European Research Network on Foundations, Software Infrastructures and Applications for large scale distributed, GRID and Peer-to-Peer Technologies
A Network of Excellence funded by the European Commission

Developing Secure Chemical Programs with Aspects

Alvaro Arenas
A.E.Arenas@rl.ac.uk
STFC Rutherford Appleton Laboratory
Harwell Science and Innovation Campus, Didcot, OX11 0QX, UK

Jean-Pierre Banâtre, Thierry Priol
{Jean-Pierre.Banatre, Thierry.Priol}@inria.fr
INRIA/IRISA
Campus de Beaulieu, 35042 Rennes cedex, France

CoreGRID Technical Report
Number TR-0166
August 31st, 2008

Institute on Knowledge and Data Management

CoreGRID - Network of Excellence
URL: http://www.coregrid.net

CoreGRID is a Network of Excellence funded by the European Commission under the Sixth Framework Programme

Project no. FP6-004265

Developing Secure Chemical Programs with Aspects

Alvaro Arenas
A.E.Arenas@rl.ac.uk
STFC Rutherford Appleton Laboratory
Harwell Science and Innovation Campus, Didcot, OX11 0QX, UK

Jean-Pierre Banâtre, Thierry Priol
{Jean-Pierre.Banatre, Thierry.Priol}@inria.fr
INRIA/IRISA
Campus de Beaulieu, 35042 Rennes cedex, France

CoreGRID TR-0166
August 31st, 2008

Abstract

This paper studies security engineering of distributed systems when following the chemical-programming paradigm, represented here by the High-Order Chemical Language (HOCL). We have analysed how to model secure systems using HOCL. Emphasis is on modularity, hence we advocate for the use of aspect-oriented techniques, where security is seen as a cross-cutting concern impacting the whole system.

We show how HOCL can be used to model Virtual Organisations (VOs), exemplified by a VO system for the generation of digital products. We also develop security patterns for HOCL, including patterns for security properties such as authorisation, integrity and secure logs. The patterns are applied to HOCL programs following an aspect-oriented approach, where aspects are modelled as transformation functions that add to a program a cross-cutting concern.

1 Introduction

This paper studies security engineering of distributed systems when following the chemical-programming paradigm, represented here by the High-Order Chemical Language (HOCL) [2]. We have analysed how to model secure systems using HOCL. Emphasis is on modularity, hence we advocate for the use of aspect-oriented techniques, where security is seen as a cross-cutting concern impacting the whole system.

Chemical programming gets its inspiration from the chemical metaphor. In HOCL, computation is seen as reactions between molecules in a chemical solution. As a high-order programming language, reactions rules are molecules that can be manipulated like any other molecules, i.e. HOCL programs can manipulate other HOCL programs. Reactions occur locally between few molecules that are chosen non-deterministically.

The structure of the paper is the following. Section 2 introduces HOCL. Then, we describe in Section 3 the use of HOCL in the specification of virtual organisations, exemplified by a system for the generation of digital products. Section 4 presents security patterns for chemical programs. Next, section 5 shows how to apply the security patterns by using aspect-oriented programming. Section 6 relates our work with others. Finally, section 7 concludes the paper and highlights future work.

This research work is carried out under the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IST-2002-004265).

2 The High-Order Chemical Language

A chemical program can be seen as a (symbolic) chemical solution where data is represented by floating molecules and computation by chemical reactions between them. When some molecules match and fulfill a reaction condition, they are replaced by the body of the reaction. That process goes on until an inert solution is reached: the solution is said to be inert when no reaction can occur anymore. Formally, a chemical solution is represented by a multiset and reaction rules specify multiset rewritings. In this paper we use the High-Order Chemical Language HOCL [2], an extension of the γ-calculus [1]. Here, we introduce the main features of HOCL, referring the reader to [2] for a more complete presentation.

In HOCL, every entity is a molecule, including reaction rules. A program is a molecule, that is to say, a multiset of atoms $(A_1, \ldots, A_n)$ which can be constants (integers, booleans, etc.), sub-solutions ($\langle M \rangle$) or reaction rules. Compound molecules (M_1, M_2) are built using the associative and commutative operator ",", which formalises the Brownian motion and can always be used to reorganise molecules.

The execution of a chemical program consists in triggering reactions until the solution becomes inert.

A reaction involves a reaction rule **replace-one** P **by** M **if** C and a molecule N that satisfies the pattern P and the reaction condition C. The reaction consumes the rule and the molecule N, and produces M. Formally:

$$(\textbf{replace-one } P \textbf{ by } M \textbf{ if } C),\ N \longrightarrow \phi M \quad \text{if } P\ match\ N = \phi \text{ and } \phi C$$

where ϕ is the substitution obtained by matching N with P. It maps every variable defined in P to a sub-molecule from N. For example, the rule in

$$\langle 0,\ 10,\ 8,\ \textbf{replace-one } x \textbf{ by } 9 \textbf{ if } x > 9 \rangle$$

can react with 10. They are replaced by 9. The solution becomes the inert solution $\langle 0,\ 9,\ 8 \rangle$.

A molecule inside a solution cannot react with a molecule outside the solution (i.e. the construct $\langle . \rangle$ can be seen as a membrane). A HOCL program is a solution which can contain reaction rules that manipulate other molecules (reaction rules, sub-solutions, etc.) of the solution.

In the remaining of the paper, we use some syntactic sugar such as declarations **let** $x = M_1$ **in** M_2 which is equivalent to M_2 where all the free occurrences of x are replaced by M_1. The reaction rules **replace-one** P **by** M **if** C are one-shot: they are consumed when they react. Their variant denoted by **replace** P **by** M **if** C are n-shot, i.e. they do not disappear when they react.

There are usually many possible reactions making the execution of chemical programs highly parallel and non-deterministic. Since reactions involve only a few molecules and react independently of the context, many distinct reactions can occur at the same time.

An important feature of HOCL is the notion of *multiplets* [2]. A multiplet is a finite multiset of identical elements. In this paper, we limit ourselves to multiplets of basic values (integers, booleans, strings). In HOCL multiplets are defined and matched using an exponential notation: if v is a basic value then v^k ($k > 0$) denotes a multiplet of k elements v. Likewise, for variable x having a basic type, notation x^k denotes a multiplet of k elements. We could also have variables in the exponentation of constants or patterns, indicating that the size of a multiplet becomes dynamic.

3 Virtual Organisations in HOCL

A Virtual Organisation (VO) can be seen as a temporary or permanent coalition of geographically dispersed organisations that pool resources, capabilities and information in order to achieve common goals. VOs are given attention by researchers within a wide range of fields, from social anthropology and organisational theory to computer science. Their importance resides in providing an abstraction to represent organisational collaborations, a topic of fresh interest given the current exploitation of Internet technology to create virtual enterprises [3], or the sharing of resources across different organisations as envisaged by Grid computing [6].

We model here a VO with the goal of generating products resulting of the collaboration of several dispersed organisations, which posseses the following characteristics:

1. The VO aims at producing some complex, sophisticated 'digital' product (e.g. a software system, or some multimedia product).

2. The VO consists of a defined number of members (organisations), each one contributing to the generation of products.

3. The product generation is considered a *knowledge-intensive* and *content-intensive* activity. VO members depend on and need access to several sources of knowledge as well as digital content assets, which they assemble/use to create the product.

4. The production process is structured along some workflow (e.g. a software production process, or a Web/content publishing process), and foresees several phases. Policies may be applied to control access to the assets, which may vary according to the phase or state in the project workflow.

For our scenario, we are assuming a very simple workflow depicted in Figure 1. The workflow consists of four phases. In the *Edit* phase, work is distributed among all VO members contributing to the generation of a product. In the *Merge* phase, parts of the product created by each VO member are combined in order to create a global product. Once the global product is created, it is passed to the VO members in the *Validate* phase, so they can "validate" the product. Finally, the process finalises if the product is approved by a determined number of members by sending the product to *Publish*.

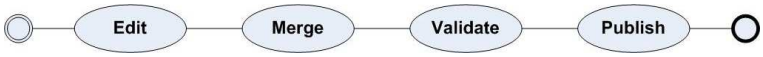

Figure 1: Workflow process for the VO supporting the generation of a product.

Chemical programming uses chemistry as a methaphor. For the case of our VO for product generation, the whole VO is modelled as a solution, which contains sub-solutions S_i:$\langle \cdots \rangle$ that represent the VO members. The product under construction is modelled as a molecule that could be tagged by another molecule representing the product status (`EDITING`, `EDITED`, `GENERATE`, `VALIDATING`, `VALIDATED`, `ACCEPTING` and `PUBLISHED`). Workflow operations ($edit$, $merge$, $publish$, etc.) are represented as reactions. Table 1 summarises the chemical modelling of the main elements of our VO.

VO Concept	**Chemical Representation**
VO	Solution
VO Member	Sub-solution
Workflow Operation	Reaction
Product	Molecule
Product Status	Molecule

Table 1: Chemical representation of the main elements of a virtual organisation for the collaborative generation of products

Figure 2 shows the HOCL program for generating a product. It consists of a solution containing all VO members —represented as subsolutions S_i for $i = 1, \cdots, k$, and molecule $GlobalProduct$, the product to be published. The reaction rule $edit$ distributes the global product to all VO members. Here we are assuming the existence of k VO members, where k is a predefined integer constant. Reaction $merge$ generates a local product, and marks the contribution of the corresponding member to the product generation by adding constant `GENERATE` to the global solution. It also includes operation $Merge$, which combines both the local and global products. The edition of a product finalises when VO members have contributed, which is represented by having $NumMerges(k)$ copies of molecule `GENERATE`. Function $NumMerges(k)$ is a domain-specific function indicating the number of copies needed to generate a product; if it is the identity function, i.e. equal to k, all participant solutions must contribute to the product generation. Note that we are exploiting here the existence of multiplets in HOCL: molecule $\texttt{GENERATE}^{NumMerges(k)}$ acts as a synchronisation barrier indicating when reaction $valid$ can occur. Reaction $valid$ distributes the final $GlobalProduct$ among the members in order to get their approval. Reaction $accept$ allows a VO member to vote for the approval of the product, which results in adding molecule `ACCEPTING` in the global solution. The whole process finalises as soon as $MinApproval(k)$ VO members approve the final product by executing reaction $publish$, which sends the final product to publishing. Funtion $MinApproval(k)$ is an abstraction of the protocol used to decide when to publish a product; for instance, if it is equal to $ceil(x/2)$, we would be using a majority vote protocol.

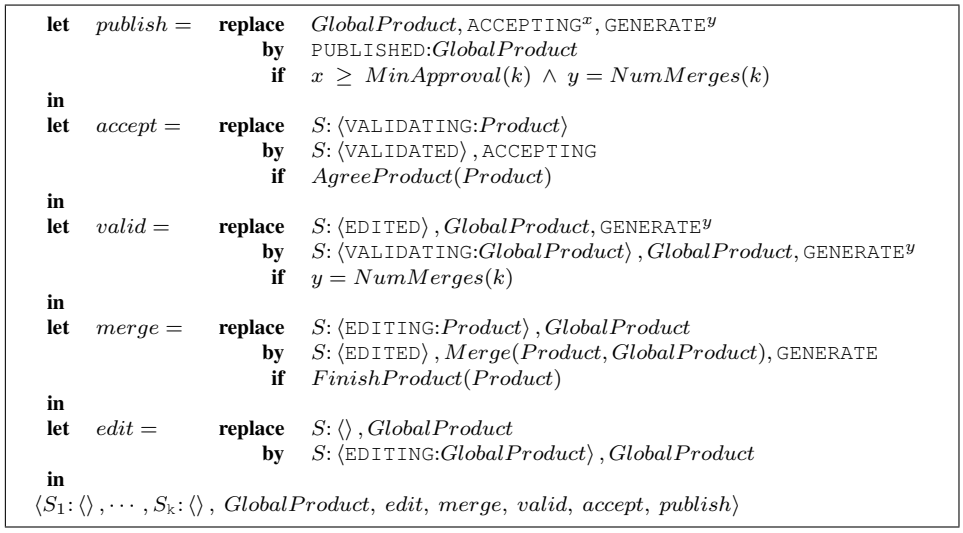

Figure 2: HOCL Program for a simple content management system

4 Security Patterns for Chemical Programming

A design pattern is a general reusable solution to a commonly occurring problem in software design [7]. For instance, object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Design patterns can be applied to achieve security goals such as confidentiality, integrity or availability [12].

In this section we define security patterns for HOCL programs. These patterns serve as templates that guide the definition of security aspects by instantiating them with domain-specific information. We define patterns for important security properties, namely *Authorisation*, *Integrity*, and *Security Logs*.

4.1 Authorisation Pattern

Authorisation concerns with the verification that an entity can perform a particular action. In the context of chemical programs, authorisation refers to the verification that a reaction could occur in a solution. The authorisation pattern, described in Figure 3, indicates that whenever a solution S reacts using reaction R, the authorisation condition $Authorised(S, R)$ holds.

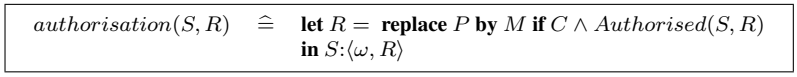

Figure 3: HOCL Pattern for Authorisation

The authorisation condition is considered as a generic condition that should be instantiated with domain-specific information. In this paper, we are interested in defining authorisation for three particular cases of attributed-based authorisation: role-based access control, authorisation based on trust values, and authorisation based on environmental conditions such as date, time, etc.

In the case of role-based access control, we associate solutions to roles and indicate which reactions can be executed by roles. Let $SolutionRole$ be a predicate associating a solution with a role, and $RoleReaction$ a predicate associating a role with a reaction. In this case the $Authorisation$ condition takes the form $SolutionRole(S, Rol) \wedge RoleReaction(Rol, R)$.

In the case of authorisation based on trust values, we assume there is a function $TrustValue(S)$ returning the trust value associated to a solution S. The $Authorisation$ condition is simply a predicate comparing the trust value of a solution with a particular value.

In the case of authorisation based on environmental conditions, we assume there are predicates such as $Date$ and $Time$ which could restrict when a reaction occurs.

4.2 Integrity Pattern

Integrity requirements prescribe that information is changed only in a specified and authorised manner. Integrity is violated when an employee (accidentally or with malicious intent) deletes important data files, when a computer virus infects a computer, or when someone is able to cast a very large number of votes in an online poll, among others.

Let $R =$ **replace** P **by** M **if** C be a reaction. The integrity condition $Integrity(P, M)$ is true if the transformation of pattern P by M is considered a valid one. In the context of chemical programs, integrity refers to the verification that the changes resulting from a reaction are consistent with the $Integrity$ predicate. The integrity pattern, described in Figure 4, indicates that whenever a solution S reacts using reaction R, both the authorisation condition $Authorised(S, R)$ and the integrity condition $Integrity(P, M)$ must hold.

$$integrity(S, R) \quad \widehat{=} \quad \textbf{let } R = \textbf{ replace } P \textbf{ by } M \textbf{ if } C \wedge Authorised(S, R) \wedge Integrity(P, M) \textbf{ in } S{:}\langle \omega, R \rangle$$

Figure 4: HOCL Pattern for Integrity

4.3 Security Log Pattern

In the case of security-critical operations, it might be required to maintain a security log of such operations. In chemical programming, this corresponds to storing in a log a reaction as well as the changes it has produced. Let $R =$ **replace** P **by** M **if** C be a reaction. The security log pattern, described in Figure 5, indicates that whenever reaction R happens, it is stored in solution Log a molecule with information about the solutions and molecules participating in R. The Log solution can be seen as a trusted third party in charge of storing and maintaining the security log.

$$logging(R) \quad \widehat{=} \quad \textbf{let } R = \textbf{ replace } P,\ Log{:}\langle \omega \rangle \textbf{ by } M,\ Log{:}\langle \omega, R{:}P{:}M \rangle \textbf{ if } C \textbf{ in } S{:}\langle \omega, R \rangle$$

Figure 5: HOCL Pattern for Security Logging

5 Security Aspects for Chemical Programming

Aspect-oriented programming (AOP) is a programming paradigm that explicitly promotes separation of concerns. In the context of security, aspects mean that the main program should not need to encode security information; instead, it should be moved into a separate, independent piece of code [15].

AOP is based on the idea that computer systems are better programmed by separately specifying the various concerns of a system and some description of their relationships, and then relying on mechanisms in the underlying AOP environment to weave or compose them together into a coherent program. The goal of AOP is to make designs and code more modular, meaning the concerns are localised rather than scattered and have well-defined interfaces with the rest of the system. This provides the usual benefits of modularity, including making it possible to reason about different concerns in relative isolation, making them (un)pluggable, amenable to separate development, and so forth.

This section introduces the main concepts of aspects and relates them with chemical programming. We then show how to apply aspects to include security in our VO for product generation.

5.1 Basic Concepts on AOP

Cross-cutting concerns are concerns whose implementation cuts across a number of program components. This results in problems when changes to the concern have to be made —the code to be changed is not localised but is in different places across the system. Cross-cutting concerns can range from high level notions like security and quality

of service to low-level notions such as caching and buffering. They can be functional, like features or business rules, or nonfunctional, such as synchronization and transaction management. The following are the main terminology used in AOP:

- *Join point*: Point of execution in the application at which cross-cutting concern needs to be applied. In the case of chemical programming, join points could be associated with reactions where the concerns need to be applied.
- *Advice*: This is the additional code that one wants to apply to an existing model. In the case of chemical programming, advice are applied to joint points (reactions) by adding/replacing some of the components of the reaction.
- *Aspect*: An aspect is an abstraction which implements a concern; it is the combination of a join point and an advice.
- *Weaving*: The incorporation of advice code at the specific joint points. There are three approaches to aspect weaving: source code pre-processing, link-time weaving, and execution-time weaving.

There is an additional concept called the *Kind of an Aspect* indicating if an advice is applied before, after, or around a join point. Since there is not a notion of sequentiality (execution order) in a chemical program, we do not exploit this feature. All aspects for chemical programming can be seen as around aspects.

5.2 Aspects for Chemical Programming

In this paper we have followed a code pre-processing technique to weave aspects in a chemical program. To do so, we represent aspects as a collection of transformation functions Ψ_{C_i}, each one modelling a different cross-cutting concern C_i. Each function Ψ_{C_i} is applied to a reaction and returns a modified version of the reaction that has been transformed according to the aspect.

Let $Reaction$ denote the set of reaction rules and Σ denote the state of a chemical program. State here refers to the solution and molecules participating in a program. The signature of a transformation function Ψ_C is defined as follows:

$$\Psi_C{:}Reaction \times \Sigma \rightarrow Reaction$$

As a way of illustration, let us define transformation Ψ_{RBAC} that applies the role-based authorisation concern to a reaction, indicating that a solution could react using a particular reaction if it is playing a role in the system. Function Ψ_{RBAC} takes as input a reaction, a solution name, and a role name, producing a new version of the reaction where the condition has been strengthened with the predicates $SolutionRole$ and $RoleReaction$, as presented in the authorisation pattern defined in sub-section 4.1. Upper part of Figure 6 shows the definition of the transformation function Ψ_{RBAC}. Let us assume that the $merge$ reaction in the VO system presented in Figure 2 can react when the solution containing it is playing the $Editor$ role. Lower part of Figure 6 shows the result of applying the transformation function Ψ_{RBAC} to $merge$.

$$\Psi_{RBAC}{:}Reaction \times SolutionName \times RoleName \rightarrow Reaction$$

$$\begin{array}{l}\forall R{:}Reaction, S{:}SolutionName, Rol{:}RolName \\ R = \textbf{replace } P \textbf{ by } M \textbf{ if } C \quad \rightarrow \\ \quad \Psi_{RBAC}(R, S, Rol) = \quad R = \textbf{replace } P \textbf{ by } M \\ \qquad\qquad\qquad\qquad\quad \textbf{if } C \wedge SolutionRole(S, Rol) \wedge RoleReaction(Rol, R)\end{array}$$

$$\begin{array}{lrl}\Psi_{RBAC}(merge, S, Editor) & = & \\ \quad merge & = \textbf{ replace} & S{:}\langle \texttt{EDITING}{:}Product\rangle\ ,\ GlobalProduct \\ & \textbf{by} & S{:}\langle \texttt{EDITED}\rangle\ ,\ Merge(Product, GlobalProduct),\ \texttt{GENERATE} \\ & \textbf{if} & FinishProduct(Product) \wedge \\ & & SolutionRole(S, Editor)\ \wedge\ RoleReaction(Editor, merge)\end{array}$$

Figure 6: Weaving a single aspect: applying the RBAC aspect to reaction $merge$.

In the case of having more than one aspect applied to a reaction, i.e. weaving several aspects at the same join point, the order in which the aspect functions is applied might be important. For our example, we will be defining aspect functions that are *commutative* according to the following definition.

Definition 1 (Commutativity of Aspects) *Let* Ψ_{C_i}:$Reaction \times \Sigma_{C_i} \rightarrow Reaction$ *and* Ψ_{C_j}:$Reaction \times \Sigma_{C_j} \rightarrow Reaction$ *be aspect functions. We say that* Ψ_{C_i} *and* Ψ_{C_j} *are commutative if the following property holds.*

$$\forall R{:}Reaction,\ T_{C_i}{:}\Sigma_{C_i},\ T_{C_j}{:}\Sigma_{C_j}\ .$$
$$\Psi_{C_i}(\Psi_{C_j}(R, T_{C_j}),\ T_{C_i})\ \equiv\ \Psi_{C_j}(\Psi_{C_i}(R, T_{C_i}),\ T_{C_j})$$

where $\equiv$ *denote syntactic equivalence.*

The commutativity definition indicates that the order in which the aspect functions is applied to a reaction is not important. Figure 7 illustrates the application of aspect functions Ψ_{RBAC} and Ψ_{LOG} to the $merge$ reaction. Upper part of the figure shows the definition of the Ψ_{LOG} function following the logging pattern defined in sub-section 4.3.

$\Psi_{LOG}{:}Reaction \times SolutionName \rightarrow Reaction$

$\forall R{:}Reaction,\ S{:}SolutionName$
$R =$ **replace** P **by** M **if** $C \rightarrow$
$\Psi_{LOG}(R, S) = R =$ **replace** $P,\ Log{:}\langle\omega\rangle$
by $M,\ Log{:}\langle\omega,\ S{:}R\rangle$
if C

$\Psi_{RBAC}(\Psi_{LOG}(merge, S), S, Editor) =$
$merge =$ **replace** $S{:}\langle \texttt{EDITING}{:}Product\rangle, GlobalProduct,\ Log{:}\langle\omega\rangle$
by $S{:}\langle \texttt{EDITED}\rangle, Merge(Product, GlobalProduct), \texttt{GENERATE},$
$Log{:}\langle\omega,\ S{:}merge\rangle$
if $FinishProduct(Product) \wedge$
$SolutionRole(S, Editor)\ \wedge\ RoleReaction(Editor, merge)$

Figure 7: Weaving several aspects: applying the RBAC and LOG aspects to reaction $merge$.

5.3 Securing the VO for Product Generation

Our approach for applying AOP techniques to chemical programs comprises the following steps. First, security requirements for the system under construction are defined. Second, the security requirements are modelled as aspect functions, following the security patterns introduced in section 4. Third, we define the join points where the aspects functions should be applied. Finally, aspects are weaved producing a new chemical program. The rest of this section describes the application of such approach for our VO for product generation.

5.3.1 Security Requirements for Product Generation

The system for product generation has the following security requirements:

1. Organisations participating in the VO could play the roles $Editor$ or $Validator$.
2. VO members playing the role $Editor$ can execute only operations related to the edit and merge phases of the workflow.
3. VO members playing the role $Validator$ can execute only operations related to the validate phases of the workflow.
4. Acceptance of a product is considered a security-critical operation requiring to be registered in a security log.
5. Acceptance is allowed only for those VO members with a trust value higher than 0.5.

5.3.2 Aspect Transformation Functions

Figure 8 shows the three aspect functions defined for our VO, according to the requirements presented above. Function Ψ_{RBAC} models role-based authorisation, following the authorisation pattern introduced in sub-section 4.1. We are assuming the underlying execution system includes functions $SolutionRole$, associating a solution with a role, and $RoleReaction$, associating a role with the reaction that can perform. Likewise, functionΨ_{TRUST} models authorisation

based on trust values, following also the pattern from sub-section 4.1. Here, it is assumed the existence of function $TrustValue$, returning the trust value of a solution. Finally, function Ψ_{LOG} models the secure log concern, following the pattern defined in sub-section 4.3.

Ψ_{RBAC}:$Reaction \times SolutionName \times RoleName \rightarrow Reaction$

$\forall R$:$Reaction, S$:$SolutionName, Rol$:$RolName$
$R =$ **replace** P **by** M **if** $C \quad \rightarrow$
$\Psi_{RBAC}(R, S, Rol) \quad = \quad R =$ **replace** P **by** M
if $C \wedge SolutionRole(S, Rol) \wedge RoleReaction(Rol, R)$

Ψ_{TRUST}:$Reaction \times SolutionName \times \Re \rightarrow Reaction$

$\forall R$:$Reaction, S$:$SolutionName, t$:$\Re$
$R =$ **replace** P **by** M **if** $C \quad \rightarrow$
$\Psi_{TRUST}(R, S, t) \quad = \quad R =$ **replace** P **by** M
if $C \wedge TrustValue(S) > t$

Ψ_{LOG}:$Reaction \rightarrow Reaction$

$\forall R$:$Reaction$
$R =$ **replace** S:P **by** M **if** $C \quad \rightarrow$
$\Psi_{LOG}(R) \quad = \quad R =$ **replace** S:P, Log:$\langle\omega\rangle$
by M, Log:$\langle\omega$, S:$R\rangle$
if C

Figure 8: Aspect functions for securing the VO for product generation.

5.3.3 Defining Join Points

Table 2 illustrates the joint points for our VO according to the requirements defined previously.

Requirement	Aspect
1, 2	$\Psi_{RBAC}(edit,\ S,\ Editor)$
1, 2	$\Psi_{RBAC}(merge,\ S,\ Editor)$
1, 3	$\Psi_{RBAC}(valid,\ S,\ Validator)$
1, 3	$\Psi_{RBAC}(accept,\ S,\ Validator)$
4	$\Psi_{LOG}(accept)$
5	$\Psi_{TRUST}(accept,\ S,\ 0.5)$

Table 2: Join points to apply aspect functions in order to secure the product generation VO

At this stage, we can see the modularity obtained by applying AOP techniques. Any change in the security requirements implies only changes in the definition of aspect functions and join points. For instance, if the requirement that the $accept$ reaction should be performed only by solutions with their trust above a particular value is removed, then the only changes required are to remove Ψ_{TRUST} function and to eliminate last row of Table 2.

5.3.4 Aspect Weaving

Finally, the aspects are weaved producing a new program. The chemical program resulting after weaving the aspects defined in Table 2 is presented in Figure 9.

6 Related Work

The work presented here has been inspired by Viega, Bloch and Chandra's work on applying aspect-oriented programming to security [15]. They have developed an aspect-oriented extension to the C programming language following also a transformational approach, where aspects are defined independently of the main application, and are then weaved

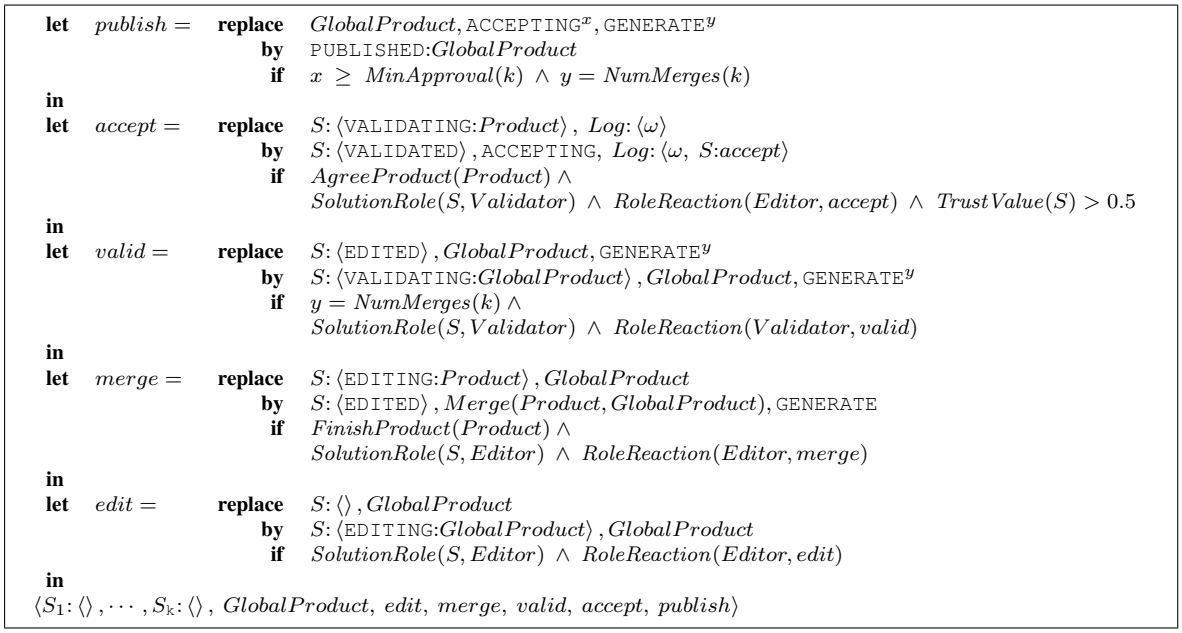

Figure 9: HOCL program for the VO system for product generation after weaving security aspects

into a single program at compilation time. Their emphasis is on security, developing aspects to replace insecure function calls by secure ones. Our approach follows a transformational approach as proposed by Viega, with the difference that the aspect definition is guided by the existence of security patterns.

Other efforts to apply aspects to security are presented in [16, 13, 4]. In [16], De Win *et al* report on the use of AspectJ [8] to secure application software, focusing on modelling access control as a cross-cutting concern. Their aspect model is applied to two real case studies: a personal information management system (PIM), and the File Transfer Protocol (FTP). Their results show the high degree of modularity that can be achieved by using aspects, and their model can be easily extended to support other authorisation models. The dynamic weaving of security aspects in service composition is described in [13]. It is achieved by separating the security control from other functional requirements, and encapsulating it into a service extension aspect. The woven aspect is then executed at run-time when services are composed. In [4], Cuppens *et al* transform availability requirements expressed in the Nomad model into availability aspects using AspectJ, which are then woven to secure a program.

Previous work on the application of aspect-oriented techniques to chemical programming include [9, 11, 10]. In [9], Mentré *et al* present the design of shared-virtual-memory protocols using the Gamma formalism; then, aspect-oriented techniques are used to translate this design into a concrete implementation, modelling cross-cutting concerns such as control and data representation. They also used a transformational approach, weaving at compilation time a Gamma program to produce an automaton. The work by Mousavi *et al* [11, 10] centred on extending Gamma with aspect-oriented concepts, including aspects for coordination, timing and distribution. For each aspect, they present new syntactic constructors and give them a structured operational semantics. The weaving process map the different aspects into a common formal semantics domain based on timed process algebra with relative intervals and delayable actions.

Close-related to our work is the application of aspects to high-order functional languages and term rewriting systems, since we plan as future work to study the run-time weaving of aspects, exploiting the high-order properties of HOCL. Aspectual Caml [14] is an aspect-oriented extension to the strongly-typed functional language Objective Caml. Aspectual Caml offers two AOP mechanisms, namely the pointcut and advice, as well as the type extension mechanism, which gives similar functionality to the inter-type declarations in AspectJ. Poly_{AML} [5] is a programming language that integrates polymorphism, run-time type analysis and aspect-oriented programming languages features. In particular, Poly_{AML} allows programmers to define type-safe polymorphic advice using pointcuts constructed from a collection of polymorphic join points.

7 Conclusion and Future Work

This paper has developed an aspect-oriented methodology for chemical programs written using the High-Order Chemical Language (HOCL). We represent aspects as a collection of transformation functions, each one modelling a different cross-cutting concern. The functions are applied (weaved) to a HOCL programs in order to generate a new program that include the concerns. Our case study has been a Virtual Organisation for the production of digital product, and the cross-cutting concerns have been security properties such as attribute-based authorisation and security logs.

We believe that our approach is generic enough and could be applied to other type of concerns such as fault-tolerant or real-time issues.

Our current approach is not fully exploiting the high-order potentiality of HOCL. We plan to tackle this as future work by weaving aspect at execution time.

Acknowledgments

Authors would like to thank Yann Radenac and Benjamin Aziz for valuable and insightful comments to earlier drafts of this report.

References

[1] J-P. Banâtre, P. Fradet, and Y. Radenac. Principles of Chemical Programming. In S. Abdennadher and C. Ringeissen, editors, *Proceedings of the 5th International Workshop on Rule-Based Programming*, volume 124(1) of *ENTCS*, pages 133–147. Elsevier, June 2005.

[2] J-P. Banâtre, P. Fradet, and Y. Radenac. Generalised Multisets for Chemical Programming. *Mathematical Structures in Computer Science*, 16(4):557–580, August 2006.

[3] L. M. Camarihna-Matos and H. Afsarmanesh, editors. *Collaborative Networked Organisations — A Research Agenda for Emerging Business Models*. Kluwer, 2004.

[4] F. Cuppens, N. Cuppens-Boulahia, and T. Ramard. Availability Enforcement by Obligations and Aspects Identification. In *ARES 2006, First International Conference on Availability, Reliability and Security*. IEEE Computer Society, 2006.

[5] D. S. Dantas, D. Walker, G. Washburn, and S. Weirich. Poly$_{\mathrm{AML}}$: a Polymorphic Aspect-Oriented Functional Programming Language. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005*, pages 306–319, 2005.

[6] I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, 15(3), 2001.

[7] E. Gamma, R. Helm, R. Johnso, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[8] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In J. L. Knudsen, editor, *ECOOP 2001 – Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–354. Springer, 2001.

[9] D. Mentré, D. Le Métayer, and T. Priol. Formalization and Verification of Coherence Protocols with the Gamma Framework. In *International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 2000)*, pages 105–113, 2000.

[10] M. R. Mousavi, M. A. Reniers, T. Basten, and M. R. V. Chaudron. Separation of Concerns in the Formal Design of Real-Time Shared Data-Space Systems. In *ACSD*, pages 71–81. IEEE Computer Society, 2003.

[11] M. R. Mousavi, G. Russello, M. R. V. Chaudron, M. A. Reniers, T. Basten, A. Corsaro, S. Shukla, R. Gupta, and D. C. Schmidt. Using Aspect-GAMMA in the Design of Embedded Systems. In *HLDVT '02: Proceedings of the Seventh IEEE International High-Level Design Validation and Test Workshop*, page 69, Washington, DC, USA, 2002. IEEE Computer Society.

[12] M. Schumacher, E. Fernandez, D. Hybertson, and F. Buschmann. *Security Patterns: Integrating Security and Systems Engineering*. John Wiley & Sons, 2005.

[13] H. Song, Y. Sun, Y. Yin, and S. Zheng. Dynamic Weaving of Security Aspects in Service Composition. In *SOSE '06. Second IEEE International Workshop on Service-Oriented System Engineering*. IEEE Computer Society, 2006.

[14] H. Tatsuzawa, H. Masuhara, and A. Yonezawa. Aspectual Caml: an Aspect-Oriented Functional Language. In *In Workshop on Foundations of Aspect Oriented Languages*, pages 320–330. ACM Press, 2005.

[15] J. Viega, J. T. Bloch, and P. Chandra. Applying Aspect-Oriented Programming to Security. *Cutter IT Journal*, 14(2):31–39, 2001.

[16] B. De Win, W. Joosen, and F. Piessens. Developing Secure Applications through Aspect-Oriented Programming. In *Aspect-Oriented Software Development*, pages 633–650. Addison-Wesley, 2005.