Using VCL as an Aspect-Oriented Approach to Requirements Modelling

Nuno Amálio, Pierre Kelsen, Qin Ma, and Christian Glodt

University of Luxembourg, 6, r. Coudenhove-Kalergi, L-1359 Luxembourg
{nuno.amalio,pierre.kelsen,qin.ma,christian.glodt}@uni.lu

Abstract. Software systems are becoming larger and more complex. By tackling the modularisation of crosscutting concerns, *aspect-orientation* draws attention to modularity as a means to address the problems of *scalability*, *complexity* and *evolution* in software systems development. Aspect-oriented modelling (AOM) applies aspect-orientation to the construction of models. Most existing AOM approaches are designed without a formal semantics, and use multi-view partial descriptions of behaviour. This paper presents an AOM approach based on the Visual Contract Language (VCL): a visual language for abstract and precise modelling, designed with a formal semantics, and comprising a novel approach to visual behavioural modelling based on *design by contract* where behavioural descriptions are total. By applying VCL to a large case study of a *car-crash crisis management system*, the paper demonstrates how modularity of VCL's constructs, at different levels of granularity, help to tackle complexity. In particular, it shows how VCL's package construct and its associated composition mechanisms are key in supporting separation of concerns, coarse-grained problem decomposition and aspect-orientation. The case study's modelling solution has a clear and well-defined modular structure; the backbone of this structure is a collection of packages encapsulating local solutions to concerns.

Key words: modularity, separation of concerns, aspect-oriented modelling, design by contract, VCL.

1 Introduction

Software systems are becoming larger, more complex and part of our everyday lives. They need to evolve in order to keep up with their complex and dynamic environments. To help reduce complexity, improve reusability, and simplify evolution, software engineering emphasises the principle of *separation of concerns* [1]. By tackling the modularisation of crosscutting concerns, *aspect-orientation* enhances traditional approaches to modularity, providing techniques to achieve designs with a good level of separation of concerns that effectively separate and isolate *non-orthogonal* (or *crosscutting*) concerns. This enables concerns to be understood and analysed in isolation, and then composed in a modular fashion. Aspect-oriented modelling (AOM) raises the level of abstraction of aspect-oriented software development by applying aspect-orientation to the construction of models of software systems.

Visual languages like UML are limited at separating concerns, not supporting concerns that are crosscutting [2]. There has been substantial work on AOM (section 11). Most existing AOM approaches: (a) extend UML to enable aspect-orientation, (b) are designed to enable code generation, (c) are based on multiple partial view descriptions of behaviour (using scenarios and state diagrams) or total descriptions based on OCL, (d) are not designed with a formal semantics (precluding, this way, formal verification), (e) are asymmetric, treating aspects differently from other modules, and (f) involve complex weaving algorithms to compose models.

This paper presents an AOM approach based on the *Visual Contract Language* (VCL) [3,4,5]. VCL is a visual language for abstract and precise modelling at the level of system requirements or high-level system designs. It embodies a novel approach to visual behavioural modelling based on *design by contract* [6] where its behavioural descriptions are total. VCL expresses operations and invariants visually; UML needs to resort to textual OCL to do this. Unlike UML and other mainstream languages, VCL is designed to have formal semantic foundations to enable formal semantic analysis. Its semantic is expressed in Z [7], using the ZOO semantic domain of object-orientation [8,9], which has been applied to UML-based models in [10,11,9]. VCL is accompanied by a tool, the Visual Contract Builder[1], which is being developed as part of the VCL effort.

VCL is novel in its modular approach to modelling based on different levels of granularity. At a more finer-grained level, VCL's contracts and constraints are modules that can be combined using logical operators. It is, however, through its coarse-grained construct of packages that VCL realises AOM. VCL packages are reusable functional units encapsulating structure and behaviour that can be used or extended by other packages. They enable the definition of modules that localise solutions to concerns. VCL is symmetric in the way it treats classical modules and aspects, not making a distinction between them. VCL packages constitute modules that can be described, understood and analysed in isolation and then used as a piece in multiple contexts to make larger packages addressing multiple concerns. In VCL, package compositions have a declarative nature, not involving complex weaving algorithms.

This paper shows how VCL tackles the complexity of large-scale systems with a large case study, the *car-crash crisis management system* (CCCMS) [12]. It illustrates VCL's package construct and associated composition mechanisms, showing how they support separation of concerns, coarse-grained problem decomposition and aspect-orientation. The resulting VCL model has a clear and well-defined modular structure; the backbone of this structure is a collection of packages encapsulating local solutions to concerns.

This paper is organised as follows. Section 2 explains the process that we followed to model the CCCMS using the VCL-based AOM approach presented here. Section 3 presents the high-level requirements model of the CCCMS. Section 4 overviews the detailed VCL system requirements model. Sections 5 to 8 build parts of the overall VCL model. Section 9 discusses the paper's results. Sec-

[1] `http://vcl.gforge.uni.lu`

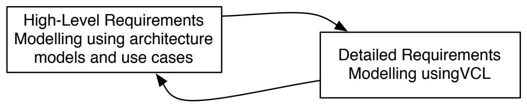

High-Level Requirements Modelling using architecture models and use cases
Detailed Requirements Modelling usingVCL

Fig. 1. Process used to model the CCCMS using VCL.

tion 10 evaluates our approach with respect to a number of qualitative criteria. Section 11 discusses related work. The final section presents the conclusions.

2 Process

VCL is a language that emphasises precision and is suited to describe requirements (or high-level design) models of software systems. The process used to model the CCCMS in our VCL-based AOM approach consists of two big steps (figure 1):

1. *High-level requirements modelling.* This identifies subsystems and their high-level functionality. It uses notations other than VCL, such as architectural diagrams, and UML use case and sequence diagrams.
2. *Detailed requirements modelling.* This builds VCL models of the identified subsystems, describing their structure and behaviour.

Figure 1 highlights the iterative nature of this process. It was necessary to iterate through these different levels of modelling; forwards and backwards. The high-level model is the basis of modelling in VCL; often, the detailed VCL model provides useful feedback to elaborate the high-level model.

Crosscutting concerns are identified through the iterative process of figure 1, being identified in both modelling stages:

– During high-level modelling, cross-cutting concerns manifest themselves by appearing repeatedly in use cases. Such concerns are then modularised as VCL packages in the detailed model.
– During VCL modelling, cross-cutting concerns emerge when we observe a behavioural pattern occurring repeatedly. In this case, it is necessary to go back and update the high-level requirements model.

The following describes these two levels of modelling in detail.

2.1 High-level requirements modelling

The high-level requirements phase consists of the following steps:

1. *Build high-level architectural models.* Architectural models describe the system's subsystems and their main units of functionality (features). This helps structuring the overall model to reflect this decomposition.

2. *Build use case models.* Main functional units (features) coming from architectural model are refined into UML use cases.
3. *Derive system operations.* System operations are derived by drawing UML sequence diagrams describing use case scenarios, highlighting the interaction between the environment and the system (see [13]). Derived system operations are then described in detail in VCL.

2.2 Detailed requirements modelling in VCL

The high-level requirements model is the basis for modelling in VCL. For each subsystem, we build a VCL model, which comprises a collection of VCL packages; one package represents the overall subsystem. The method followed to build VCL packages is as follows:

1. *Build or reuse VCL packages addressing generic concerns.* From the high-level requirements model, we derive a set of generic concerns, some of which are crosscutting. These generic concerns are modularised as VCL packages; there is at least one package for each generic concern. VCL packages should be designed to be independent in order to achieve *low-coupling.*
2. *Build VCL packages to address problem domain concerns.* Problem domain concerns (crosscutting or not) are derived from the high-level requirements model. They are modularised in VCL as packages. The aim is to model relevant fragments of the problem domain in isolation, abstracting away from other concerns of the system and trying to achieve low-coupling.
3. *Build composite packages.* Larger packages are built incrementally from the individual packages using VCL's composition mechanisms. They may represent the configuration of generic concerns, compositions to represent generic or problem domain concerns, and compositions linking domain and aspects packages. Ultimately, there is a VCL package for each subsystem.

3 High-level requirements model of CCCMS

The process highlighted above (section 2, figure 1) was used to build the VCL-based requirements model of the CCCMS [12]. The complete model is given in [14].

The following describes the high-level requirements model of CCCMS.

3.1 Subsystems and their functional features

The diagram of figure 2 describes the systems architecture of the CCCMS, highlighting a decomposition into subsystems. This reflects architectural decisions that have been taken by balancing the case study's requirements. The CCCMS's subsystems are as follows:

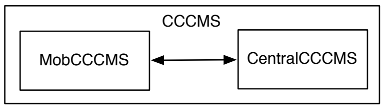
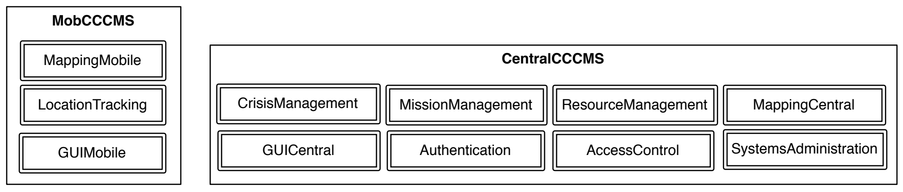


Fig. 2. Systems diagram showing subsystems of CCCMS. MobCCCMS (Mobile CCCMS) is deployed on mobile devices and used in real-time by resources sent to missions. CentralCCCMS (Central CCCMS) is used at crisis management headquarters. Subsystems exchange information through messages.



(a) *MobCCCMS* (b) *CentralCCCMS*

Fig. 3. Diagrams describing features of the CCCMS's subsystems.

- MobCCCMS is a system that runs on mobile devices to assist in real-time resources deployed to rescue missions. It addresses the *mobility* non-functional requirement (NFR) (see [12]). This consists of user interfaces to provide resources with sensible information for the execution of rescue missions.
- CentralCCCMS is the system used at the crisis management control-centre by crisis coordinators. It addresses the persistence NFR of [12]; all data related with domain functionality is held in this subsystem.

The high-level functional units (features) of the CCCMS's subsystems are identified in the block diagrams of figure 3; each block represents a feature. Table 1 describes these features indicating the requirements they address.

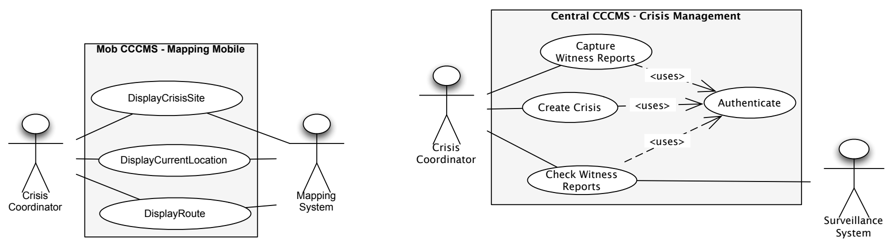
3.2 Use cases and system sequence diagrams

Use cases were organised around the subsystems and their features. Each subsystem comprises a set of use cases. The use case model of [12] was re-factored to take subsystems and their features into account, to clarify some omissions and ambiguities that we found in the requirements, and to enable a clear derivation of system operations from the use cases.

Figure 4 presents sample use case diagrams belonging to the use case model of CCCMS [14] for the features MappingMobile and CrisisManagement. In [14], the scenarios of each use case are described using UML sequence diagrams to derive system operations. Figure 5 presents sample system sequence diagrams for use cases of feature CrisisManagement. All messages going from external actors into the system, identify system operations. System sequence diagrams of

Subsystem	**Feature**	**Description**	**Requirements**
MobCCCMS	Mapping Mobile	Handling of maps on mobile devices.	Mobility NFR of [12].
MobCCCMS	Location Tracking	Tracking location of resources.	Mobility NFR of [12].
MobCCCMS	*GUIMobile*	Displays information for resources deployed to missions.	Accuracy NFR of [12].
CentralCCCMS	Crisis Management	Management of crisis.	Use cases of [12].
CentralCCCMS	Resource Management	Management of resources.	Use cases of [12].
CentralCCCMS	Mission Management	Management of missions.	Use cases of [12].
CentralCCCMS	Mapping	Handling of maps in control centre.	Accuracy NFR of [12].
CentralCCCMS	*GUICentral*	Graphical user interfaces for crisis coordinators.	Use cases of [12].
CentralCCCMS	Authentication	Authentication of users.	Security NFR of [12].
CentralCCCMS	Access Control	Access-control security policies.	Security NFR of [12].
CentralCCCMS	Systems Administration	Tasks related to administration of users and access control policies.	Use cases of [12].

Table 1. Features of CCCMS's subsystems, and their requirements.



(a) *MappingMobile* feature (b) *CrisisManagement* feature

Fig. 4. Sample use case diagrams of *CCCMS*.

figure 5 highlight crosscutting functionality; in this case the functionality related to authentication and the *Login* system operation.

4 Detailed VCL model

The process described in section 2.2 is used to build detailed VCL models for subsystems MobCCCMS and CentralCCCMS, taking into account all features and their system operations identified in the high-level requirements model (section 3).

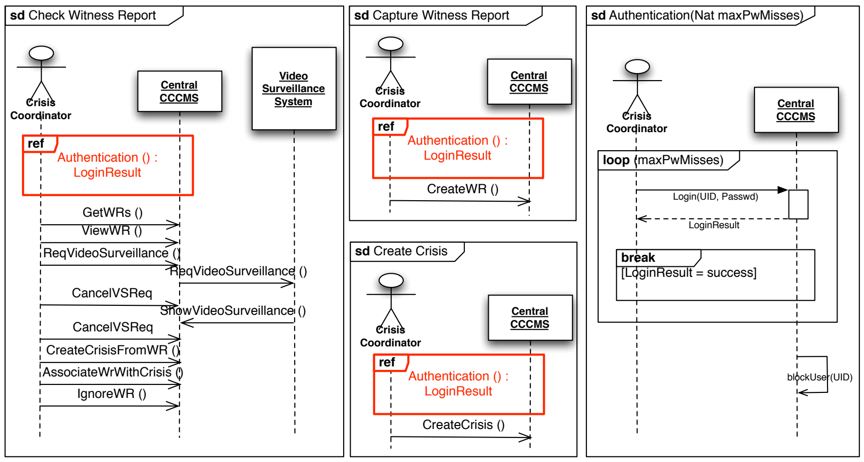


Fig. 5. Sequence diagrams describing scenarios of CrisisManagement feature. Diagrams highlight (in red) *authentication* crosscutting functionality.

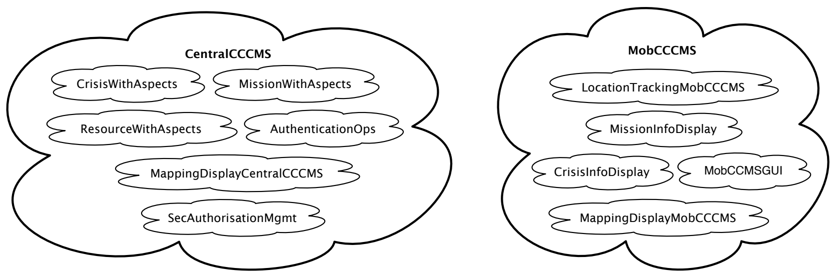


Fig. 6. Package diagrams defining packages representing CentralCCCMS (left) and MobCCCMS (right) subsystems.

Each subsystem is represented as a VCL package. A subsystem package is built by incorporating packages representing the subsystem's concerns. Figure 6 presents VCL package diagrams for subsystems CentralCCCMS and MobCCCMS. In VCL, packages are represented as *clouds* to allude to the fact that they define a world of their own. A VCL package diagram highlights the package being defined (in bold) and the packages being extended. Package extension means incorporation; state structures and operations defined in the incorporated package become part of the composite package[2].

[2] Semantically, incorporation means conjunction; structures of composite are those of packages it incorporates, plus those that composite defines as its own.

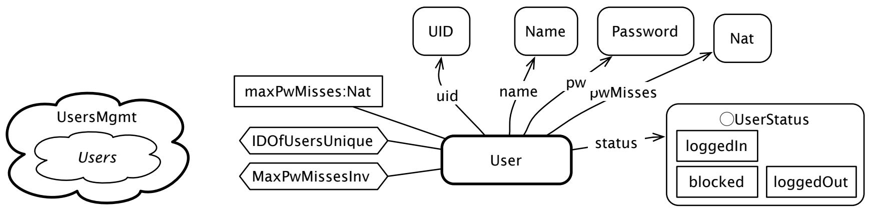


Fig. 7. Package diagram of package UsersMgmt, which extends Users (left). Structural diagram of package Users defining User blob (right).

Package diagrams of figure 6 highlight constituent individual VCL packages that are also, themselves, composite packages. All VCL packages of the overall model of CCCMS are defined in [14]. Each subsystem has its own VCL model with its set of constituent packages, some of them common to both subsystems.

The next sections present fragments of the overall VCL model to illustrate the process described in section 2.2. They highlight VCL's capabilities to support abstract modelling in an aspect-oriented way, illustrating VCL and its package construct in their capability to capture generic concerns, support flexible problem decomposition, and enable a *plug-and-play* style of modular composition where modules are plugged to make a whole. The following sections show how VCL packages modularise generic concerns (section 5), how VCL packages are used to customise generic packages to some context (section 6), how VCL packages modularise concerns of the problem domain, which can either be crosscutting or not (section 7), and how VCL packages can be composed to make larger packages, ultimately to arrive at a package of a subsystem (section 8).

5 Packages that localise generic concerns

VCL packages can modularise solutions to generic concerns to enable their use in various settings. They constitute a functional unit that can be reused and has state of its own; each package comprises a definition of structure (described using structural and constraint diagrams) and behaviour (described using contracts).

The following shows the VCL modularisation of the following generic crosscutting concerns: *authentication of users*, *access control*, *management of session activity*, *security management*, *logging*, *mapping* and *video-surveillance*. In [14] we also address the *location tracking* concern.

To illustrate VCL's formal Z semantics, we provide in [14] the Z representation of some of the packages developed in the next sections.

5.1 Users

Common to both *authentication* and *access control*, are concerns related with *users*. This section defines packages to represent and manage users.

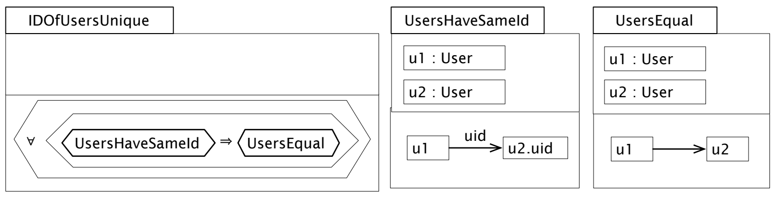


Fig. 8. Constraint diagram defining constraint IDOfUsersUnique.

Package definitions. To represent user information and manage users, we introduce packages Users and UsersMgmt; both describe user-related concerns. Figure 7 (left) defines package UsersMgmt, which extends Users; this means that the former incorporates the latter and adds something of its own. The Z model resulting from these VCL packages is given in [14].

Structure. Packages encapsulate structures, which are defined in the package's VCL structural diagram (SD). SDs' main construct is the *blob*, a *rounded contour* denoting a *set*, that represents some system entity. There are two types of blobs: *domain* (bold line) and *value* (normal line). Domain blobs are part of the state of the overall system; they are dynamic and need to be maintained by the system. Value blobs define an immutable set of values that do not need to be maintained by the system. To represent objects (members of some blob) VCL uses *rectangles.*

Figure 7 presents SD of package Users. Domain blob User represents users of a system; it is to be used by other packages requiring user-related functionality. Value blob UserStatus defines a set by enumerating its elements inside its contour (symbol ◯ says that a blob is defined by what it encloses); it says UserStatus comprises distinct elements named loggedIn, loggedOut and blocked. The labelled arrows emanating from User are called *property edges*; they define properties possessed by all elements of the set. User objects have a user identifier (uid), the actual name of the user (name), a password (pw), a record of the number of password misses (pwMisses) kept for security reasons, and a *login* status (status), representing the fact that a user may be *logged-out*, *logged-in* or *blocked* because the number of password tries exceeded the allowed maximum. The object (rectangle) linked to User defines a local constant visible only in the scope of this blob; maxPwMisses of blob Nat (natural numbers) represents the maximum number of allowed consecutive password misses.

In VCL, *elongated hexagons* represent *constraints.* The one connected to User defines a local invariant (in OO terms a class invariant), which restricts the number of valid instances of this blob. Invariant IDOfUsersUnique is defined in the *constraint diagram* of figure 8, which expresses graphically, in a style akin to the *predicate-calculus* (see [5,3] for details), that if two users have the same id, then they must be the same user.

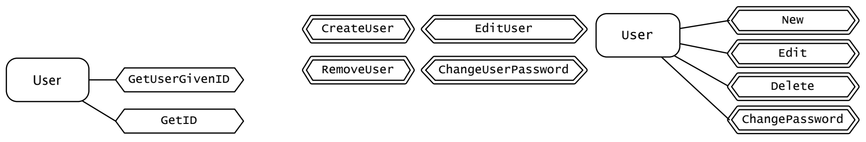


Fig. 9. Behavioural diagrams of package Users (left) and UsersMgmt (right).

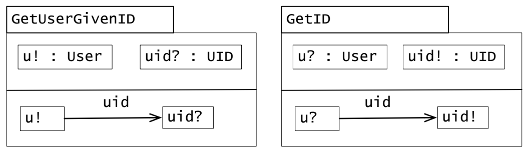


Fig. 10. Constraint diagrams defining local observe operations GetUserGivenID and GetID of blob User in package Users.

Behaviour. VCL's unit of behaviour is the *operation*. VCL packages comprise a collection of structures, such as blobs and property edges; operations manipulate the information stored in these structures. Contracts define operations, describing what they must do. In VCL, operations may be *local* or *global*. They are local when they describe the internal behaviour of a single structure. A global operation describes the collective behaviour of a collection of structures. The global operations of a package define the behaviour that the package offers to the outside world.

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. Query operations observe some state of the system and they involve a single state. They are defined in constraint diagrams, differing from ordinary constraints in that they return values (the observations). In BDs, update operations are represented as *contracts* (*double-lined elongated hexagons* labelled with the name of the operation); observe operations are represented as *constraints* (single-lined elongated hexagons); in VCL's tool, double-clicking on operations represented in a BD takes the user to their definition. Local operations are connected to the structure whose behaviour they operate upon. Global operations stand alone.

Figure 9 presents BDs of package Users (left) and UsersMgmt (right). BD of package Users introduces observe operations GetUserGivenID and GetID of blob User, which yield, respectively, a user object given a user identifier and a user identifier from a user object. Package UsersMgmt defines operations for managing users; it introduces global operations CreateUser, EditUser, RemoveUser and ChangeUserPassword, and local operations of blob User.

Because they involve a single state, query operations are defined using constraint diagrams. Constraint diagrams of figure 10 define observe operations Ge-

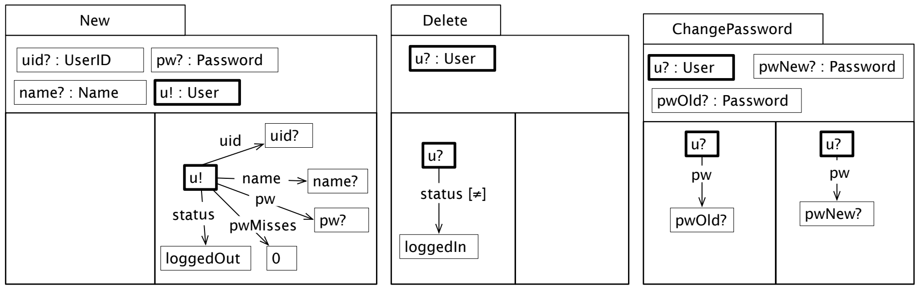


Fig. 11. Contract diagrams defining local operations New, Delete and ChangePassword of blob User.

tUserGivenID (left) and GetID (right) of blob User. They differ from the normal constraint of figure 8 in that they output a value (the observation). GetUser-GivenID receives a user id as input (uid?) and outputs the corresponding User object (u!).[3] In VCL, inputs are decorated with ?; outputs with !. GetID receives a user object as input (u?) and outputs a user id (uid!).

VCL contract diagrams describe update operations. They have a name, a declarations compartment, and a predicate compartment sub-divided into pre- and post- condition compartments. Predicate compartments have a differential meaning regarding an active unit (object, link or blob), which is represented in bold: (a) an active unit on the left (precondition), but not on the right means deletion; (b) an active unit that is on the right compartment (postcondition), but not on the left means creation; (c) the state of an active unit is updated if it is both on left and right compartments.

Figure 11 presents contract diagrams for local operations New, Delete and ChangePassword of blob User. They are as follows:

– New receives as inputs a user id (uid?), a password (pw?), and an actual name of a user (name), and assigns these to the properties of the newly created User object (u!) (an active object) in the postcondition compartment; u! is an output of the contract.
– Delete receives the User object to delete (input u?), and states as a precondition that u? may be deleted provided its status is not logged-in.
– ChangePassword takes as inputs a user (u?), old password (pwOld?) and new password (pwNew?). Precondition requires that user's new password matches old password. Postcondition sets user's password (pw) to new password.

Global operations incorporate (or extend) local ones, and they define some extra behaviour of their own (in the form of pre- and post- conditions). This form of contract composition is achieved through *contract importing*. A contract

[3] In VCL contract and constraint diagrams, objects say the set to which they belong; such sets must be visible in the package of the contract or constraint.

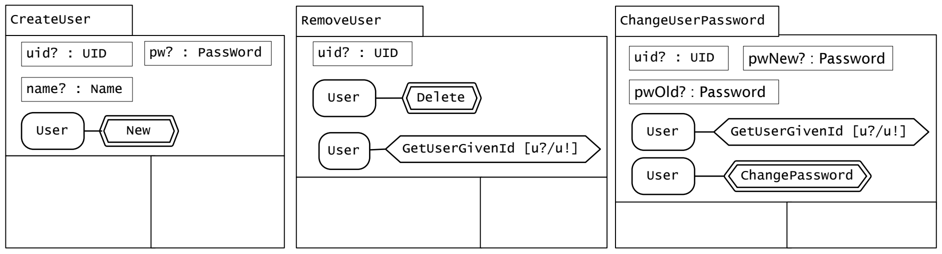


Fig. 12. Contract diagrams defining global operations CreateUSer, RemoveUser and ChangeUserPassword of package UsersMgmt.

placed on the declarations compartment means that it is being imported. Figure 12 presents contract diagrams for global operations CreateUser, RemoveUser and ChangeUserPassword. These contracts import, respectively, local operations New, Delete and ChangePassword of blob User.

The meaning of contract importing is *conjunction*. When a contract imports other contracts, this means that the overall contract is formed as the conjunction of all imported contracts plus the part that the importer defines as its own; pre- and post-conditions of imported contracts are conjoined with pre- and post-conditions of importer. There is no implicit or required ordering in the conjunctions underlying contract importing (in logic, conjunction is commutative). VCL operations specify a computation, when operations are conjoined that means that corresponding computations are performed in parallel synchronised on the communication channels being shared. The actual sequential ordering of operations is a decision to be taken at lower levels of abstraction, such as implementation.

Contracts may import constraints as well as other contracts. A constraint is imported when placed on the declarations compartment; in this case, the constraint's state refers to the contract's before-state (the precondition). Usually, we place query operations in the declarations part to convey the fact that they are operations, and because they usually operate on the before state. We place constraints describing pre- or post-conditions in the appropriate predicate compartment.

In contracts, inputs and outputs are communication channels; those having the same name in importer and imported contracts are shared. Sharing means that whatever goes through the channel in composite also goes through the channel with same name in parts; whenever channel names are shared, the bindings involved in the communication do not have to be made explicit. In figure 12, inputs defined in contract CreateUser are shared with imported contract User.New of figure 11. When a communication channel (input or output) is declared in imported contract, but not in composite that means that declared input is existentially quantified in composite contract and is not made available to the outside world. In contract CreateUser, output u! of User.New is existentially quantified.

Contracts of figure 12 are as follows:

- CreateUser takes inputs uid?, name? and pw? corresponding to inputs with same name in operation User.New (see figure 11). Output u! of User.New is existentially quantified in CreateUser. Pre- and post- conditions are those of User.New.
- RemoveUser takes as input a user id (uid?) of user to delete. The declarations compartment imports query operation GetUserGivenId (defined in package Users, see [14]) and the local contract Delete of blob User. The importing of the observe operation includes a renaming expression; the output u! is renamed to u? to enable synchronisation with operation Delete, which uses this input. As input u? is not explicitly declared, it is existentially quantified. Precondition is predicate of GetUserGivenId conjoined with precondition of Delete; postcondition is that of Delete.
- ChangeUserPassword takes as inputs a user id (uid?), old password (pwOld?) and new password (pwNew?). The last two inputs match those of imported contract ChangePassword of User. Imported query GetUserGivenId includes a rename expression to enable synchronisation with ChangePassword. Precondition is predicate GetUserGivenId conjoined with ChangePassword's precondition; postcondition is that of ChangePassword.

5.2 Authentication

The authentication security concern deals with authentication of users to enable them to gain access to the system's resources. The VCL packages addressing this concern provide a solution based on password control; they are as follows:

- Authentication represents the core of authentication; it constitutes an aspect.
- AuthenticationOps includes authentication operations, enabling users to login and logout from a system.
- AuthenticationMgmt includes operations to enable management and administration of authentication.

The Z model resulting from these VCL packages is given in [14].

Package Authentication. It modularises the core of a general solution to the concern of user authentication. This package focuses on structure; it is to be extended by other packages to provide authentication-related functionality. Package Authentication (figure 13, left) *extends* package Users.

To avoid clutter and improve usability of SDs, a system of views (part of the design of VCL's tool) is provided. SDs have a *global* and a *local* view. The global view highlights a package's main entities and the relations that exist between them; the local view highlights the details of some blob. Figure 13 (centre) presents the global view of Authentication package's SD.

SD of figure 13 refers to blob User of Users and introduces domain blob Session (set of sessions users can open in the system). In SDs, *Relational edges* or *associations* are labelled directed lines connecting pairs of blobs (direction is indicated by arrow symbol). They describe relations between concepts, denoting

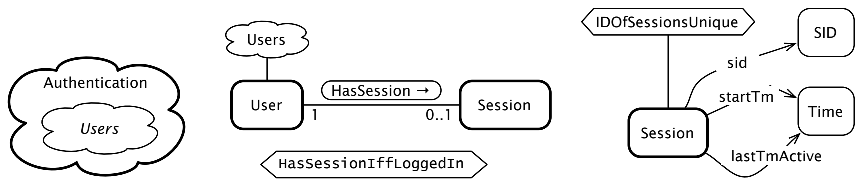


Fig. 13. Package Authentication extends Users (left). Global structural diagram of Authentication (centre). Local structural diagram for blob Session (right).

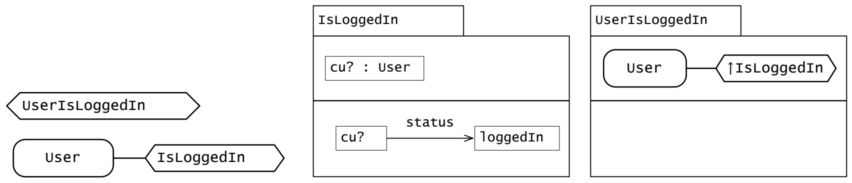


Fig. 14. Behavioural diagram of Authentication (left). Contract diagrams defining observe operations IsLoggedIn of blob User (centre), and global UserIsLoggedIn of package Authentication (right).

a mathematical relation between sets. In the diagram, HasSession is a *relational edge* between User and Session; its UML-style multiplicity constraint says that a user has at most one session and that a session has one user.

In VCL SDs, zooming yields the local details of blobs (their *local view*). Figure 13 (right) presents Session's local SD. Session objects have a session identifier (sid); they record their starting time (startTm) and the last time they were active (lastTmActive). Constraint IDOfSessionsUnique represents Session's local invariant requiring uniqueness of sids (see [14]); its definition is similar to the constraint diagram of figure 8, which requires uniqueness of User identifiers.

Authentication's BD (figure 14, left) introduces observe operations UserIsLoggedIn (global) and local IsLoggedIn of blob User, where the former promotes the latter. These operations check whether some user is logged or not; they are defined in figure 14 (centre, and right).

Package AuthenticationOps. It provides login and logout operations. Its package diagram (figure 15, left) says that it extends Authentication. Its SD (figure 15, centre) introduces value blob LoginResult, representing the set of values to be output as a result of a Login operation.

As SDs, BDs are articulated with a zooming system based on views to reduce clutter. The local view shows local behaviour of a particular modelling element; the global view highlights the package's overall behaviour.

Figure 15 (right) presents global BD of package AuthenticationOps, identifying the following operations:

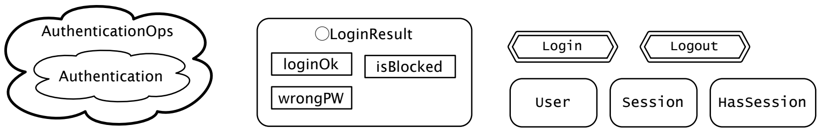


Fig. 15. Package AuthenticationOps extends Authentication (left). SD of AuthenticationOps introduces blob LoginResult (centre). Global view of AuthenticationOps package's behaviour diagram (right).

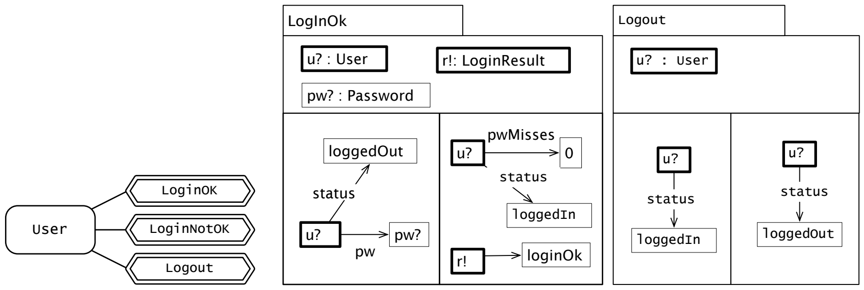


Fig. 16. Local behaviour diagram of blob User in package AuthenticationOps (left). Contract diagrams of User operations LoginOk and Logout.

– Login: it authenticates users granting them access to the system by opening a system session.
– Logout: it is used by users to terminate their system sessions.

Local BD of User in package AuthenticationOps is given in figure 16 (left). This introduces operations related with login and logout from the local perspective of blob User. Operation LoginOk (figure 16, centre) describes a successful login. Operation Logout (figure 16, right) describes a logout. Operation loginNotOk (figure 17) describes all cases of an unsuccessful login.

Local BDs of blob Session and relational edge HasSession are described in detail in [14]. Remaining contracts of package AuthenticationOps, local and global, are also described in detail in [14].

Package AuthenticationOps is to be incorporated by CentralCCCMS package to provide user authentication services (see figure 6).

Package AuthenticationMgmt. It introduces administration operations of authentication. It is built from package Authentication (figure 18, left).

Its BD (figure 18, right) introduces global operations ReactivateUser and BlockUser (see [14] for their definitions).

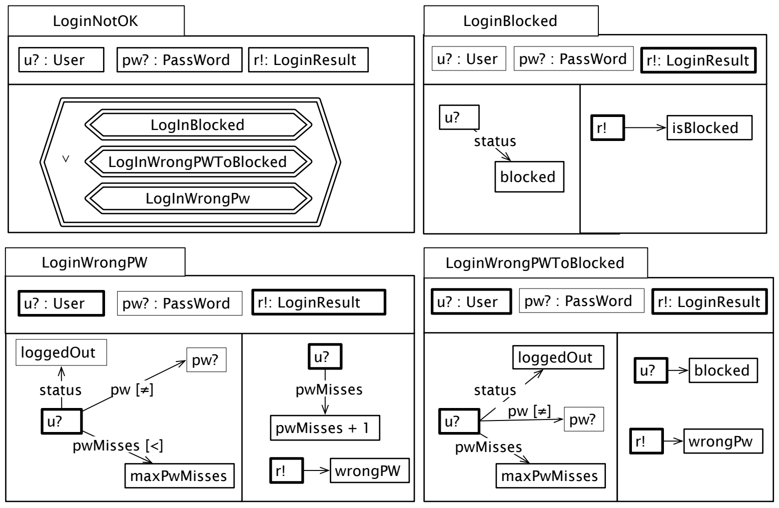


Fig. 17. Contract diagrams describing operation LoginNotOk of blob User.

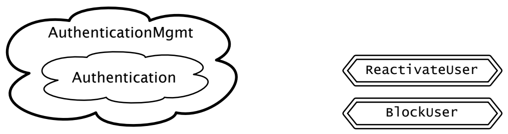


Fig. 18. Package AuthenticationMgmt extends Authentication (left). Global behavioural diagram of package AuthenticationMgmt (right).

5.3 Session Management

The session management concern deals with the management of the activity of user sessions. It is addressed by package SessionMgmt (figure 19, left), which extends package AuthenticationOps.

This package's SD (figure 19, centre) introduces constant maxInactivityTm, which captures the maximum period of inactivity that some session may reach. Its BD (figure 19, right) introduces global operation UpdSessionActivity, which says that there has been activity in some user session.[4]

Figure 20 defines the local operations of blob Session:

- UpdActiveTm is used to say that there has been activity in some user session. It takes as inputs a session object (s?) and current time (now?); postcondition compartment sets property lastActiveTm of object s? to current time.
- IsExpired says whether some session has expired or not. It takes as inputs a session (s?) and a current time (now?). The predicate compartment defines

[4] Note that SessionMgmt does not make available to the outside world the global operations of package Session; it merely uses those operations internally.

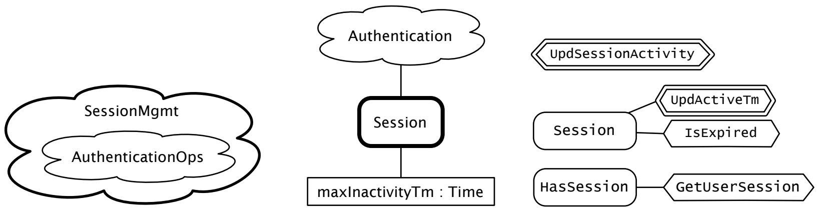


Fig. 19. Package diagram showing package SessionMgmt extends AuthenticationOps (left). Structural diagram showing package SessionMgmt introduces constant maxInactivityTm to blob Session defined in Authentication package (centre). SessionMgmt's Behavioural diagram (right).

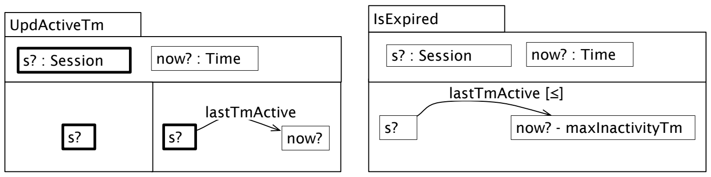


Fig. 20. Contracts diagram of local operation UpdActiveTm (left) and constraint diagram for query IsExpired (right).

constraint by saying that the last time a session was updated (property lastActiveTm) must be less or equal than the current time subtracted by the maximum time of inactivity (constant maxInactivityTm).

Figure 21 defines UpdSessionActivity, a global (or package) operation. It identifies two possible cases: (a) session has not expired and the last time active needs to be updated (UpdSessionActivityOk), or (b) the session has expired and therefore needs to be terminated (UpdSessionActivityNotOk). In UpdSessionActivityOk, note the use of $\uparrow$ symbol; this imports the communications channels (inputs and outputs), as well as the predicate part of the contract (by default contract importing just imports the predicate). Contracts are as follows:

– UpdSessionActivity takes as inputs a current user (cu?) and a current time (now?); it imports query HasSession.GetSession to get session of current user, which is used by the operations combined in the disjunction.
– UpdSessionActivityOk imports contract Session.UpdActiveTm to update last active time of session. The precondition requires that the current session has not expired (constraint Session.IsExpired is negated).
– UpdSessionActivityNotOk takes a session (s?), current time (now?) and current user (cu?) as inputs. It imports query User.GetID to get the user identifier of current user (cu?). The precondition says that the session must have expired (constraint IsExpired). Contract Logout of package AuthenticationOps is imported, so that current user is logged out and the session is deleted.

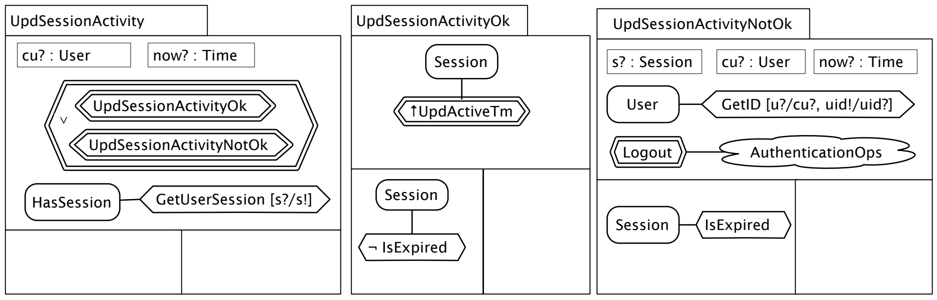


Fig. 21. Contracts diagrams of global operation UpdSessionActivity.

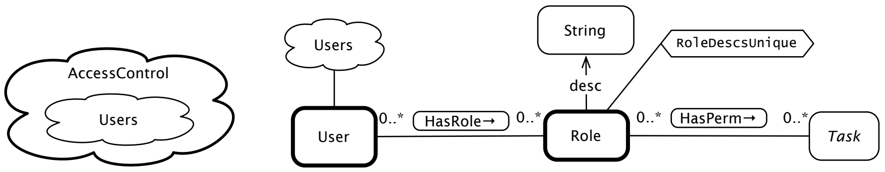


Fig. 22. Package AccessControl extends Users (left). Structural diagram for the AccessControl package (right).

Package SessionMgmt defines an aspect that is to be added to the problem domain packages so that the system takes the crosscutting functionality that it embodies into account. This involves customising the package for the CCCMS context (section 6.3) and then compose the customisation of this aspect into domain packages to enable its crosscutting functionality (section 8).

5.4 Access Control

Package Authentication (section 5.2) encapsulates a simple security mechanism of user authentication: users authenticate themselves to gain access to system's resources. In most cases, however, not all users have the same permissions in terms of the system resources they can use. This section addresses this concern by providing packages with solutions based on the rôle-based access control (RBAC) [15] scheme. This is based on the rôles that users hold within an organisation; access permissions are defined at the level of rôles.

The Z representation of the access control packages presented in the next sections is given in [14].

Package AccessControl. It describes a simple RBAC scheme, introducing the notion of *access control policies* to define who is authorised to use certain system's resources. Package AccessControl (Fig 22, left) extends package Users (section 5.1).

Figure 22 (right) presents the VCL SD of this package. It is as follows:

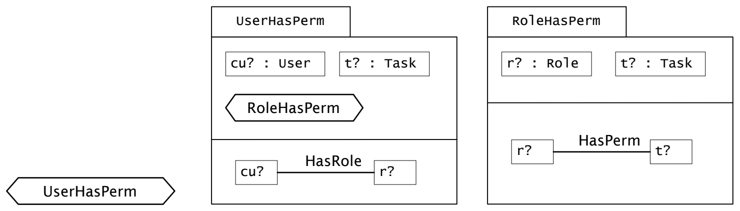


Fig. 23. Behavioural diagram of package AccessControl (left). Constraint diagram defining query UserHasPerm of package AccessControl (right).

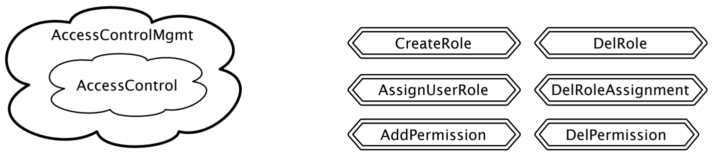


Fig. 24. Package AccessControlMgmt extends AccessControl (left). Global behavioural diagram of package AccessControlMgmt (right).

- Domain blob Role represents a set of rôles; a Role has a description that identifies it. Abstract blob Task defines the set of tasks that can be performed in a system; it defines the resources that a system offers. Task is abstract because the actual set is dependent on the application domain; AccessControl is meant to be domain-independent.
- Relational edge HasPerm defines permissions, saying which rôles are allowed to execute tasks; a Task can be executed by many Roles, and a Role may execute many tasks. Relational edge HasRole indicates rôles a user can play; a User may play many rôles; a Role may be played by many users.
- Invariant RoleDescsUnique says that rôle descriptions assigned to roles are unique. See [14] for its VCL definition.

Figure 23 (left) presents BD of package AccessControl. Observe operation UserHasPerm says whether a user is allowed to execute a given task; it is defined in constraint diagram of figure 23 (right).

Package AccessControlMgmt. This package extends AccessControl (Fig 24, left). It provides operations to create and delete rôles, permissions, and rôle assignments, as shown by its global BD presented also in figure 24 (right). See [14] for full definitions of behaviour provided by this package.

5.5 Authenticated Access Control: composing aspects

Package Authentication (section 5.2) provides a general user authentication scheme. AccessControl (section 5.4) provides a general rôle-based access control scheme.

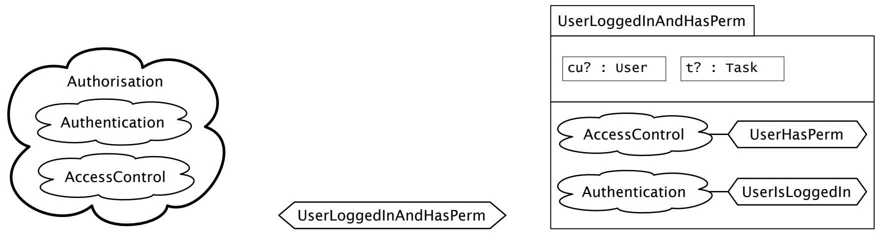


Fig. 25. Package Authorisation extends Authentication and AccessControl (left). Authorisation's behavioural diagram (centre). Constraint diagram defining behavioural constraint UserIsLoggedInAndHasPerm.

We put these two packages together in a single package to provide a general solution to authenticated rôle-based access control.

Package Authorisation (figure 25, left) extends both Authentication and AccessControl. It provides the functionality of incorporated packages, plus some extra behaviour of its own. Its BD (figure 25, centre) introduces observe operation UserLoggedInAndHasPerm, which checks if some user is logged in the system and has permission to execute some task. This query is defined in constraint diagram of figure 25 (right); it puts together two observe operations.

Package Authorisation constitutes an aspect to be joined with problem domain packages so that the functionality it embodies is part of the overall system. This generic package is customised to the CCCMS context in section 6.1; the customisation is then composed with problem domain packages in section 8. The Z model that defines this VCL package is given in [14].

5.6 System administration and security management

The administration and management operations of packages UsersMgmt, AuthenticationMgmt and AccessControlMgmt need to be performed in a secure way. These packages are not secure because they allow any user to add or delete users, rôles and permissions. This section deals with this problem by introducing a package that localises the system administration concern, and then adds this crosscutting concern to the various packages to make them secure.

Package SysAdmin (figure 26, left) extends the package Users to enable system administration by a restricted set of users. SD of this package (figure 26, centre) introduces blob UserAdmin, which is defined as a subset of blob User from package Users. UserAdmin represents users that are system administrators and have authority to perform system related tasks. This package constitutes an aspect that is to be added to the management and administration packages.

Package SecAuthorisationMgmt (figure 26, right) provides a secure way of managing users, rôles, permissions and rôle assignments. It extends packages UserMgmt, AuthenticationMgmt, AccessControlMgmt and SysAdmin (see above). The Z model resulting from this VCL package is given in [14].

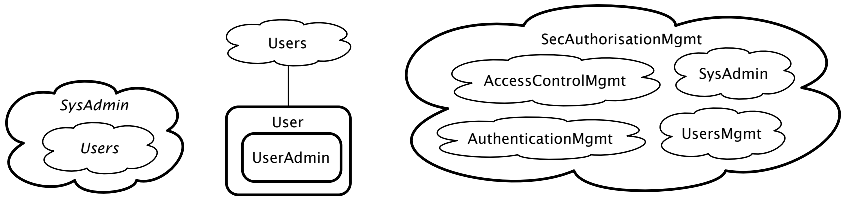


Fig. 26. Package SysAdmin (system administration) extends package Users (left). SD of package SysAdmin (centre). Package SecAuthorisationMgmt (secure authorisation management) extends packages UsersMgmt, AuthenticationMgmt, AccessControlMgmt and SysAdmin (right).

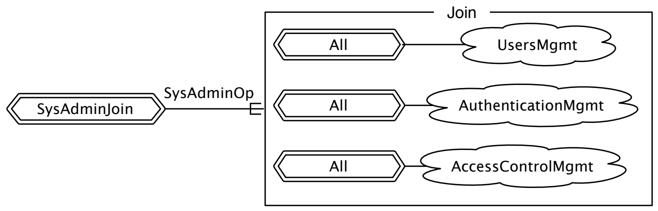


Fig. 27. Behavioural diagram of package SecAuthorisationMgmt. Join extension is used to add secure system administration operations.

Figure 27 presents the BD of package SecAuthorisationMgmt. This illustrates the *join extension* mechanism, which is used here to weave in the system administration aspect. 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 on a *join-box*. In figure 27, the join contract is SysAdminJoin, which is joined with the group of operations within the box, comprising all operations from UsersMgmt, AuthenticationMgmt and AccessControlMgmt (contracts named with keyword *All* refer to all global operations of a package). Label SysAdminOp on the line from SysAdminJoin to the box, names the interface operation used in SysAdminJoin.

Semantics of join extension is conjunction. It means that each operation from the group placed in the join-box is conjoined with the join contract. In the join-extension of figure 27, each operation of the group is conjoined with SysAdminJoin to make a new operation named after the original operation whose behaviour is extended.

Figure 28 presents the contract diagram of join contract SysAdminJoin (left) and the result of join composition for operation CreateUser of package UsersMgmt (right). These are as follows:

– SysAdminJoin declares one input to represent current user (cu?), and imports the contract to which the join is to apply along with its inputs (symbol $\uparrow$); this contract is represented by the name SysAdminOp. The precondition re-

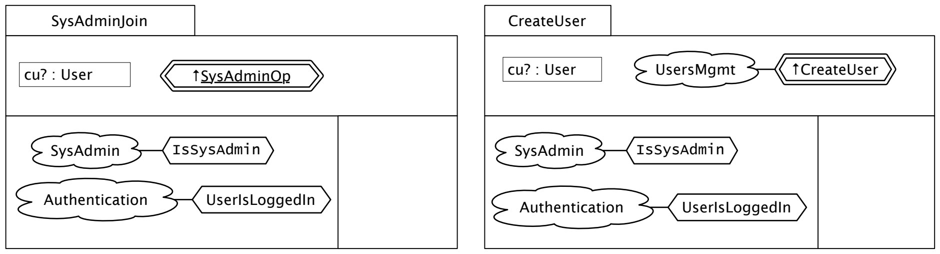


Fig. 28. Contract diagram of join operation SysAdminJoin in package SecAuthorisationMgmt (left) and result of join extension for operation CreateUser of package UsersMgmt.

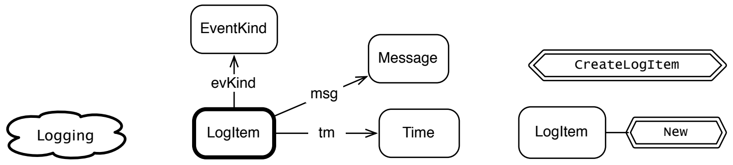


Fig. 29. Package Logging (left), its SD (centre) and its BD (right). SD defines domain blob LogItem. BD introduces global operation CreateItem and local operation New of blob LogItem.

quires that the current user is a system administrator that is logged in the system; this is expressed by placing the constraints from packages Authentication and SysAdmin in the precondition compartment.

- CreateUser mimics what is defined in contract SysAdminJoin: input cu?, imported contract, in this case CreateUser from package UsersMgmt, and statement of precondition.

Package SecAuthorisationMgmt constitutes a module that is to be incorporated by the CentralCCCMS package to provide a secure way of managing security-critical information (see figure 6, left).

5.7 Logging

Package Logging (figure 29, left) defines a general solution to the logging concern. It logs the activity of a system by recording events as they are executed.

Figure 29 (centre) defines Logging's SD. Domain blob LogItem represents events to be logged; LogItems have a creation time (tm), a logging message (msg) and the type of event being logged (evKind).

Figure 29 (right) presents the BD of package Logging. This introduces the global (or package) operation CreateLogItem, and the New local operation of LogItem.

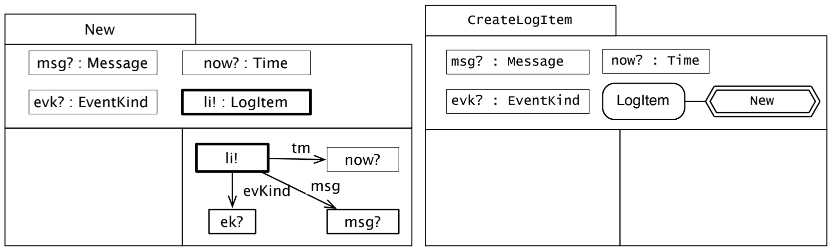


Fig. 30. Contract diagram for local operation New of blob LogItem (left) and global operation CreateLogItem (right) in Package Logging.

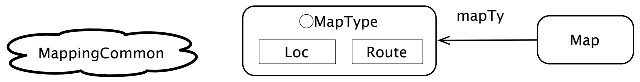


Fig. 31. Package (left) and structural (right) diagrams of package MappingCommon.

Figure 30 gives contract diagrams for these operations. They are as follows:

- New receives as inputs a logging message (msg?), a logging event (evk?), and a current time (now?), outputting the newly created LogItem object (li!); the postcondition compartment sets attributes of object li! from given input objects.
- CreateLogItem defines the same inputs as imported local contract New. This establishes that those inputs are shared communication channels between these two contracts; as they are defined in the global contract, those inputs are to be provided by the package's environment.

Package Logging constitutes an aspect that needs to be added to problem domain packages so that the crosscutting functionality that it embodies is part of the overall system. This package is customised to the CCCMS context in section 6.2 and the customisation package is composed with problem domain packages in section 8.

5.8 Mapping

The *mapping* concern represents functionality related with the handling of maps. Our model considers that there is an external mapping system that provides maps upon request (use case diagram of figure 4, page 6 identifies such a system as an actor). Package Mapping defines a general solution to the mapping concern by providing an interface to this external system. The following defines the VCL packages addressing the mapping generic concern.

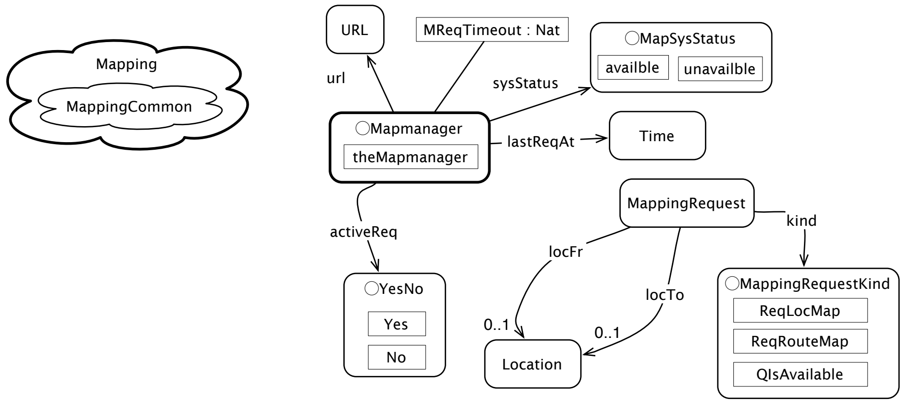


Fig. 32. Package (left) and structural (right) diagrams of package Mapping.

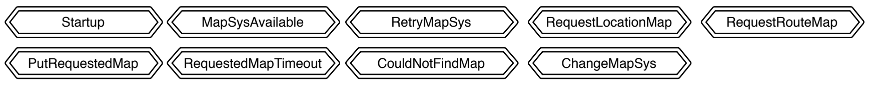


Fig. 33. Global Behavioural diagram package Mapping.

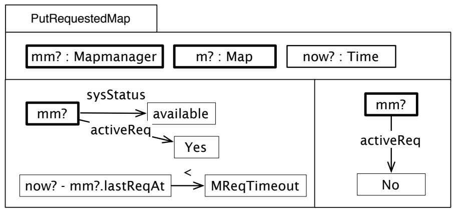
Package MappingCommon. It defines structures that are common across several mapping packages. Its package and structural diagrams are given in figure 31. SD introduces blobs MapType (defining the two types of maps, location or route) and Map (defining a map to be displayed).

Package Mapping. It encapsulates the general solution to the mapping concern. Its package and structural diagrams are given in figure 32. Blob MapManager is the package's controller; it contains a single object[5]. A MapManager holds a url of the mapping system, an availability flag of the mapping system (sysStatus), a time at which the last request to the mapping system was made (lastReqAt), a flag indicating whether there is an active mapping request or not (activeReq). In addition, blob MapManager includes a constant MReqTimeout, indicating the timeout period associated with a request to the mapping system.

Blob MappingRequest represents a request to the mapping system. A MappingRequest has a request kind of blob MappingRequestKind; this represents queries to ask whether the external mapping system is available (QIsAvailable), to request a location map (RequesLocMap), or request a route map (RequesRouteMap). MappingRequest also also has two locations (locFr and locTo).

Figure 33 presents Mapping's global BD showing the operations offered to the outside world. This includes operations to: start the mapping controller

[5] This follows *singleton* pattern of [16].



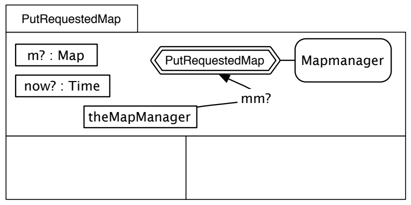


Fig. 34. Contract diagrams of operation PutRequestedMap of blob MapManager (left), and global operation with same name (right).

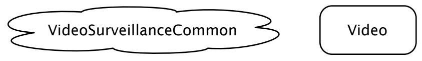


Fig. 35. Package (left) and structural (right) diagrams of package VideoSurveillanceCommon.

(Startup), check if external mapping system is available (MapSysAvailable), retry an availability query on the external mapping system (RetryMapSys), request a location map from the external mapping system (RequestLocationMap), request a route map from the external mapping system (RequestRouteMap), load requested map (PutRequestedMap), issue a timeout when requested map is not delivered on time (RequestedMapTimeout), indicate that requested map could not be found (CouldNotFindMap), and change the external mapping system (ChangeMapSys).

Figure 34 presents contract diagrams for the operations of Mapping that are involved in defining the global operation PutRequestedMap. It presents a contract diagram for local operation PutRequestedMap of blob MapManager (figure 34, left); this requires (pre-condition) that the external mapping system is available, that there is an active request and that the timeout period for the request has not expired; the post-condition says that there is no active request. Global contract PutRequestedMap imports its local counter-part and says that the local operation is to be executed on the sole MapManager instance.

5.9 Video Surveillance

The *video-surveillance* concern is related with the handling of videos requested from the external video surveillance system. In our model, we consider that there is an external video surveillance system that provides video footage upon request (use case diagram of figure 4, page 6 identifies such a system as an actor). Package VideoSurveillance defines a general solution to the *video-surveillance* concern by providing an interface to this external system. The following defines the packages addressing this generic crosscutting concern.

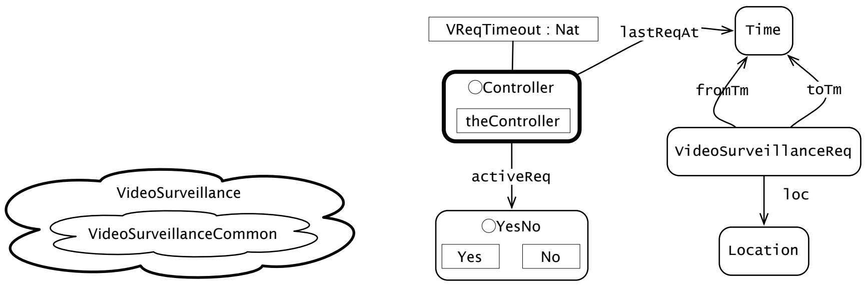


Fig. 36. Package (left) and structural (right) diagrams of package VideoSurveillance.

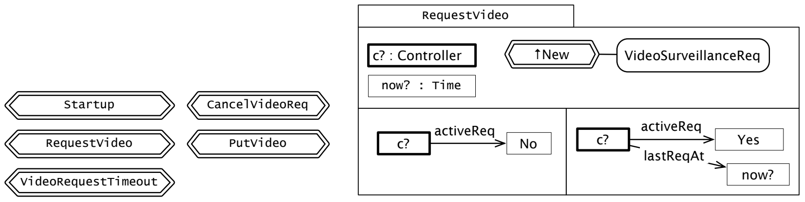


Fig. 37. Global behaviour diagram of package VideoSurveillance. Contract diagrams of operation ViewVideoSurveillance of blob Controller (left)

Package VideoSurveillanceCommon. It defines structures that are common across several packages that need video surveillance. Figure 35 presents its package (left) and structural (right) diagrams. SD introduces blob Video, which defines a video footage to be shown to the user.

Package VideoSurveillance. It defines the actual general solution to the video surveillance concern. Its package (left) and structural (right) diagrams are given in figure 36. Blob Controller is the package's controller; it represents a singleton set. Its property edges include: time of last request to the mapping system was made (lastReqAt), flag indicating whether there is an active video request (activeReq). In addition, Controller has constant VReqTimeout, indicating the timeout delay of a request to the external video-surveillance system.

Blob VideoSurveillanceReq represents a request to the external video surveillance system. It has properties: location (loc) and a time period (fromTm and toTm); these indicates location and time-period of requested video footage.

Figure 37 (left) presents VideoSurveillance's global BD showing the operations offered to the outside world. This includes operations to: start the controller (Startup), request video footage from external system (ReqVideoSurveillance), cancel a video surveillance request (CancelVideoReq), load requested video (PutVideo), and issue a timeout when a video request expires (VideoRequestTimeout).

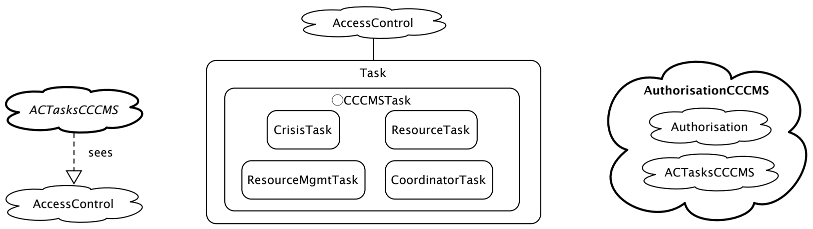


Fig. 38. Package ACTasksCCCMS sees AccessControl (left). Package ACTasksCCCMS specialises Task in the context of CCCMS (centre). Package AuthorisationCCCMS extends Authorisation and sees ACTasksCCCMS (right).

Figure 37 (right) presents the contract diagram of operation RequestVideo of blob Controller. This requires (pre-condition) that there is no active request; the post-condition sets the active request flag and the lastReqAt property to the current time (input now?). The global contract of RequestVideo would import the local contract with the same name and request it to be carried out by the sole Controller instance (similarly to Mapping's global contract of figure 34).

6 Customising generic packages for the CCCMS context

Generic packages are designed to be used in various settings. Usually, they need to be configured (or adjusted) for the new context in which they are used. The next sections illustrate such customisations; they show how generic packages from section 5 are customised to the CCCMS.

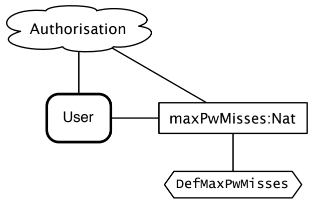
6.1 Customising Authorisation by defining a configuration

Customisation of package Authorisation is performed by defining packages ACTasksCCCMS and the configuration AuthorisationCCCMS (figure 38).

Package ACTasksCCCMS defines the set of tasks that are subject to access control in the CCCMS; it *sees* package AccessControl (figure 38, left). The *sees* arrow defines a dependency relationship; in the figure, it means that it uses structures defined in AccessControl, but those structures are not part of the state of the newly defined package. Package ACTasksCCCMS defines set CCCMSTask, which subsets set Task of package AccessControl (see [14] for further details) and defines all CCCMS tasks to be subject to access control.

Package AuthorisationCCCMS (figure 38, right) customises package Authorisation (authentication+access control, section 5.5) to the CCCMS; it extends packages Authorisation and ACTasksCCCMS. It defines a set of initial rôles and permissions, and customises constant maxPwMisses of blob User.

Figure 39 (left) presents SD of package AuthorisationCCCMS. This package does not add any new state; it merely inherits state from package Authorisation.



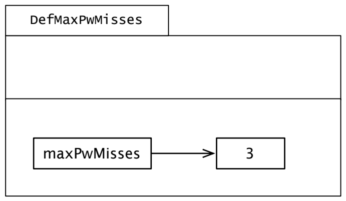


Fig. 39. SD of package AuthorisationCCCMS (left). Definition of constant maxPwMisses (right).

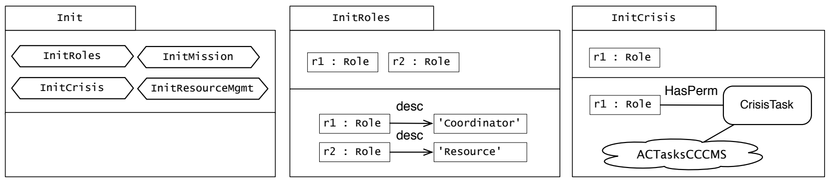


Fig. 40. Constraint diagrams specifying initialisation of the package AuthorisationCCCMS (Init), initialisation of roles (InitRoles) and an initialisation for the permissions for the tasks related with crisis management (InitCrisis).

SD says that constant maxPwMisses of blob User has a constraint defining it. This constraint defines maxPwMisses to hold value 3 (figure 39, right).

Figure 40 presents constraint diagrams defining the initial state of package AuthorisationCCCMS. In VCL, the keyword Init associated with a constraint indicates that an initialisation of some structure or package is being defined. Constraint diagrams of figure 40 are as follows:

– Init defines package's initial state. It imports other constraint diagrams defining initial state of set Role (InitRoles), and initial state of permissions with respect to tasks of crisis management (InitCrisis), mission management (InitMission) and resource management (InitResources).
– InitRoles defines rôles Coordinator and Resource.
– InitCrisis says that the role r1 (Coordinator) has permission to execute all tasks of set CrisisTask defined in package ACTasksCCCMS.
– Remaining constraint diagrams defining initial states are defined in [14].

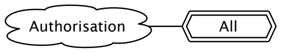


Fig. 41. BD of AuthorisationCCCMS package.

Package AuthorisationCCCMS needs to extend the behaviour of Authorisation. Its BD is given in figure 41. Its global operations are all operations of Authorisation, which are kept unaltered; this is defined by using *integral extension* on all global operations of package Authorisation. This is expressed by connecting a contract with keyword *All* to the

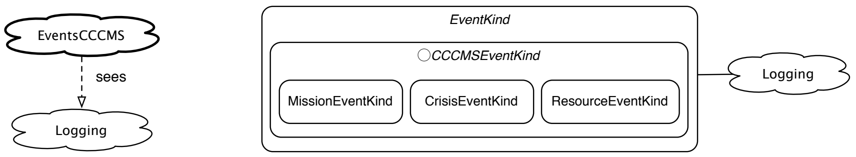


Fig. 42. Package EventsCCCMS extends Logging (left). EventsCCCMS's SD specialise blob EventKind defined in Logging (right).

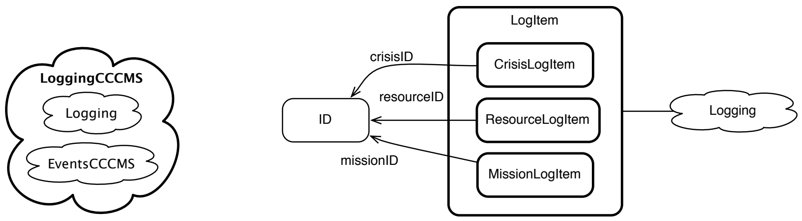


Fig. 43. Package LoggingCCCMS extends Logging and sees EventsCCCMS (left). LoggingCCCMS's SD introduces blobs CrisisLogItem, ResourceLogItem and MissionLogItem, which are defined as subsets of blob LogItem defined in package Logging (right).

package where operations come from. When extensions are defined in this way, no extra contract definitions are necessary.

6.2 Customising Logging by adjusting behaviour

To use the generic Logging package in CCCMS, its structure and behaviour need to be adjusted. This is done following the incremental or additive approach to change that characterises VCL: we define new structures and operations that extend those of the generic package.

Packages EventsCCCMS and LoggingCCCMS customise package Logging (section 5.7) to the CCCMS context. This involves the following adjustments:

- Package EventsCCCMS defines the events to be logged in CCCMS, by defining set CCCMSEventKind, a subset of EventKind blob defined in Logging (Figure 42).
- Blob LogItem of package Logging does not carry enough information for the needs of CCCMS. LoggingCCCMS extends this blob to carry the required extra information, providing operations for the new structure.

Package LoggingCCCMS (figure 43, left) extends package Logging and sees package EventsCCCMS. Its SD (figure 43, right) introduces three sub-blobs (subsets) of blob LogItem: CrisisLogItem, ResourceLogItem and MissionLogItem. These blobs represent log items of events related with crisis management, resource management and mission management, respectively. They define a property to hold an identifier of either a crisis, mission or resource.

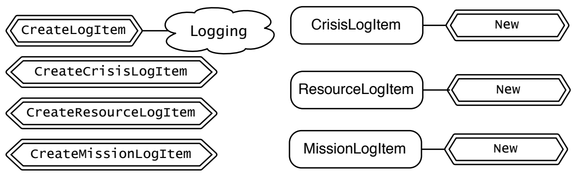


Fig. 44. Behavioural diagram of package LoggingCCCMS, comprising local operations New of blobs CrisisLogItem, MissionLogItem and ResourceLogItem, and global operations CreateLogItem (defined from Logging by integral extension), CreateCrisisLogItem, CreateResourceLogItem and CreateMissionLogItem.

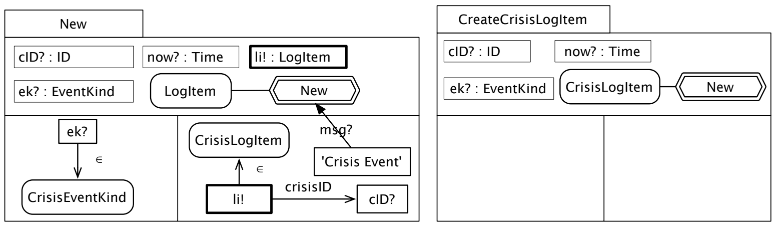


Fig. 45. Contract diagram defining operations New of blob CrisisLogItem and CreateCrisisLogItem.

Figure 44 presents BD of package LoggingCCCMS. This defines local operations New of blobs CrisisLogItem, MissionLogItem and ResourceLogItem, and global operations CreateLogItem (defined by integral extension), CreateCrisisLogItem, CreateResourceLogItem and CreateMissionLogItem. Global operation CreateLogItem enables logging of general log items; remaining global operations log items related with crisis, resource and mission management. Contracts for these operations are fully defined in [14].

Operation New of blob LogItem, defined in package Logging, creates general LogItem objects; in package LoggingCCCMS, we need new operations to take specialisations of LogItem into account; this is done using *behavioural extension*. This is illustrated in figure 45, which gives contract diagrams for operations New of blob CrisisLogItem and global operation CreateCrisisLogItem:

– Operations New of blob CrisisLogItem declares inputs cID?, ek? and now? and imports operation New of blob LogItem (defined in package Logging), which is the operation to specialise. Predicate compartments define the extra behaviour of this operation. The precondition says that input ek? must belong to set CrisisEventKind defined in package EventsCCCMS. The postcondition sets property crisisID of object li! to input cID?, requires that *li* belongs to set CrisisLogItem, and sends the message to be logged with value 'Crisis Event' to imported operation New over input msg?.

– Global operation CreateCrisisLogItem declares same inputs as imported CrisisLogItem.New. Pre- and post-conditions are as in imported contract.

6.3 Customising session management

VCL package SessionMgmtCCCMS customises package SessionMgmt (figure 31). It specifies the value of constant maxInactivityTm (maximum period of inactivity) to be 30 minutes and defines its operations by integral extension (as operations of LoggingCCMS in figure 44). See [14] for full details.

6.4 Customising mapping

VCL package MappingCCCMS represents customisation of package Mapping (figure 32) to the CCCMS. It specifies the value of constant MReqTimeout (timeout of a request to the mapping system) to be 30 seconds and defines its operations by integral extension (as operations of package LoggingCCMS in figure 44). See [14] for full details.

6.5 Customising video-surveillance

VCL package VideoSurveillanceCCCMS represents customisation of package VideoSurveillance (figure 35) to the CCCMS. It specifies value of constant VReqTimeout (timeout of a request to the video-surveillance system) to be 30 seconds and defines its operations by integral extension (as operations of package LoggingCCMS in figure 44). See [14] for full details.

7 Packages that localise Problem Domain Concerns

A key design principle that we followed in modelling this large-scale case study, is to decompose by *localising* key functionalities and then incrementally add details by defining extensions. We call this design principle: *keep key modules as local as possible*. This principle is followed in order to maintain *low-coupling* in our package design. This can be observed in our design for the general packages that describe a solution to a crosscutting concern, such as Authentication and AccessControl (section 5). We can see that from these more localised packages, we defined an extension that brings the two packages together to define Authorisation, and then we add further detail to adjust Authorisation to the problem domain context (section 6).

In the design of the packages that capture the problem domain, the same principle is followed. Core domain packages focus on domain concerns only in order to keep things as small and as local as possible. Domain packages are then composed with other packages to bring about the desired system.

Figure 46 overviews the core domain packages of the CentralCCCMS subsystem. This includes:

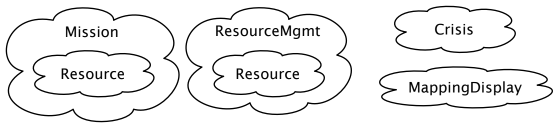


Fig. 46. Core packages that modularise problem domain concerns in CentralCCCMS subsystem.

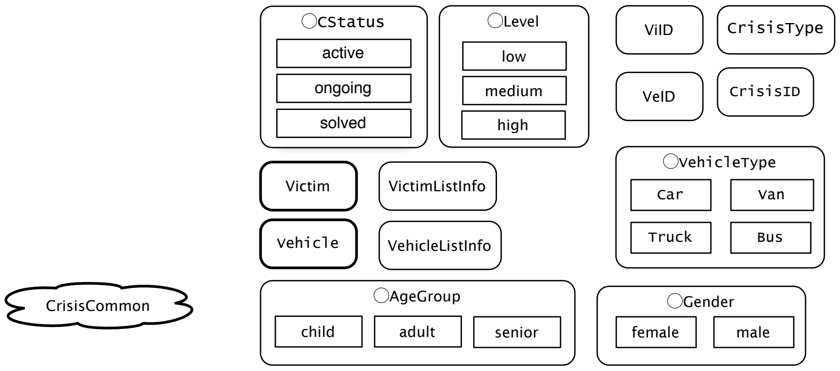


Fig. 47. Package diagram describing package CrisisCommon (left) and its global structural diagram (right).

- package Mission, which extends package *Resource* and localises the *mission management* concern;
- package ResourceManagement, which extends package Resource and localises the *resource management* concern;
- package Crisis, which localises the *crisis management* concern;
- and package MappingDisplay, which localises the concern of *map handling* that is common to both MobCCCMS and CentralCCCMS subsystems.

All these packages are fully defined in [14]. The next sections present the domain packages that address the crisis management concern, and illustrate VCL's capability to capture crosscutting domain concerns with package MappingDisplay.

7.1 Crisis Management

The following overviews the VCL model of the crisis management domain concern, presenting VCL packages CrisisCommon, WitnessReports and Crisis.

Package CrisisCommon. It introduces structures that are common across crisis-related functionality. Its package and global structural diagrams are given in figure 47. Its complete definition is given in [14].

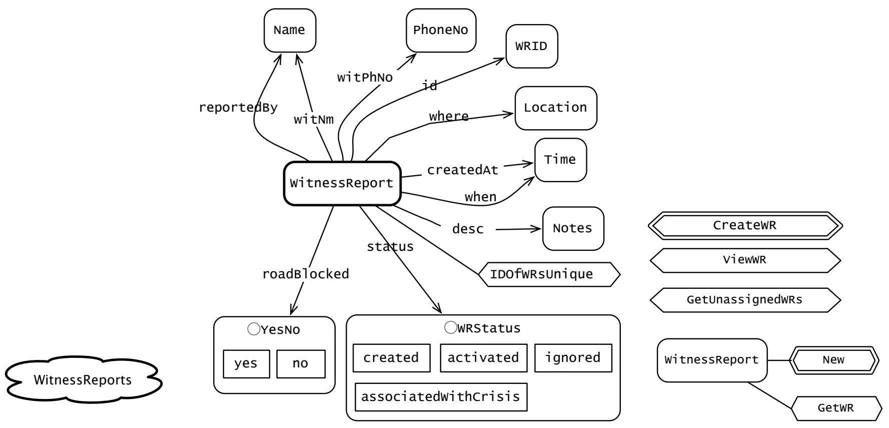


Fig. 48. Package diagram describing package WitnessReports (left), and its structural (centre) and behavioural diagrams (right).

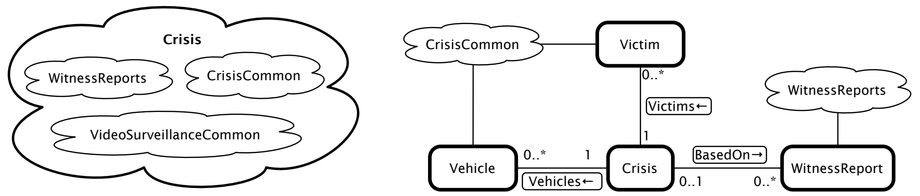


Fig. 49. Package (left) and structural (right) diagrams of package Crisis.

Package WitnessReports. It encapsulates functionality related with witness reports. Its package, structural and behavioural diagrams are given in figure 48. The SD introduces blob WitnessReport, which holds information of car crashes reported by witnesses. The BD introduces global operations CreateWR (to create a witness report), ViewWR (to view a witness report), GetUnassignedWrs (to obtain those witness reports that are active and have not been assigned to a crisis), together with local operations of blob WitnessReport. Full definition of this package is given in [14].

Package Crisis. It encapsulates the functionality of the crisis management feature. Its package and structural diagrams are given in figure 49. The package diagram says that Crisis extends packages CrisisCommon, WitnessReports and VideoSurveillanceCommon (section 5.9, page 25). The SD introduces blob Crisis, which represents the actual crisis instance opened in the system, and some relational edges. Relational edges Vehicles and Victims indicate, respectively, the vehicles and victims involved in a crisis. Relational edge BasedOn indicates the witness reports associated with a crisis instance.

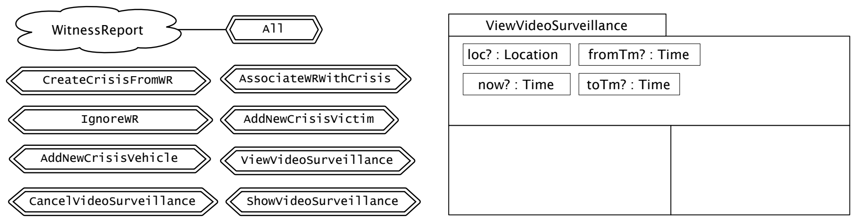


Fig. 50. Behavioural diagram of package Crisis and contract diagram defining operation ViewVideoSurveillance.

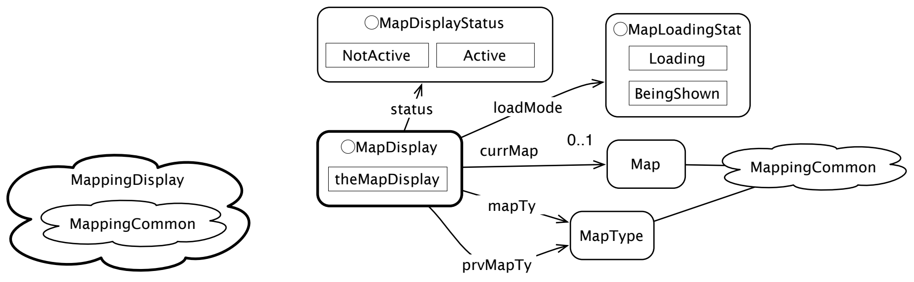


Fig. 51. Package diagram describing package MappingDisplay, which extends MappingCommon. Structural diagram of package MappingDisplay.

Figure 50 presents the global BD of package Crisis. This uses integral extension to say that all global operations of package WitnessReports are brought into Crisis, and introduces the following global operations:

- CreateCrisisFromWR: create a crisis from some witness report.
- AssociateWRWithCrisis: associates a witness report with an existing crisis.
- IgnoreWR: ignores a witness report because it is considered irrelevant.
- AddNewCrisisVictim: adds a victim record as part of an existing crisis.
- AddNewCrisisVehicle: adds a vehicle record as part of an existing crisis.
- ViewVideoSurveillance: asks for video surveillance footage of some location (contract diagram defining this operation is given in figure 50).
- CancelVideoSurveillanceReq: cancels a previous request for video surveillance.
- ShowVideoSurveillanceReq: shows the video surveillance request footage.

This package's complete definition is given in [14].

7.2 Package MappingDisplay

This package describes a generic user interface for handling maps that is common to both MobCCCMS and CentralCCCMS. It is package focused on the display of maps to the user that abstracts away from other features of mapping (such as the

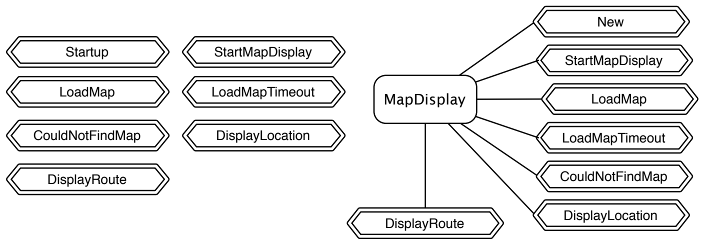


Fig. 52. Behavioural diagram of package MappingDisplay.

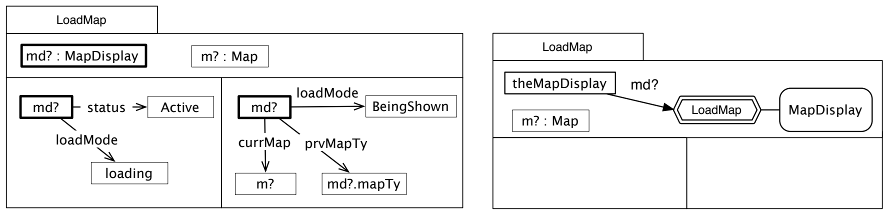


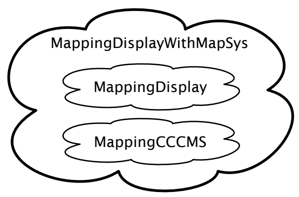
Fig. 53. Contract diagrams of local operation LoadMap of blob MapDisplay (left) and global operation with same name (right) in package MappingDisplay.

interface to the mapping system). Its package diagram is given in figure 51 (left); package MappingDisplay extends package MappingCommon (section 5.8). Package MappingDisplayWithMapSys (section 8) extends this package to incorporate the Mapping package defined in section 5.8.

Structure. Figure 51 (right) presents the SD of package MappingDisplay. Blob MapDisplay is a singleton. Blob MapDisplay has properties: map to display (currMap), current map type (mapTy), type of previous map (prvMapTy), load mode (loadMode), which indicates whether a map is currently being loaded (Loaded) or it has been loaded and is being shown (BeingShown), and current status of the display (status), which can either be Active or NotActive.

Behaviour. Figure 52 presents the BD of package MappingDisplay. This identifies this package's global operations together with local operations of blob MapDisplay. This includes operations to: initialise the map display controller (Startup), start displaying some map (StartMapDisplay), load a new map to display (LoadMap), take the appropriate action when there is timeout in loading some map (LoadMapTimeout), indicate that the requested map has not been found (CouldNotFindMap), display a location map for some location (DisplayLocation), and display a route map (DisplayRoute).

Figure 53 presents contract diagrams of local operation LoadMap of blob MapDisplay together with the global operation with same name. Local LoadMap



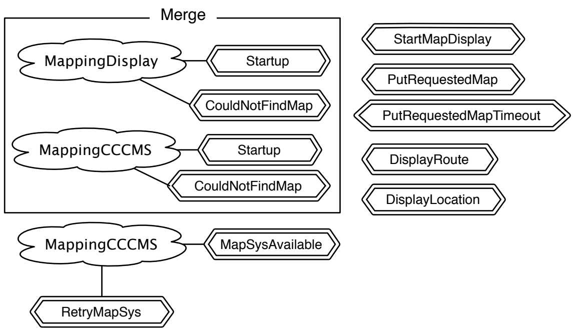


Fig. 54. Package (left) and behavioural (right) diagrams of package MappingDisplayWithMapSys.

requires that the map display is active and that the loading mode is Loading; the post-condition takes the map sent as input and sets it as the current map, changes the loading mode to BeingShown and sets the previous map type to that of the current map. Global LoadMap just dispatches the request to the sole MapDisplay instance. Remaining operations of this package are defined in [14].

8 Composing domain packages with aspect packages

The previous section fragmented the problem domain into coherent modules. These modules are confined to problem domain phenomena. They talk about crisis, missions, and mission resources, excluding access control, logging and other crosscutting concerns. This section shows how aspect packages are joined with problem-domain packages. This is illustrated with the crosscutting concerns: authentication, access control, session management, logging and mapping.

VCL package composition has already been illustrated. We have seen in sections 5 and 7 how it enables composition of any kind of module, be it a classical module or an aspect. This section continues this theme by showing how aspect packages can be joined into domain packages. In our AOM approach based on VCL, this can be done in two ways: (a) directly by specifying a package that represents the composition and incorporates the packages being composed, (b) or indirectly via a join interface. The following illustrates these two approaches.

8.1 Direct composition

The direct composition approach is illustrated with mapping and video-surveillance.

Mapping. Package MappingDisplayWithMapSys (figure 54) links domain package MappingDisplay (section 7.2) with generic Mapping package (section 5.8) to

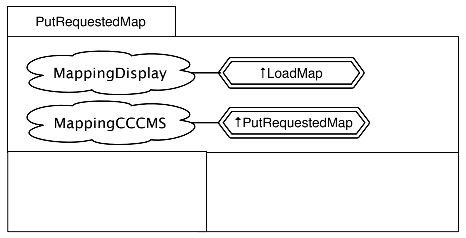


Fig. 55. Contract diagram of global operation PutRequestedMap of package MappingDisplayWithMapSys.

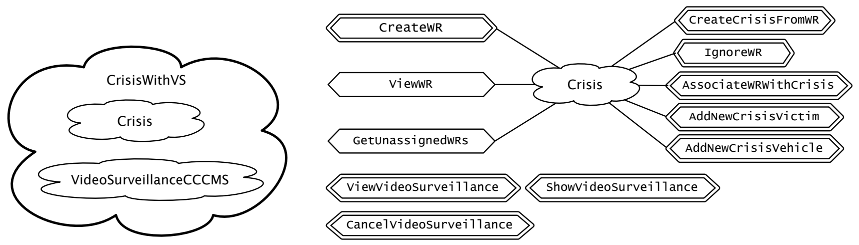


Fig. 56. Package (left) and behavioural (right) diagrams of package CrisisWithVS.

make a package that displays maps and interacts with an external mapping system. Figure 54 presents its package (left) and behavioural (right) diagrams. The package diagram says that this package incorporates packages MappingDisplay and Mapping. These two packages are linked at the behavioural level. BD says that the operations Startup and CouldNotFindMap of packages MappingDisplay and Mapping are to be *merged*; this is expressed using VCL's *merge join box*, which creates, through merging (conjunction) a new operation with the same name as those being merged. BD uses *integral extension* to say that operations MapSysAvailable and RetryMapSys are to be used integrally. The remaining operations of BD perform the behavioural composition using *custom extension*, being defined in their own contract diagrams. Figure 55 shows how this is done for operation PutRequestedMap; it puts together operations PutRequestedMap from Mapping (figure 34, page 25) and LoadMap from MappingDisplay (figure 53, page 35).

Video surveillance. Package CrisisWithVS links the domain package Crisis (section 7.1) with the generic VideoSurveillance package (section 5.9) to make a package that handles crisis and interacts with a video surveillance system. Its package (left) and behavioural (right) diagrams are given in figure 56. The package diagram says that it incorporates both packages Crisis and VideoSurveil-

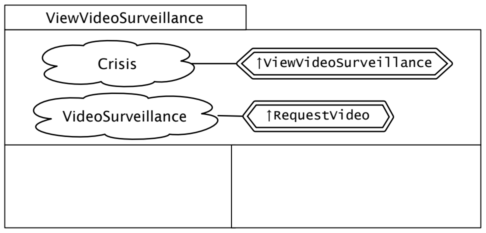


Fig. 57. Contract diagram of global operation ViewVideoSurveillance of package CrisisWithVS.

lance. These two packages are combined at the behavioural level. The BD uses integral extension to say that operations of package Crisis are used integrally in the new package (CreateWR, etc.). Remaining operations interact with the video surveillance system and are defined using *custom extension*.

Figure 57 presents contract diagram of operation ViewVideoSurveillance, which joins two operations using a custom extension. This puts together operations ViewVideoSurveillance from Crisis (figure 50, page 34) and RequestVideo from VideoSurveillance (figure 37, page 26).

8.2 Indirect composition via a join interface

In many cases, package compositions cannot be performed directly. Sometimes an interface between the packages being composed is needed. This provides extra information to link the two worlds so that composition is meaningful; at the same time, it ensures that the two worlds can be specified apart and unaware from each other; the link between the two is described in a *join interface* package. The method for building compositions via join interfaces is as follows:

1. Build join interfaces that describe link (or bridge) between different packages. This involves defining the extra information (usually operations) required to link the packages being composed.
2. Add the join interfaces to a base package (here a domain package) to make the base package fit for composition. This is usually done using a merge extension.
3. Compose aspect packages with the package representing the base enriched with the interfaces. This is usually done using join extension.

Above, we added video-surveillance to domain package Crisis. This resulted in package CrisisWithVS, which is now composed with more aspects. The following illustrates the method of indirect composition with package CrisisWithVS.

Defining join interfaces. Join interfaces packages for adding aspects to package CrisisWithVS are as follows (figure 58):

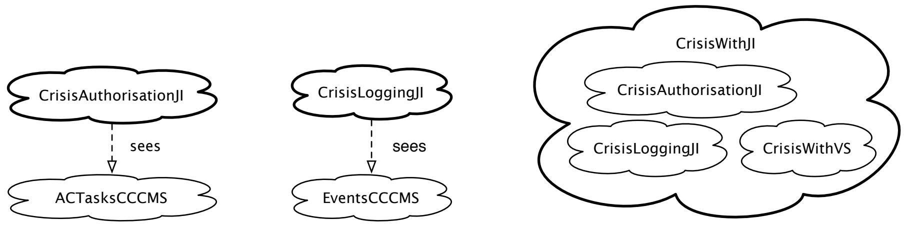


Fig. 58. Adding the required join interfaces to package Crisis. Package CrisisAuthorisationJI defines join interface for package Authorisation. CrisisLoggingJI defines joins interface for Logging. Package CrisisWithJI extends CrisisWithVS with required join interfaces.

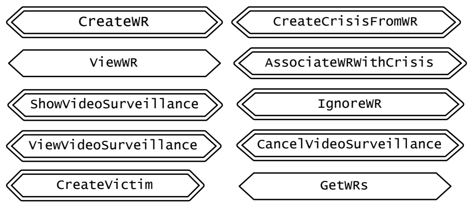


Fig. 59. Behavioural diagram is the same in packages CrisisAuthorisationJI and CrisisLoggingJI.

- Authorisation requires each global operation of CrisisWithVS to indicate the corresponding access-controlled task (an object of set Task defined in package ACTasksCCCMS); this is described in join-interface package CrisisAuthorisationJI.
- Logging requires each global operation of CrisisWithVS to indicate corresponding logging event (from set EventKind defined in EventsCCCMS); this is described in the join-interface package CrisisLoggiongJI.
- Join with SessionMgmt does not need an interface package.
- Package CrisisWithJI (Crisis with join interfaces) enriches the base package CrisisWithVS to make it fit for composition with aspect packages; it extends join interface packages and CrisisWithVS.

BDs of packages CrisisAuthorisationJI and CrisisLoggingJI (figure 59) comprise operations named after global operations of CrisisWithVS.

Figure 60 presents contract diagrams of operation CreateWR in CrisisAuthorisationJI (left) and CrisisLoggingJI (right). In CrisisAuthorisationJI, the contract diagram declares output t! and says in postcondition compartment that its value is TCreateWR, which is the access-controlled task corresponding to this operation. In CrisisLoggingJI, CreateWR declares output ek! and says in postcondition compartment that its value is EvCreateWR, which is the logging event corresponding to operation CreateWR.

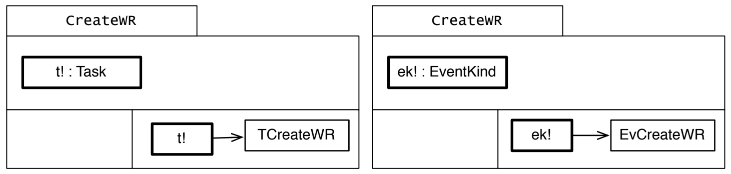


Fig. 60. Contract diagrams for join interface operations CreateWR in CrisisAuthorisationJI (left) and CrisisLoggingJI (right).

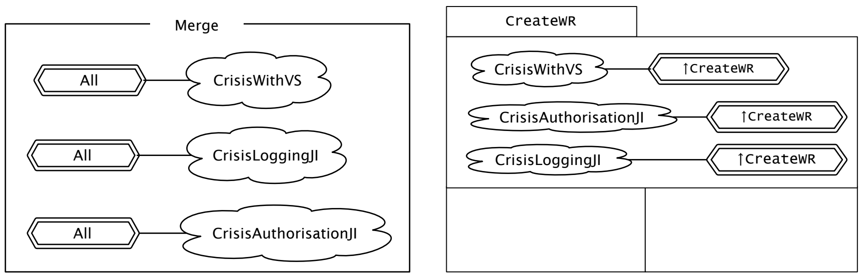
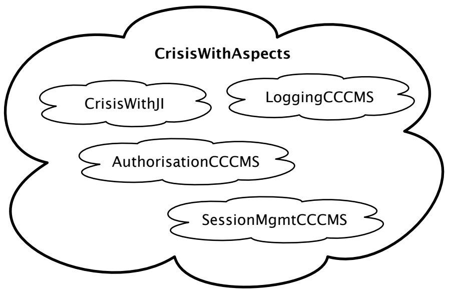


Fig. 61. Behavioural diagram of package CrisisWithJI, which uses *merge extension* (left). Operation CreateWR of package CrisisWithJI that results from *merge extension* (right).

Enriching a base package with join interfaces. Individual join interfaces need to be added to the base package that is going to be the target of the ultimate composition. Package CrisisWithJI represents this composition; it extends the join interface packages and base package CrisisWithVS(figure 58, right). Joining of behaviour is done using *merge extension*. BD of CrisisWithJI (figure 61, left) illustrates this; it declares operations with the same name as globals of CrisisWithVS and packages defining the join interfaces. Operations with same name are *merged*, which results in their composition through conjunction. Figure 61 (right) shows the effect of the merge for operation CreateWR.

Adding aspects to the base package. Once a base package is equipped with a join interface, it is possible to incorporate aspects. This is done following the extension or additive approach that characterises VCL.

Package CrisisWithAspects represents the result of adding aspects to the base package CrisisWithVS. It extends packages CrisisWithJI, LoggingCCCMS, AuthorisationCCCMS and SessionMgmtCCCMS (figure 62, left). Figure 62 (right) presents its BD, which defines the global behaviour of the package using *join extension*. BD says that the behaviour of package CrisisWithAspects is made of all global operations defined in the package CrisisWithJI, but extended with the join contracts AuthorisationOp, SessionMgmtOp and LoggingOp.



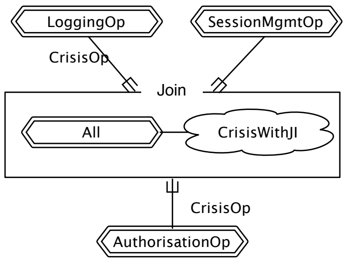


Fig. 62. Package CrisisWithAspects extends packages CrisisWithJI, Authorisation-CCMS, LoggingCCMS and SessionMgmtCCCMS (left). Behavioural Diagram of package CrisisWithAspects specifying the join extensions (right).

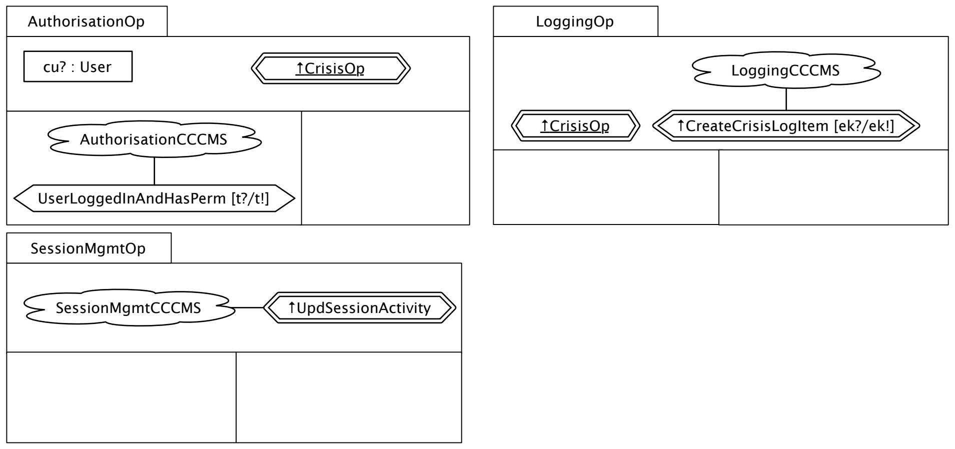


Fig. 63. Contract diagrams of join operations LoggingOp, AuthorisationOp and SessionMgmtOp.

Join operations of figure 62 (right) are described in figure 63. They are as follows:

– AuthorisationOp imports operation to join (CrisisOp) and declares input cu? to represent current user. Precondition says that current user must be logged in and has the required execute permissions for given task (constraint User-LoggedInAndHasPerm).
– LoggingOp imports operation to join (CrisisOp), and CreateCrisisLogItem, which represents extra behaviour to add.
– SessionMgmtOp imports UpdSessionActivity (see figure 21, page 18). This results in the updating of activity for session of the current user.

Package CrisisWithAspects is to be incorporated by the subsystem package CentralCCCMS (see figure 6) so that this system's model includes problem domain functionality of package Crisis augmented with all required crosscutting concerns. The process illustrated above is applied to the construction of other packages, namely: ResourceWithAspects and MissionWithAspects, also part of CentralCCCMS (see figure 6).

9 Discussion

This paper illustrates an AOM approach based on VCL with a large-scale case study. It shows how VCL's package construct enables the modularisation of concerns that can either be generic or problem-specific, and crosscutting or not. It highlights VCL's *plug-configure-and-play* based on packages, where the configuration step is in most cases trivial.

VCL is a language with an abstract and declarative nature designed with a formal semantics. It tends to emphasise the *what* rather than the *how*; it describes what is to be achieved and computed, abstracting away from the details of how it is achieved. VCL's contracts, for instance, describe what an operation is to perform, without going into specific details on how the operation is to be implemented. This style follows the abstract mathematical semantics of VCL, which is based on set theory, predicate calculus, and design by contract. A VCL model results in a Z specification, which can be subject to formal semantic analysis using proof (assisted by a theorem prover, such as Z/Eves) to enable verification and validation of VCL models.

The VCL-based AOM approach presented here is *symmetric*. Crosscutting and non-crosscuting modules all are represented equally as VCL packages that can be composed using VCL's compositions mechanisms. This paper provides several illustrations of this: section 5 composes aspect packages, section 7 composes packages addressing non-crosscutting concerns, and section 8 composes aspect packages with non-crosscutting problem domain packages.

VCL's package composition mechanisms provide various ways of specifying compositions. They are as follows:

- *State Extension.* This occurs when composite packages define their own state structures by combining state from the packages being extended; this tends to create more tightly coupled compositions. The Crisis package (figure 49, page 33) was built in this way.
- *Custom behavioural extension.* Most package compositions presented here involve behaviour only: different behaviours from different packages are combined to make new behaviours. Custom extension combines in a single operation the behaviours from the different packages being imported. This is illustrated in compositions of generic mapping with package MappingDisplay and video-surveillance with package Crisis in section 8.1.
- *Integral, merge and join extensions.* VCL provides other ways of composing behaviour; these specialise custom extension to facilitate behavioural composition. Integral extension makes available in a new context operations coming from some package being extended; the operation is used in the new context integrally (it is not changed in any way). Merge extension merges

(or conjoins) operations from different packages but with the same name in a new context. Join extension enables the addition of aspects to a group of operations. These extensions are illustrated in sections 5 and 8.

VCL takes a *plug-configure-and-play* approach to composition. Its compositions are additive and non-invasive (composed packages are not changed); they describe how packages are plugged in or out. The configuration of compositions is, in most cases, trivial; it involves integral, join and merge extensions, which have nice modular properties. More complicated configurations involve state and custom extensions. Section 8.2 showed how concerns of access control, authentication and logging could be added to a collection of packages representing problem domain concerns in a non-invasive way (that is, composed packages were not changed), where the laborious part consisted of defining the surrounding interface between the packages being composed. Once this was done, the composition itself was trivial; it involved *merge* and *join* extension. In other cases, the composition involves a custom extension, which, although more laborious, can also be specified in a modular and non-invasive way.

VCL is designed to capture concerns with functional solutions. This reflects functional nature of VCL packages, which are functional units encapsulating structure and behaviour. France et al [2] consider two broad categories of concerns: *concrete* and *quantitative*. Concrete concerns have functional solutions, whereas quantitative concerns are based on the quantification of qualities or attributes of a system. VCL-based AOM approach presented here is applicable to concrete concerns only. Therefore, all requirements of [12] that refer to quantitative qualities (such as *the system shall not exceed a maximum failure rate of* 0.001%) are not expressible in VCL.

VCL described most requirements of CCCMS case study [12]. We did not find difficulties in describing functional requirements. Requirements classified as non-functional in [12], such as those dealing with security, mobility, persistence, and accuracy, could also be described in VCL because they could be expressed functionally. Again, requirements corresponding to quantitative concerns could not be described because VCL has not been designed to express such properties.

The size of the problem was the biggest obstacle to model the CCCMS. This often hampered our ability to conceptualise the problem, to separate concerns, and to find the right abstractions. However, once this was achieved, we could proceed with VCL modelling. Despite the difficulties, VCL's mechanisms of separation of concerns helped to tackle the complexity of the problem. The technical report that accompanies this paper [14] is 234 pages long; the overall VCL model of CCCMS addresses eight croscutting system concerns and is made of 42 VCL packages.

10 Evaluation

This section evaluates VCL against some qualitative criteria using the VCL model of CCCMS presented here.

10.1 Scalability

VCL enables the decomposition of a problem into meaningful and manageable pieces. VCL packages represent modules that are very focussed in terms of what they do and small in terms of size. VCL's plug-configure-and-play approach to composition, where the configuration step is in most cases trivial, clearly states how a whole is composed out of its parts. It enables a non-invasive approach to composition that clearly separates the details of the composition from the inner details of the composed packages. This helps scalability by reducing complexity, aiding understanding and facilitating the construction of large models. A system can be understood by analysing its parts in isolation together with the compositions fitting different parts into ensembles. In the VCL model of CC-CMS, the generic packages have less than ten structures and global operations; larger packages describe the composition of smaller parts into a whole.

VCL's package composition mechanisms do not incur a big overhead of complexity. Semantically, all they do is to define a contract, which is then conjoined with the contract(s) being imported. In the case of custom and join extensions, the composite contract may add an extra precondition and postcondition. In the end, all that needs to be done when going from the visual world to the underlying Z world is to build these conjunctions.

10.2 Usability

There are many features in VCL that benefit usability:

– VCL is a visual language, which by itself tends to aid usability [17]. Visual representations, when well designed, tend to facilitate human processing [17].
– VCL's *zooming* features help to reduce clutter in diagrams, which benefits usability. Users zoom in to see details and zoom out to get an overview; this is possible in both definitions of structure and behaviour. In the SD of package Authentication (figure 13, page 14), the global view highlights the package's main structures and the relations that exist between them; through zooming it is possible to see the details of blobs Session and User (Fig. 13, right, page 14). The same applies to BDs: the global view highlights the package's global behaviour, through zooming it is possible to see the behaviour of local structures, and by clicking on the operation (in VCL's tool) it is possible to see its definition — for instance, see global BD of package AuthenticationOps (figure 15) and local BD of blob User (figure 16).
– VCL is designed with an underlying formal semantics, providing a layer that abstracts away from the underlying mathematics. This enables users to build large formal models using intuitive visual concepts, saving them from the details of mathematical formulas[6]. We found that it was more productive

[6] VCL's formal Z semantics is illustrated in [14] with the Z specification of package Authorisation and its sub-packages (sections 5.1 to 5.5). Complete VCL formalisation and generation of Z in VCL's tool, are ongoing efforts at the time of writing.

to specify in VCL than in Z directly, and that VCL was easier to learn than Z; this, of course, need further empirical validation[7].
– VCL packages and its plugging style of modular composition benefit usability by facilitating large-scale modelling. Packages can address a single concern or more than one concern; concerns can be removed or added using VCL's plug-configure-and-play approach.

10.3 Reuse

VCL packages enable large-scale reuse at the level of abstract models. In the VCL model presented here, packages Authentication and AccessControl represent generic solutions to common concerns; they can be reused in multiple contexts requiring a similar solution. We have also seen how we could define packages that were reused in both VCL models of CentralCCCMS and MobCCCMS (package MappingDisplay of section 7.2, page 34).

In the VCL-based approach presented here, adaptation and customisation of generic packages to a context is done through a configuration in an additive and non-invasive fashion, which enhances reuse. This can be observed in the customisation of generic packages to the CCCMS context done in section 6.

10.4 Correctness and testability

VCL is designed with a formal semantics. It takes a translational approach to semantics, where Z specifications are generated from VCL models[8]. These Z specifications are key for correctness and testability in VCL because they enable formal validation (or testing) and verification of certain desired properties. In Z, this is done using theorem proving.

A visual approach to formal validation of UML-based models that have a ZOO semantics (also used for VCL), called *snapshot analysis*, is developed in [18,9]. This tests a Z specification against examples and counter-examples (represented as snapshots) using theorem proving. We intend to incorporate this approach in VCL in the future. Snapshot-analysis of [18,9] is as follows:

– The modelled system's state space can be tested using single snapshots (UML object diagrams), which can either constitute a valid model instance or not (examples and counter-examples). Theorem proving ([18,9] uses the Z-Eves prover) checks whether the snapshot is valid or not. Snapshots effectively constitute tests, and they can reveal subtle errors in models.

[7] VCL model presented here was developed by a team made of one Z expert, two Alloy experts and one programmer. According to our experience, VCL appeared to be easier to learn and use than Z, and it appeared that it was more productive (and abstract) to specify in VCL than in Z directly.

[8] Generation of Z from VCL is an ongoing effort at time of writing; [14] illustrates the generation of Z from VCL for some packages of the CCCMS's VCL model.

– Operations of a model can be validated using snapshot-pairs. Snapshot-pairs have a before state and an after state; they can represent either a valid or invalid state transition. Theorem proving checks the validity of snapshot pairs.

The ZOO-based *snapshot analysis* of [18,9] has not been applied to the VCL model presented here. It could have been applied in the following ways:

– We could check the security properties of access control. For instance, to check that users are not authorised to execute tasks for which they do not have the required permissions. This would involve building a snapshot-pair that would achieve this (a non-authorised user executing some task) whose invalidity would be checked using theorem proving. This could be done in isolation within the scope of package AccessControl or in the broader model of the system where AccessControl has been weaved in.
– We could check properties of authentication, session management and other system concerns in a similar way.

10.5 Evolution

The VCL-based AOM approach illustrated here emphasises localisation of concerns and incremental construction of packages. This facilitates evolution because changes can be done either locally, as they are confined to individual packages, or incrementally, by adding new packages to encapsulate new requirements; often without requiring big changes to the overall model.

The model of the CCCMS presented here evolved in this way. Initially, we developed a simple model that focussed on the problem domain in order to gain a better understanding of the problem. Incrementally, we kept adding more features following the process described in section 2, by building new packages that would be composed to make a larger whole. It was often the case that we needed to go back and elaborate parts of the model; however, most of these changes were local with little impact upon the overall model.

10.6 Variability

VCL's package construct and its composition mechanisms enable the design of hierarchies of packages describing a family of model solutions. This done by defining packages capturing the commonality of some domain, which are then specialised for several variants. For instance, in the model presented here, package Users captures the commonality that exists between Authentication and AccessControl. We could have built packages representing more than one model of RBAC in a similar way; these various models could be defined by extending a base RBAC model.

10.7 Aspect Interaction

VCL is a declarative and abstract language. Its models are set in an abstract world where computations of individual components are performed in parallel. This mitigates the ordering problem that is typical of aspect interaction [19]. In the operations of our case study, there is a before-state (pre-condition) and an after state (post-condition); individual changes to the system's components are considered to be done in parallel; there is, therefore, no sequential ordering of computations. This precludes the ordering problems associated with aspect interaction. In our VCL models, all that is required to add aspects is to enforce the pre-condition, or the post-condition, or both; the execution order of computations representing pre- and post-condition predicates does not matter at this level of abstraction.

For example, the custom extension of figure 55 composes the generic mapping with MappingDisplay; the two operations being composed LoadMap and PutRequestedMap are considered to be performed in parallel, but synchronised on the communication channels that they share. In an implementation, PutRequestedMap would be executed before LoadMap, but at this level of abstraction this does not matter — that is a decision to be taken by a refinement of this model. This simplifies most compositions to a simple conjunction, without the need to specify an ordering of computations.

11 Related Work

VCL is a visual language for describing precisely structures, their constraints and behaviour at the abstract level of requirements (or high-level designs). It has a semantics based on set theory, predicate calculus and design by contract. VCL can be used on its own or in conjunction with other languages. This paper uses architectural block diagrams, and UML use case and sequence diagrams to describe the high-level requirements of the CCCMS; VCL is used to define a detailed yet abstract requirements model.

11.1 Contracts notation

VCL uses diagrams of contracts to describe operations. VCL contracts use two compartments placed side-by-side (left and right) to describe pre- (left) and post- (right) conditions. This is inspired by Catalysis' snapshot-pairs [20], where each snapshot represents a specific system state. VCL contract diagrams, however, denote an operation specification (a relation between before and after states).

Some approaches augment UML with contracts described as pairs of UML object-diagram pairs. Lohmann et al. [21,22,23] translate, using graph transformations, UML class diagrams and contracts to Java skeletons and JML assertions. Visual OCL [24] also uses graph transformations to go from contracts to OCL. Like VCL, these approaches follow a translational approach to semantics.

Constraint diagrams [25,26] notation has many similarities with VCL; it describes behaviour based on pre- and post-conditions and uses circles to represent sets and *insideness* to represent the subset relation. Unlike VCL, this approach does not take a translational approach to semantics; instead, the language is given a semantics to enable modelling and reasoning at the visual level. Constraint diagrams, however, is a formally defined notation; VCL presented here is a design of a language with an outline of a formal semantics. VCL's design illustrated here, however, provides better modularity mechanisms than all these approaches to contracts: contracts are modules that can be composed, and packages are coarse-grained modular units encapsulating structure and behaviour.

11.2 AOM

Several aspect-oriented modelling (AOM) approaches enhance UML to support modularisation of crosscutting concerns [2,27,28,29]. France et al. [2,27] propose an approach to architectural modelling based on class models and textual OCL operations, where template-based aspect models are instantiated for a particular context and then composed with a base model; the result of the composition is a merge based on signatures (a conjunction). VCL differs in that it targets requirements modelling, it is entirely visual, and does not use templates. The composition mechanisms of [2] are akin to VCL's merge extension; VCL provides other mechanisms of composition, such as custom and join extensions.

Other AOM approaches represent crosscutting behaviour as scenarios [28,29]. Whittle and Araújo's asymmetric approach [28] builds scenarios of crosscutting behaviour as interaction templates; these are composed with base scenarios (described as UML sequence diagrams) through an integration operator to synthesise state diagrams. The asymmetric approach of Kienzle et al. [29] defines aspect models made of class, state and sequence diagrams; it relies on *pointcuts* to describe points of insertion of crosscutting behaviour; behavioural models are composed through a weaving algorithm. VCL differs from these approaches in that it is symmetric, it uses contracts to totally describe behaviour (as opposed to partial scenario descriptions), and does not require complex weaving algorithms — compositions are done at the level of semantics, hidden from the visual world, and they involve simple conjunction with some merging of names. Unlike [28], VCL does not need ordering constraints to specify compositions involving crosscutting behaviour; at VCL's level of abstraction computations occur in parallel. Unlike [29], VCL does not need pointcuts; package operations are specified separately from compositions, which are described externally in a non-invasive way.

The protocol modelling AOM approach [30,31] has many things in common with VCL. It is, like VCL, symmetric and declarative, and designed for abstract modelling with an underlying formal semantics. Both take a similar declarative approach to composition not requiring weaving algorithms: the semantics of the composition operators rely on the properties of a composition operator of an underlying formal language; protocol modelling relies on the CSP parallel composition operator; VCL presented here relies on the schema conjunction operator of Z. The protocol machines approach defines compositions of individual protocol state machines, which are descriptions of behaviour. VCL offers similar compositions for contracts, VCL's behavioural artifact. In addition, VCL provides more flexible operators for coarser-grained compositions at the package level. The composite package may define some extra behaviour of its own. VCL provides join and merge extension mechanisms to enable flexible behavioural compositions at the package level, avoiding the need for individual behavioural compositions.

12 Conclusions

This paper illustrates a AOM approach based on VCL with the large car-crash crisis management system case study [12]. It illustrates VCL with fragments of the overall model to highlight VCL's AOM features. The complete VCL model of the case study is given in [14].

This paper illustrates several VCL features that are novel: (a) a modular approach to visual description of behaviour based on contracts, and (b) the coarse-grained modularity mechanism of packages, enabling the construction and flexible composition of classical modules and packages. The most relevant contribution of this paper is to show that VCL's novel modularity mechanisms are able to cope with the demands of large-scale modelling. In particular, the paper shows (a) how VCL's contract notation is capable of capturing complex behaviour, (b) how VCL's package construct enables modularisation of concerns, which can either be generic or problem-specific, crosscutting or not crosscutting; (c) how VCL's composition mechanisms provide a general and symmetric approach to the composition of concerns; and (d) how VCL packages constitute a coarse-grained unit of reuse enabling coarse-grained modular composition.

References

1. Parnas, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058 (1972)
2. France, R., Ray, I., Ghosh, S. Aspect-oriented approach to early design modelling. *IEE Proc. Softw.*, 151(4):173–185 (2004)
3. Amálio, N., Kelsen, P., Ma, Q. Specifying structural properties and their constraints formally, visually and modularly using VCL. In *EMMSAD 2010*, vol. 50 of *LNBIP*. Springer (2010)
4. Amálio, N., Kelsen, P. VCL, a visual language for modelling software systems formally. In *Diagrams 2010*, LNCS. Springer (2010)
5. Amálio, N., Kelsen, P. The visual contract language: abstract modelling of software systems visually, formally and modularly. Technical Report TR-LASSY-10-03, LASSY, Univ. of Luxembourg (2010). Available at `http://vcl.gforge.uni.lu/doc/vcl-tech-rep.pdf`
6. Meyer, B. Applying "design by contract". *Computer*, 25(10):40–51 (1992)
7. Spivey, J. M. *The Z notation: A reference manual.* Prentice-Hall (1992)
8. Amálio, N., Polack, F., Stepney, S. An object-oriented structuring for Z based on views. In *ZB 2005*, vol. 3455 of *LNCS*, pp. 262–278. Springer (2005)

9. Amálio, N. *Generative frameworks for rigorous model-driven development.* Ph.D. thesis, Dept. Computer Science, Univ. of York (2007)
10. Amálio, N., Polack, F., Stepney, S. UML+Z: Augmenting UML with Z. In Abrias, H., Frappier, M., eds., *Software Specification Methods.* ISTE (2006)
11. Amálio, N., Polack, F., Stepney, S. Frameworks based on templates for rigorous model-driven development. *ENTCS*, 191:3–23 (2007)
12. Kienzle, J., Guelfi, N., Mustafiz, S. Crisis management systems: a case study for aspect-oriented modeling. *Transactions on Aspect-Oriented Software Development*, 7:1–22 (2010)
13. Larman, C. *Applying UML and patterns.* Prentice-Hall (2002)
14. Amálio, N., Ma, Q., Glodt, C., Kelsen, P. VCL specification of the car-crash crisis management system. Technical Report TR-LASSY-09-03, LASSY, Univ. of Luxembourg (2009). Available at `http://vcl.gforge.uni.lu/doc/vcl-cccms.pdf`
15. Sandhu, R. S., Coyne, E. J., Feinstein, H. L., Youman, C. E. Role-based access control models. *Computer*, 29(2) (1996)
16. Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns: Elements of reusable object-oriented software.* Addison-Wesley (1994)
17. Larkin, J. H., Simon, H. A. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Sciece*, 11:65–99 (1987)
18. Amálio, N., Stepney, S., Polack, F. Formal proof from UML models. In *Proc. ICFEM 2004*, vol. 3308 of *LNCS*, pp. 418–433. Springer (2004)
19. Wilke Havings, I. N. L. B., Aksit, M. A graph-based approach to modeling and detecting composition conflicts related to introductions. In *AOSD'07*, pp. 85–95. ACM Press (2007)
20. D'Souza, D., Wills, A. C. *Objects, Components and Frameworks with UML: the Catalysis approach.* Addison-Wesley (1998)
21. Lohmann, M., Sauer, S., Engels, G. Executable visual contracts. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 63–70 (2005)
22. Engels, G., Lohmann, M., Sauer, S., Heckel, R. Model-driven monitoring: an application of graph transformation for design by contract. In *ICGT 2006* (2006)
23. Heckel, R., Lohmann, M. Model-driven development of reactive information systems: from graph transformation rules to JML contracts. *International Journal on Software Tools for Technology Transfer*, 9(2):193–207 (2007)
24. Ehrig, K., Winkelmann, J. Model transformation from visual OCL to OCL using graph transformation. *ENTCS*, 152:23–37 (2006)
25. Fish, A., Flowe, J., Howse, J. The semantics of augmented constraint diagrams. *Journal of Visual Languages and Computing*, 16:541–573 (2005)
26. Howse, J., Schuman, S., Stapleton, G. Diagrammatic formal specification of a configuration control platform. *ENTCS*, 259:87–104 (2009)
27. Reddy, R., Ghosh, S., France, R., Straw, G., Bieman, J. M., et al. Directives for composing aspect-oriented design class models. *TAOSD LNCS*, 3880:75–105 (2006)
28. Whittle, J., Araújo, J. Scenario modelling with aspects. *IEE Proc. Softw.*, 151(4):157–171 (2004)
29. Kienzle, J., Abed, W. A., Klein, J. Aspect-oriented multi-view modelling. In *AOSD'09*. IEEE (2009)
30. McNeile, A., Simons, N. Protocol modelling: a modelling approach that supports reusable behavioural abstractions. *Software and Systems Modelling*, 5(1):91–107 (2006)
31. McNeile, A., Roubtsova, E. CSP parallel composition of aspect models. In *AOM'08*. ACM Press (2008)