and verification after any step of modelling slows down the modelling process, restricts the number of demonstrations of the model to the client and makes the modelling process inflexible.

However, the understanding of the nature of aspects can simplify the analysis. Aspect-abstractions exist at the same level as objects and they are not distributed from object. Therefore they need synchronization with objects and access to shared data. The lessons learned from the analysis of aspect-oriented modelling approaches trying to use asynchronous composition for aspects shows that approaches confuse aspects and other abstraction sorts. Each type of abstraction should be used on its own place. Aspects and objects present problem frames, share data and communicate synchronously. The distributed components or services built from aspects and objects communicate asynchronously (Roubtsova and McNeile., 2009). Such separation will simplify the AOM analysis.

The mixin-based group of AOM approaches offers to AOM a synchronous and agile way of modelling.

- The semantic restriction when one machine cannot alter the state of other machines makes it impossible to specify invasive aspects.
- Using sets of events and states as join points is more abstract and technology independent than using operation calls and returns.
- Derived states provide unlimited weaving abstractions.
- Determinism of machines is assured by the concept of quiescent states.
- The CSP parallel composition technique guarantees the property of local reasoning on aspects about the behaviour of the whole system. The local properties of mixins survive their composition and are preserved in the whole behaviour of the system.

  As a consequence of local reasoning, analysis and execution of mixin-based models does not require generation of the complete model or a computation tree after adding/removing/changing of aspects.

  The semantics of event refusal splits the global computation tree into sets of traces of woven protocol machines. The traces of the whole model are generated on the fly by a tool or a middleware implementing CSP parallel composition algorithm.
- All these features reduce the analysis of mixin-based models to verification or testing of separate machines and syntactic checks of specification of join points.

As the proposed semantics is notation independent and it can be applied in many approaches, the next step is to apply it and investigate the implementation of systems in mixin-based and object-oriented languages. Most probably that offering of a CSP parallel composition middleware service will result in a new group of aspect-oriented languages.

ACKNOWLEDGEMENTS

The author thanks all participants of the workshops ABMB'05, ABMB'06, AOM'08, BM-MDA'09, AOM'10 in Bellairs, and BM-FA'10 for fruitful discussions of different modelling approaches.

REFERENCES

Amálio, N. and Kelsen, P. (2010). VCL, a Visual Language for Modelling Software Systems Formally. In *Diagrams*, pages 282–284.

Ammour, S. and Desfray, P. (2006). A Concern-based Technique for Architecture Modelling Using the UML Package Merge. *M.Aksit and E.Roubtsova eds. Proceedings of ABMB2005, ENTCS*, 163(1):7–18.

Baniassad, E. and Clarke, S. (2004). Theme: An Approach for Aspect-Oriented Analysis and Design. *ICSE 2004. Proceedings. 26th International Conference on Software Engineering. IEEE*, pages 158–167.

Cardone, R. and Lin, C. (2004). Using Mixin Technology to Improve Modularity. *Aspect-Oriented Software Development. R.Filman, T.Elrad, S.Clarke, M.Akşit eds.Addison-Wesley*, pages 219–241.

Chitchyan, R., Rashid, A., Sawyer, P., Garcia, A., Alarcon, M., Bakker, J., Tekinerdogan, B., Clarke, S., and

Jackson, A. (2005). Survey of Aspect-Oriented Analysis and Design Approaches. *Technical Report D11 AOSD-Europe-ULANC-9, AOSD-Europe*.

Chwif, L., Barretto, M. R. P., and Paul, R. (2000). On simulation model complexity. In *WSC '00: Proceedings of the 32nd conference on Winter simulation*. Society for Computer Simulation International.

Ciraci, S., Havinga, W. K., Aksit, M., Bockisch, C. M., and van den Broek, P. M. (2009). A Graph-Based Aspect Interference Detection Approach for UML-Based Aspect-Oriented Models. Technical Report TR-CTIT-09-39, Enschede.

Clarke, S. and Baniassad, E. (2005). *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison Wesley.

Ebert, J. and Engels., G. (1994). Observable or invocable behaviour-you have to choose. *Technical report. Universitt Koblenz, Koblenz, Germany*.

Filman, R., Elrad, T., Clarke, S., and Akşit, M. (2004). *Aspect-Oriented Software Development*. Addison-Wesley.

Filman, R. and Friedman, D. (2004). Aspect-Orineted Programming is Quantification and Obliviosness. *Aspect-Oriented Software Development. Addison-Wesley*, pages 21–35.

Harel, D. and Gery, E. (1997). Executable Object Modelling with Statecharts. *IEEE Computer, 30(7)*, pages 31–42.

Hoare, C. (1985). *Communicating Sequential Processes*. Prentice-Hall International.

Hölzl, M. M., Knapp, A., and Zhang, G. (2010). Modeling the Car Crash Crisis Management System Using HiLA. *T. Aspect-Oriented Software Development*, 7:234–271.

Holzl1, M., Knapp, A., and Zhang, G. (2010). Modeling the car crash crisis management system using HiLA. *S. Katz et al. (Eds.): Transactions on AOSD VII, LNCS 6210*, pages 234–271.

Jackson, M. (2005). *Problem frames and software engineering*, volume 47. Butterworth-Heinemann.

Katz, S. (2006). Aspect Categories and Classes of Temporal Properties. *Transactions on Aspect-Oriented Software Development, LNCS 3880*, pages 106–134.

Kiczales, G. and Mezini, M. (2005). Aspect-Oriented Programming and Modular Reasoning. *Proc. of the International Conference on Software Engineering*, pages 49–58.

Mahoney, M., Bader, A., Elrad, T., and Aldawud, O. (2004). Using Aspects to Abstract and Modularize Statecharts. *In the 5th Aspect-Oriented Modeling Workshop In Conjunction with UML 2004*.

McNeile, A. and Roubtsova, E. (2008). CSP parallel composition of aspect models. *AOM'08: Proceedings of the 2008 AOSD Workshop on Aspect-Oriented Modeling*, pages 13–18.

McNeile, A. and Roubtsova, E. (2009). Composition semantics for executable and evolvable behavioral modeling in MDA. In *BM-MDA '09: Proceedings of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture*, pages 1–8, New York, NY, USA. ACM.

McNeile, A. and Simons, N. (2003). State Machines as Mixins. *Journal of Object Technology*, 2(6):85–101.

Meyer, B. (1991). Design by contract. *Advances in Object-Oriented Software Engineering, eds. D. Mandrioli and B. Meyer*, pages 1–50.

Mosser, S., Blay-Fornarino, M., and France, R. (2010). Workflow Design Using Fragment Composition - Crisis Management System Design through ADORE. *T. Aspect-Oriented Software Development*, 7:200–233.

OMG (2003). *Unified Modeling Language: Superstructure version 2.1.1 formal/2007-02-03*.

Op de Beeck, S., Truyen, E., Boucke, N., Sanen, F., Bynens, M., and Joosen, W. (2006). A study of aspect-oriented design approaches. *Technical Report CW435, Department of Computer Science, Katholieke Universiteit Leuven*.

Reddy, R., Ghosh, S., France, R., and Straw, B. (2006). Directives for composing aspect-oriented design class models. *Transactions on AOSD, LNCS 3880* , pages 75–105.

Reina, A. M., Torres, J., and Toro, M. (2004). Separating Concerns by Means of UML-profiles and Metamodels in PIMs. *In Proc. of the 5th Aspect-Oriented Modeling Workshop (UML04), Lisbon, Portugal*.

Robinson, S. (1994). *Successful simulation: A practical approach to simulation projects*. McGraw-Hill, Maidenhead,UK.

Roubtsova, E. and McNeile., A. (2009). Abstractions, composition and reasoning. *Proceedings of the 13th Intl Workshop on Aspect-Oriented Modeling, Charlottesville. Virginia, USA, ACM DL*, pages 19–24.

Schauerhuber, A., Schwinger, W., Kapsammer, E., Retschitzegger, W., and Wimmer, M. (2007). A survey on aspect-oriented modeling approaches. Technical report, E188 - Institut für Softwaretechnik und Interaktive Systeme; Technische Universität Wien.

Stein, D. and Hanenberg, S. (2006). Why Aspect-Oriented Software Development And Model-Driven Development Are Not The Same - A Position Paper. *M.Aksit and E.Roubtsova eds. Proceedings of ABMB2005, ENTCS*, 163(1):71–82.

Stein, D., Hanenberg, S., and Unland, R. (2005). Visualizing Join Point Selections Using Interaction-Based vs. State-Based Notations Exemplified With Help of Business Rules. *EMISA-2005*, pages 94–107.