Aspect-Oriented Model Development at Different Levels of Abstraction

Mauricio Alférez[1], Nuno Amálio[2], Selim Ciraci[3], Franck Fleurey[4], Jörg Kienzle[5], Jacques Klein[2], Max Kramer[6], Sebastien Mosser[7], Gunter Mussbacher[8], Ella Roubtsova[9], and Gefei Zhang[10]

[1] Universidade Nova de Lisboa, Portugal, mauricio.alferez@di.fct.unl.pt
[2] University of Luxembourg, {nuno.amalio,jacques.klein}@uni.lu
[3] University of Twente, the Netherlands, ciracis@ewi.utwente.nl
[4] SINTEF IKT, Norway, Franck.Fleurey@sintef.no
[5] McGill University, Canada, Joerg.Kienzle@mcgill.ca
[6] Karlsruhe Institute of Technology, Germany, max.kramer@student.kit.edu
[7] INRIA Lille - Nord Europe, sebastien.mosser@inria.fr
[8] SCE, Carleton University, Canada, gunter@sce.carleton.ca
[9] Open University of the Netherlands and Munich University of Applied Sciences, Germany, ella.roubtsova@ou.nl,ella.roubtsova@hm.edu
[10] Ludwig-Maximilians-Universität München and arvato systems, Germany, gefei.zhang@pst.ifi.lmu.de

Abstract. The last decade has seen the development of diverse aspect-oriented modeling (AOM) approaches. This paper presents eight different AOM approaches that produce models at different level of abstraction. The approaches are different with respect to the phases of the development lifecycle they target, and the support they provide for model composition and verification. The approaches are illustrated by models of the same concern from a case study to enable comparing of their expressive means. Understanding common elements and differences of approaches clarifies the role of aspect-orientation in the software development process.
Key words: Aspect-oriented modeling, localization of concerns, composition, verification, localization of reasoning

1 Introduction

Separation of concerns is a key software engineering principle that helps to reduce complexity, improve reusability, and simplify evolution. Aspect-oriented software development (AOSD) takes traditional support for separating concerns a step further by allowing developers to modularize their descriptions along more than one dimension [14].

Drawing inspiration from aspect-oriented programming research, AOM brings the aspect-orientation to design, analysis and requirements phases of software development. Aspect-oriented modeling (AOM) approaches, in particular, aim to provide means for

- *localizing of crosscutting concerns* at the level of models to guarantee traceability of concerns across the software development lifecycle and reuse of different realizations of a concern within and across software models;
- *verification* of models with crosscutting concerns;
- *localizing of reasoning* on models of concerns about the behaviour of the whole model.

This paper surveys a set of AOM approaches working at different levels of abstraction. The aim is to compare the techniques of localization of aspects and the techniques of reasoning on aspect models and identify research challenges in AOM. Section 2 identifies the abstraction level of each of eight different AOM approaches and illustrates each of the approaches with a model of the same concern. All chosen approaches have demonstrated their scalability by taking the challenge of modelling the case study of a crisis management system (CMS) [8]. Section 3 discusses the approaches and identifies future directions for AOM research.

2 AOM at Different Levels of Abstraction

2.1 Authentication Concern

The description of the authentication concern is taken from the requirements for a Crisis Management System [8]. This concern is modelled and used for correctness analyses in all compared AOM approaches.

```
The system authenticates users on the basis of the access policies when they first access
any components or information. If a user remains idle for 30 minutes or longer, the system
shall require them to re-authenticate.

Use Case: AuthenticateUser
    Scope: Car Crash Crisis Management System
    Primary Actor: None
    Secondary Actor: CMSEmployee
    Intention: The intention of the System is to authenticate the CMSEmployee to allow access.

    Main Success Scenario:
        1. System prompts CMSEmployee for login id and password.
        2. CMSEmployee enters login id and password into System.
        3. System validates the login information. Use case ends in success.
    Extensions:
        2a. CMSEmployee cancels the authentication process. Use case ends in failure.
        3a. System fails to authenticate the CMSEmployee.
        3a.1 Use case continues at step 1.
        3a.1a CMSEmployee performed three consecutive failed attempts.
              Use case ends in failure.
```

2.2 Feature Abstractions

In Variability Modelling Language for Requirements (VML4RE) [1] the *Authentication* concern is *localized* as one reusable feature identified by its name "Authentication"(Figure 1). Feature model visualizes the dependencies between the *Authentication* feature and the *Session Handling* and *Administration* features.

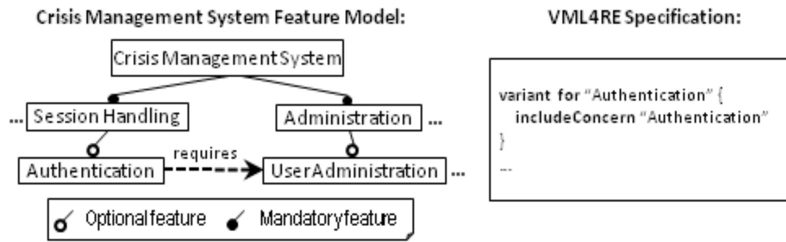


Fig. 1. VML4RE model

Authentication requires the *UserAdministration* feature because the system authenticates users on the basis of access policies associated to them. Feature model also identifies *Authentication* as an optional feature.

The VML4RE specification is a good starting point for concern modelling. Different requirements models can be created to describe the features, using notations of UML2.0 or approaches presented in this overview. For example, if the *Authentication* feature is selected for a specific product according to the VML4RE specification in Figure 1, the requirements specification will include the specification concern called "Authentication" in the chosen specification notation (e.g. a use case specification or a sequence diagram). Also, according to combinations of more than one feature it is possible to apply concerns that modify or add new parts in the requirements specifications. Features do not specify system internals, they only capture requirements for a system, so the *verification* of the system's behaviour and *any reasoning* about it *are not applicable* at the level of feature modelling.

2.3 Use cases

At the level of use case modelling the *Authentication* use case is described step by step. As it is recognized as a reusable unit, it should contain pointcut designators (instructing where, when, and how to invoke the advice) and join points (defining places in the model where an advice should be inserted) [13]. Such concepts do not exist in conventional use case notations.

Aspect-oriented User Requirements Notation (AoURN) [12] supports conventional concepts of use case and workflow modelling techniques but also enables localizing of aspects. The primary goal of AoURN is to model any functional or non-functional concern of the system under development that can be expressed with scenarios.

Fig. 2 depicts the AoURN scenario model for the *Authentication* concern. The authentication scenario starts at the *authenticate* start point and ends either at the *authenticated* or *fail* end point. Various conditions are checked: the *User* may have to *enter credentials*, and the *System* may *authenticate* or *block* the *User*. The pointcut stub *RequiresAuthentication* represents all locations where

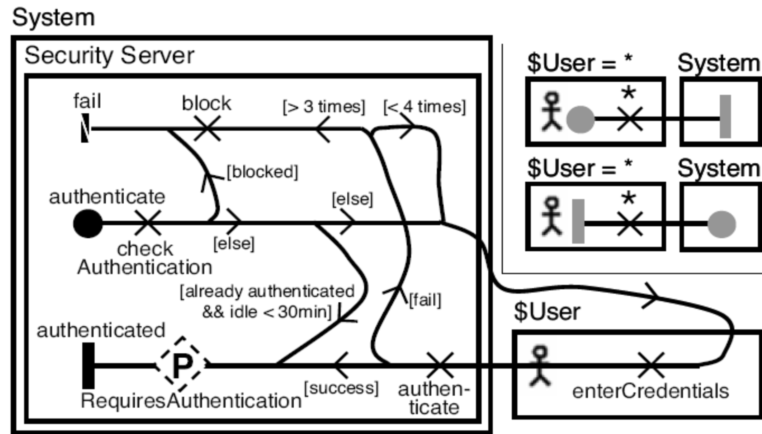


Fig. 2. AoURN model

the Authentication concern is to be applied. At one glance, it is apparent that the concern is to be applied before these locations since the concern-specific behaviour occurs before the pointcut stub. In this case, a simple sequential composition is desired, but AoURN scenario models can be composed in many different ways (e.g., as alternatives, optionally, in parallel, in loops, or in an interleaved way). The composition rules are exhaustive in that their expressiveness is only restricted by the AoURN scenario language itself.

Patterns define the actual locations where the concern is to be applied: in this case, each time there is an interaction between an actor and the *System* as shown in the two sub-models above the *User* component. The variable *$User* defined in the patterns allows the concern to reuse the matched component.

AoURN models involve neither detailed data nor message exchanges. This makes them well suited for the early stages of the software development process. AoURN scenario definitions can be analyzed, enabling regression-testing of the scenario model. AoURN combines aspect-oriented modeling with goal-oriented modelling allowing to model the reasons for choosing a concern using goal models.

Use cases identify abstract actions coming from the environment and abstract responses of the system, driving the system from one state to another, but do not capture system local storage and the internal behavior. While it is possible to validate use case models, system *verification* and *local reasoning* on aspects about the whole system behaviour are *not applicable* at the level of use cases. Further system specification involves choices. Actions may become operations, messages, or events recognized by objects and aspects. Depending on these choices, different modelling techniques may be used.

2.4 Classes and Sequence Diagrams

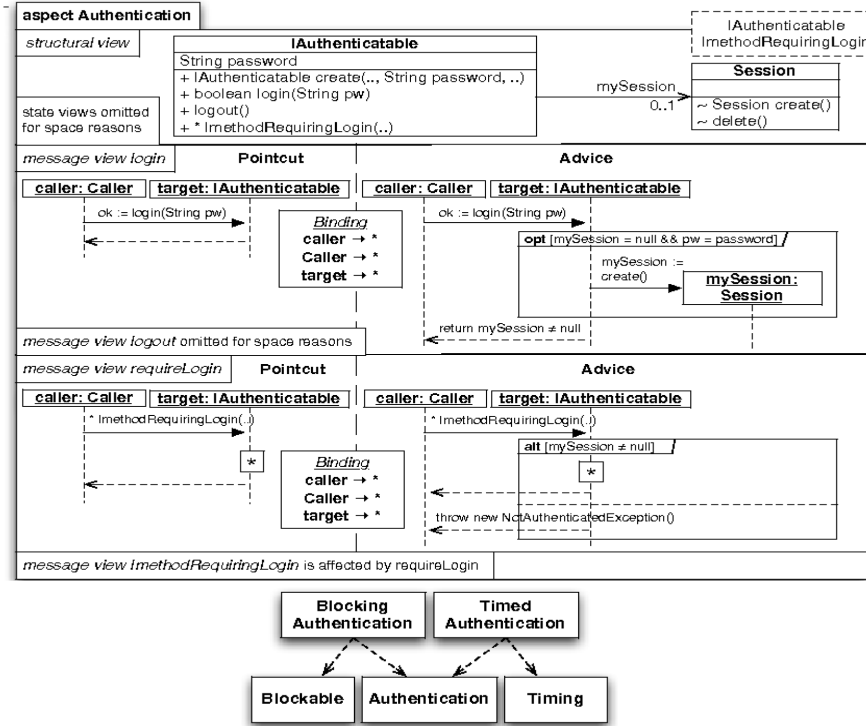


Fig. 3. RAM model

Reusable Aspect Models (RAM) [7] describes the structure and behaviour of a concern using class, state and sequence diagrams. Fig. 3 shows how the structural view of the *Authentication* concern associates *Session* objects with |*Authenticatable* objects.

The *Authentication* behavior is described in *state views* and *message views*. State views detail the method invocation protocol of objects using state diagrams. Message views specify the message passing between objects using sequence diagrams. For example, the *login* message view in Fig. 3 shows how a *session* object is instantiated upon a successful login attempt. The *requireLogin* message view demonstrates how method invocations of |*methodRequiringLogin* of an |*Authenticatable* object are disallowed if no session is currently established. To apply the authentication aspect, the mandatory instantiation parameters must be mapped to model elements of the base model. For instance, to enable user

authentication, the mapping |*Authenticatable* $\rightarrow$ *User*, |*methodRequiringLogin* $\rightarrow$ * *(..)* would ensure that no public method of a *User* object can be invoked before the user authentication.

To reuse structure and behavior of low-level aspect models in models that provide more complex functionality, the RAM approach supports the creation of complex aspect dependency chains. Blocking authentication and timed authentication are modeled by reusing the aspects *Authentication*, *Blockable* and *Timing* (see bottom of Fig. 3).

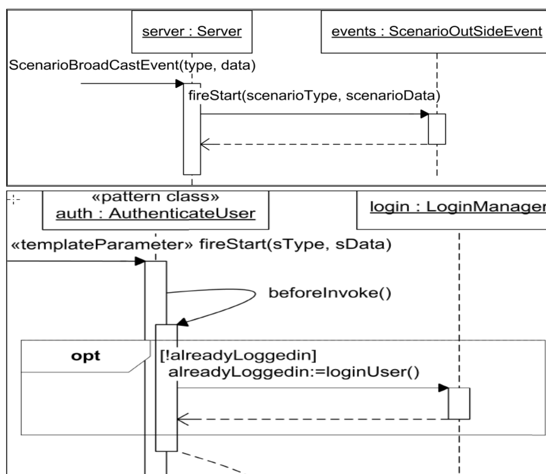


Fig. 4. GrACE:Sequence diagrams

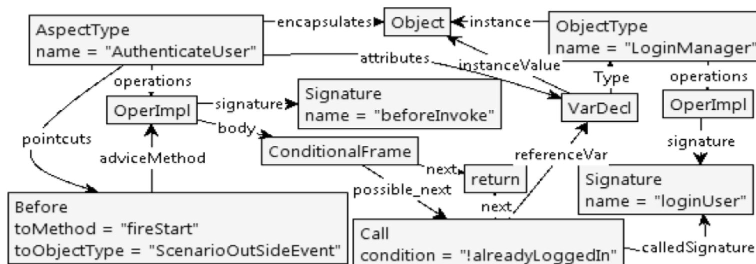


Fig. 5. GrACE:DSM Authenticate

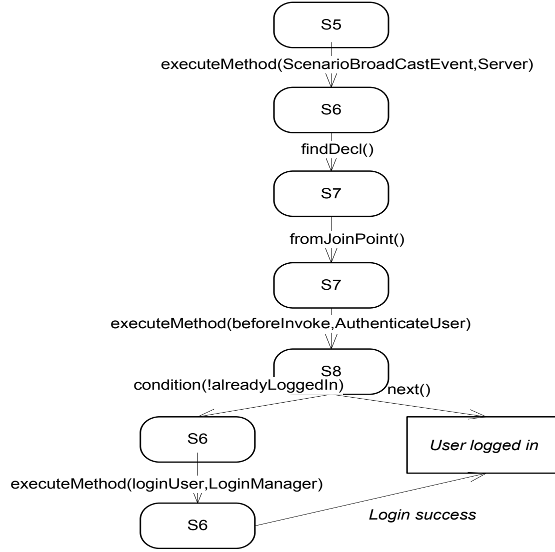


Fig. 6. GrACE:Execution tree

GrACE (Graph-based Adaptation, Configuration and Evolution) approach expresses the base and concerns models as class and sequence diagrams. The specification also contains an activator action and the execution constraint, a sequence of method invocations, expressed with Computational Tree Logic (CTL). With this input, GrACE simulates the execution and the composition of the input diagrams starting from the activator action. The end result of this is the execution tree where each branch is a possible composition showing all the methods invoked in it [5]. Then, a verification algorithm verifies whether the input execution constraints is violated or not. In case it is violated, a feedback is provided to the user. In this way, the user can verify the behavior of the concerns in the composed model. For simulation, GrACE specializes graph-based model checking by defining a model called Design Configuration Model (DCM) for representing UML based AOMs with graphs, and modeling OO- and AspectJ-like execution semantics with graph-transformation rules over this model.

GrACE uses the mapping from Theme/UML [6] to a Domain Specific Language. Hence, the concerns are modeled as "themes" in Theme/UML. Fig. 4 presents an excerpt from the sequence diagram of the theme *Authenticate*, which defines a pointcut to the template method *fireStart*. The advice for this pointcut is defined in the method *beforeInvoke()*, which checks if the user is already authenticated.

The themes are converted to DCMs for *verification*. GrACE toolset includes a prototype tool which automates the conversion from Theme/UML to DCM. Fig. 5 shows the graph-based DCM of the theme *Authenticate*. The node labeled

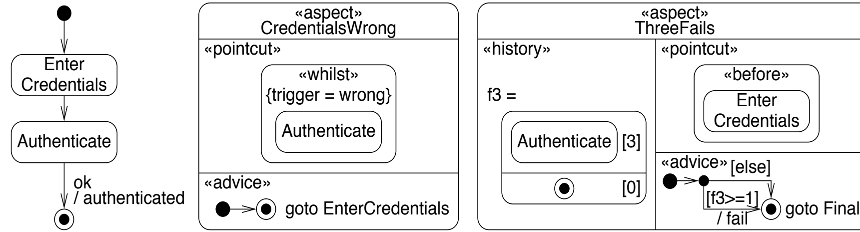


Fig. 7. HiLA model

AspectType with the attribute name *AuthenticateUser* represents the template class *AuthenticateUser*. The node with the attribute *toMethod* set to *fireStart* that is connected to the aspect type node represents the template parameter of the theme *Authenticate*.

To illustrate composition of aspects, Fig. 6 presents an excerpt of the execution tree generated from the simulation of the base model and the theme *Authenticate* shown in Fig 4. At state $S5$, this excerpt starts with the dispatch of the method *Server.ScenarioBroadCastEvent()*. In the activation bar of this method, the first action is a call action. Hence, the transformation rule *findDecl* matches at state $S6$ and identifies *ScenarioOutSideEvent.fireStart()* as the receiver of the call. Because the aspect *AuthenticateUser* defines a pointcut to this method, the transformation rule *formBeforeJoinPoint* matches and gives the execution to the advice.

2.5 Classes and State Machines

A UML Behaviour State Machine (BSM) usually presents behaviour of one classifier. Aspects extend the behaviour specified for classifiers. HiLA modifies the semantics of BSM allowing classifiers to apply additional or alternative behaviour. The High-Level Aspects (HiLA) approach [15] introduces AspectJ-like constructs into UML state machines. The basic static structure usually contains one or more classes. Each base state machine is attached to one of these classes and specifies its behavior. 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 base state machines. Dynamic (high-level) aspects only apply to state machines and specify additional or alternative behaviour to be executed at certain "appropriate" points of time in the base machines execution.

Fig. 7 presents the scenario of the *Authentication* concern. Modeling with HiLA is a top-down process. The main success scenario of a (behavioral) concern is modeled in the base machine: first the user enters his credentials (*EnterCredentials*), which are then validated (in *stateAuthenticate*). The extension for authentication failures is modeled with the pattern *whilst* (stereotype *whilst*). State *Authenticate* is active. If the current event is wrong (tagged value *"trigger*

= *wrong"*, then the advice is executed, which forces the base machine to go to state *EnterCredentials*, where the user can try again.

The history-based extension, which allows the system to accept at most three trials to log in is modeled in *Three Trials*. The history property $f3$ counts how often its pattern, which specifies continuous sequences containing three and no) final state occurrences, are contained in the execution history so far. The pointcut selects the points of time just *before* state EnterCredentials gets active. If $f3 = 1$, is satisfied, which means that the user has already tried to log in three times unsuccessfully and now tries to authenticate again, the advice takes the base machine to the final state (label *goto Final*), and ends in failure (signal *fail*). Otherwise the advice does not do anything.

Aspects are composed together by the weaving process. Additional behaviors defined by *whilst* aspects are woven as additional transitions. History-awareness is achieved by entry actions to keep track of states activation; *before* (and *after*) aspects are woven into transitions selected by the pointcut.

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.

2.6 Services

The ADORE framework[1] is an approach to support aspect-oriented business processes modeling, using the orchestration of services paradigm.

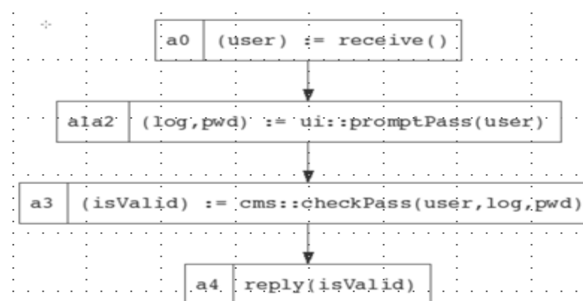


Fig. 8. Orchestration:cms:authUser

Models describing business–driven processes (abbreviated as *orchestrations*, defined as a set of partially ordered activities) are composed with process fragments (defined using the same formalism) to produce a larger process. *Fragments*

[1] *http://www.adore-design.org*

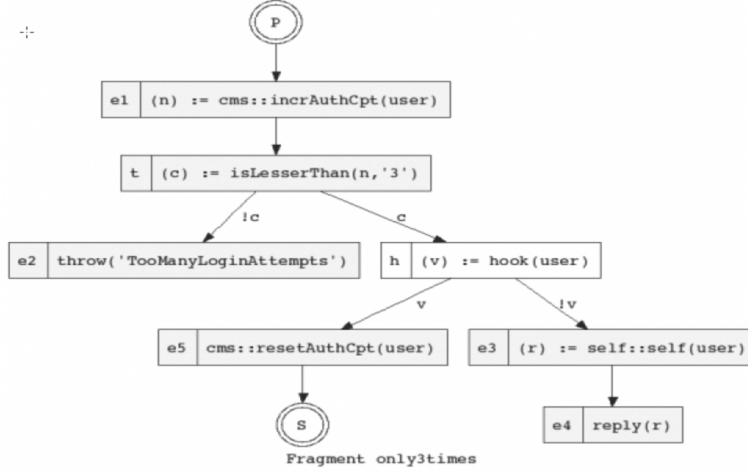


Fig. 9. Authentication concern in an ADORE model

realize models with little behavior and describe different aspects of a complex business process. ADORE allows a business expert to model these concerns separately, and use automated algorithms to compose them.

Using ADORE, designers can define *composition units* (abbreviated as *composition*) to describe the way fragments should be composed with orchestrations. The merge algorithm used to support the composition mechanism[11] computes the set of actions to be performed on the orchestration to automatically produce the composed process. When the engine detects shared join points, an automatic *merge* algorithm is triggered to build a *composed* concern. ADORE also provides a set of *logical rules to detect conflicts* inside orchestrations and fragments (*e.g.*, non-deterministic access to a variable, interface mismatch, lack of response under a given condition set).

We represent in Fig.8 the initial orchestration dealing with the *authentication* concern. It represents the base success scenario, as described in the requirements. To model blocking the user after 3 failed attempts, we use the fragment depicted in Fig.9. The composition algorithm produces the final behavior by integrating the fragment into the legacy orchestration.

2.7 Mixins

A Protocol Model [10] of a system is a set of protocol machines (PMs) that are composed to model the behavior of the system. Fig. 10 shows a protocol model of the security concern composed from PMs *Employee Main*, *Clock*, *Singleton*, *Password Handler* and *Want Time Out*.

The specification of a PM is described in a textual file as it is shown below for machine *Employee Main*.

```
BEHAVIOUR Employee Main
      ATTRIBUTES Employee Name: String, !Employee Status: String,
                (Security Password: String), Max Tries: Integer
      STATES created, deleted
      TRANSITIONS @new*!Create Employee=created,
                  created*Session Event=created,
                  created*!Set Password=created,
                  created*Log In=created, created*Log Out=created,
                  created*Time Out=created, created*Reset=created,
                  created*Delete Employee=deleted
      EVENT Create Employee
            ATTRIBUTES Employee: Employee Main,
                  Employee Name: String,Security Password: String,
                  Max Tries: Integer
      EVENT Delete Employee
            ATTRIBUTES Employee: Employee Main
      EVENT Set Password
            ATTRIBUTES Employee: Employee Main,
            Security Password: String, Max Tries: Integer
     EVENT Log In
           ATTRIBUTES Employee:: Employee Main, Password:String
     EVENT Log Out
           ATTRIBUTES Employee: Employee Main
     EVENT Time Out
           ATTRIBUTES Employee: Employee Main
     GENERIC OUT
          MATCHES Time Out, Log Out
```

The graphical presentation is a secondary artefact; it does not contain all the elements of the specification. The specification of a PM includes its *local storage* and the *alphabet* of *event types*. The local storage is represented as a set of its attributes. For example, attribute *Password Handler.Tries* for the PM *Employee Main*. Each event type is specified by metadata. For example, event *Set Password* contains attribute *Security Password: String*.

An event instance comes from the environment and it is atomic. Instances of PMs are created with happening of events. PM instances can be included into other PMs. A PM instance behaves so that it either accepts or refuses an event from its alphabet, depending on its state and the state of other included machines. Events that are not in its alphabet are ignored. To evaluate the state a machine may read, *but not alter*, the local storage of the included machines.

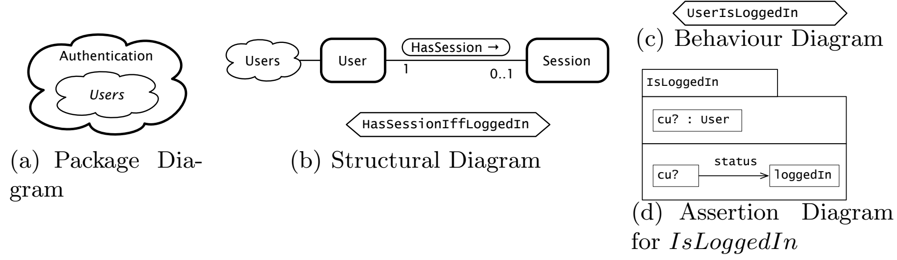
The complete system is composed using CSP parallel composition [4] extended by McNeile [10] for machines with data. This composition techniques serves as a natural weaving algorithm for aspects. The alphabet of the composed machine is the union of the alphabets of the constituent machines; and the local storage of the composed machine is the union of the local storages of the constituent machines. When presented with an event the composed machine will accept it if and only if all its constituent machines, that have this event in their alphabet, accept it. If at least one of such machines refuses the event it is refused by the composed machine. The concept of event refusal is critical to implement CSP composition for composition of protocol machines. This allows for modelling of the situation when events occur and the system cannot accept them.

Fig. 10. Protocol Model of the Authentication Concern

Join points are often events. *Password Handler* specifies join points for the *Authentication* concern. A *Session Event* is accepted if the *Password Handler* is in state *logged in*, when both machines *Employee Main* and *Password Handler* may accept it.

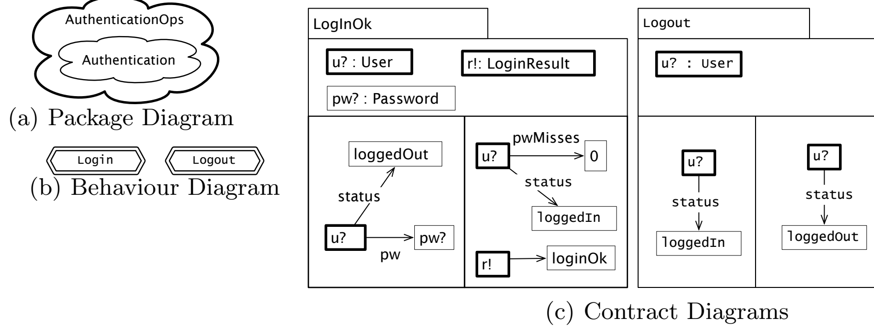
Quantification on events is defined by *generalized events*, e.g. event $Out = \{LogOut, TimeOut\}$. Quantification on states is defined as derived states. State abstractions specified with derived states are allowed to be used in join point specifications. For example, the state *logged In* is derived as *"If (Tries <= 4 && Password = Security Password)"*, and as a result the generic event *Out* representing events *Time Out* and *Log Out* becomes possible.

The CSP composition based algorithm is used for aspect weaving and *simulation.* It produces system traces from parts of traces of aspects with te Modelscope tool. This algorithm guarantees that the order of accepted events in traces of aspects is not changed in the result of their composition (see the proof in [9]). The aspect interference of composed aspects may block some traces of aspects, but it will not change the order of events in them. This property of *local reasoning* [9] provided by the PM approach prevents appearance of invasive aspects and eases the reuse and evolution of protocol models.



(a) Package Diagram (b) Structural Diagram (c) Behaviour Diagram (d) Assertion Diagram for $IsLoggedIn$

Fig. 11. VCL package *Authentication*, addressing the authentication concern



(a) Package Diagram (b) Behaviour Diagram (c) Contract Diagrams

Fig. 12. VCL package *AuthenticationOps*, addressing the authentication concern

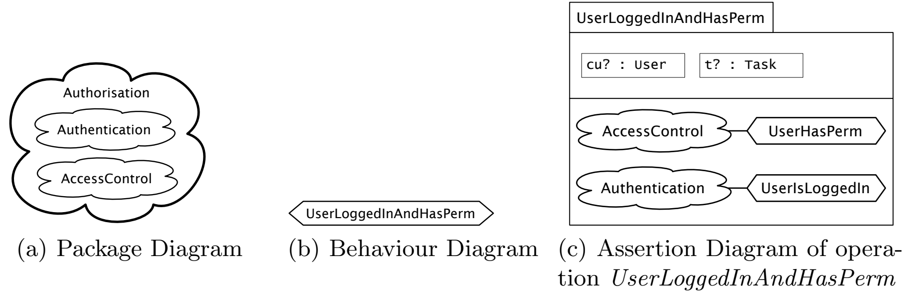
2.8 Contracts

The Visual Contract Language (VCL) [2, 3] takes an approach to behaviour modelling that is based on *design by contract*. 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.

Figure 11 presents the VCL package *Authentication*, which localizes part of the *authentication* concern. *Authentication* extends package $Users$. State structures of a package are defined in the package's *structural diagram* (SD); together they define the package's state space. The SD of package *Authentication* (Fig. 11(b)) says that a $User$ of package $Users$[2] is associated with a $Session$ through the relational-edge $HasSession$. In addition, the diagram includes an invariant $HasSessionIffLoggedIn$, stating that each session must be associated with a user that is *logged-in* [3]. Figure 11(c) gives the global behaviour diagram of package *Authentication* with the global observe operation $UserIsLoggedIn$, which says whether a user is logged-in or not; this is described using a VCL assertion diagram (Fig. 11(d)).

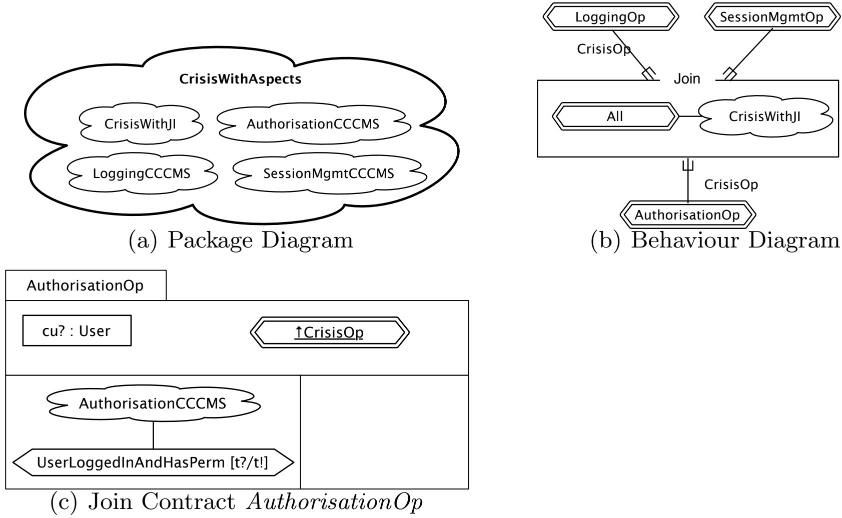
Authentication operations (*Login* and *Logout*) are defined in VCL package *AuthenticationOps*, which extends package *Authentication* (Fig. 12). Operations that perform state changes are defined in contract diagrams, composed of a pre- and a post-condition. Figure 12(c) shows two contract diagrams for blob

[2] A blob defines a set of objects.



(a) Package Diagram (b) Behaviour Diagram (c) Assertion Diagram of operation *UserLoggedInAndHasPerm*

Fig. 13. VCL package *Authorisation*



(a) Package Diagram (b) Behaviour Diagram

(c) Join Contract *AuthorisationOp*

Fig. 14. VCL package *CrisisWithAspects*

User. Operation *LoginOk* says in the pre-condition that a login is successful if the password given as input (*pw*?) matches the password of the user being authenticated (*u*?.*pw*); post-condition says that the status of the user is *loggedIn*, the number of passwords misses is 0, and the operation reports success (value *loginOk*) to its environment (output *r*!). Operation *Logout* says that provided the user status is logged in (pre-condition), then the user status is changed to logged-out (post-condition).

Figure 13 presents package *Authorisation*, which puts two aspects together: *Authentication* and *AccessControl* (see [3]). This package defines the observe operation *UserLoggedInAndHasPerm*, which checks whether a user is logged in and has the right permissions to execute some task; this puts together the observe operations *UserHasPerm* of *AccessControl* and *UserIsLoggedIn* of *Authentication*. VCL's contracts and assertions are modules that can be combined using logical operators.

Aspects are composed using *join extension*, which is illustrated in Fig. 14. In 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 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.

3 Discussion and Conclusion

Abstraction	*Localization of concerns*	*Verification*	*Localization of reasoning*
Features	VML4RE	n.a.	n.a.
Use cases	AoURN	n.a.	n.a.
Classes, sequence diagrams	RAM. GrACE	+	-
Classes, state machines	HiLA	+	-
Services, orchestration	Adore	rule based	-
Mixins	Protocol Modelling	+	+
Contracts	VCL	+	-

Table 1. AOM approaches at different levels of abstraction

Our overview shows how aspect-orientation is used at different levels of abstraction. All approaches achieved localization of concerns and better traceability of requirements in models.

Each abstraction level supports its own composition technique and this technique defines the possibilities of reasoning on models. The modelling techniques that use the same composition techniques as programs, i.e. sequential compositional composition, alternative, cycle and inheritance have an execution tree as a result of model composition and need to use verification techniques for model analysis. The modelling techniques that use the ideas of design by contract need to rely on theorem proving for system analysis. In general, the localization of reasoning on aspects cannot be achieved with these composition forms as it cannot be achieved in programs [13]. The result of composition has to be analyzed to ensure correctness of behaviour. The modelling techniques with the mixins semantics and the CSP composition (used also in some programming languages [13]) localize reasoning on aspects and objects, and the behaviour of aspects survives in the result of composition. So, the choice of composition semantics is the major challenge of the AOM research.

The models in the presented approaches show that using aspects in models always increases fragmentation of models. This simplifies model construction, but does not simplify model understanding. However, fragmentation of complex

models of real size applications is unavoidable. The experience of the presented approaches shows that any investment into tool support, allowing for search in sets of model fragments and model simulation, improves model understanding and transforms the fragmentation into an advantage.

References

1. M. Alférez, J. Santos, A. Moreira, A. Garcia, U. Kulesza, J. Araújo, and V. Amaral. Multi-View Composition Language for Software Product Line Requirements. In *2nd Int. Conference on Software Language Engineering*, Denver, USA, 2009.
2. N. Amálio and P. Kelsen. Modular Design by Contract Visually and Formally using VCL. In *VL/HCC 2010*, 2010.
3. N. Amálio, P. Kelsen, Q. Ma, and C. Glodt. Using VCL as an Aspect-Oriented Approach to Requirements Modelling. *TAOSD*, VII:151–199, 2010.
4. C.A.R.Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
5. S. Ciraci, W. K. Havinga, M. Aksit, C. M. Bockisch, and P. M. van den Broek. A Graph-Based Aspect Interference Detection Approach for UML-Based Aspect-Oriented Models. Technical Report TR-CTIT-09-39, Enschede, September 2009.
6. S. Clarke and R. J. Walker. Generic Aspect-Oriented Design with Theme/UML. In *Aspect-Oriented Software Development*, pages 425–458. Addison-Wesley, 2005.
7. J. Kienzle, W. A. Abed, and J. Klein. Aspect-Oriented Multi-View Modeling. In *AOSD 2009*, pages 87 – 98. ACM Press, March 2009.
8. J. Kienzle, N. Guelfi, and S. Mustafiz. Crisis Management Systems: a Case Study for Aspect-Oriented Modeling. *TAOSD*, 7:1–22, 2010.
9. A. McNeile and E. Roubtsova. CSP Parallel Composition of Aspect Models. In *AOM'08*, pages 13–18, 2008.
10. A. McNeile and N. Simons. Protocol Modelling. A Modelling Approach that Supports Reusable Behavioural Abstractions. *SoSyM*, 5(1):91–107, 2006.
11. S. Mosser, M. Blay-Fornarino, and M. Riveill. Web Services Orchestration Evolution: A Merge Process For Behavioral Evolution. In *2nd European Conference on Software Architecture (ECSA'08)*. Springer LNCS, Sept. 2008.
12. G. Mussbacher and D. Amyot. Extending the User Requirements Notation with Aspect-oriented Concepts. In *SDL 2009*, 2009.
13. R. Filman, T. Elrad, S. Clarke, M. Akşit. *Aspect-Oriented Software Development*. Addison-Wesley, 2004.
14. P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *ICSE'99*, 1999.
15. G. Zhang and M. Hölzl. HiLA: High-Level Aspects for UML State Machines. In S. Ghosh, editor, *Reports & Rev. Sel. Papers Wshs. at MoDELS09*, volume 6002 of *LNCS*, pages 104–118. Springer, 2010.