Developing Autonomic and Secure Virtual Organisations with Chemical Programming

Alvaro E. Arenas*, Jean-Pierre Banâtre† and Thierry Priol†

*STFC Rutherford Appleton Laboratory, UK
†INRIA Rennes - Bretagne Atlantique, France
alvaro.arenas@stfc.ac.uk, Jean-Pierre.Banatre@inria.fr,
Thierry.Priol@inria.fr

Abstract. This paper studies the development of autonomic and secure Virtual Organisations (VOs) when following the chemical-programming paradigm. We have selected the Higher-Order Chemical Language (HOCL) as the representative of the chemical paradigm, due mainly to its generality, its implicit autonomic property, and its potential application to emerging computing paragidms such as Grid computing and service computing. We have advocated the use of aspect-oriented techniques, where autonomicity and security can be seen as cross-cutting concerns impacting the whole system. We show how HOCL can be used to model VOs, exemplified by a VO system for the generation of digital products. We develop patterns for HOCL, including patterns for traditional security properties such as authorisation and secure logs, as well as autonomic properties such as self-protection and self-healing. 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

The concept of Virtual Organisation (VO) is given attention by researchers within a wide range of fields, from social anthropology and organisational theory to computer science. Its importance resides in providing an abstraction to represent organisational collaborations, a topic of fresh interest given the current exploitation of Internet to create virtual enterprises [5], or the sharing of resources across different organisations as envisaged by Grid computing [7].

This paper studies the development of VOs when using a chemical programming paradigm. Chemical programming is a computational paradigm inspired by the chemical metaphor, where computation is seen as reactions between molecules in a chemical solution. Examples of chemical-programming frameworks include P-Systems [13], the Higher-Order Chemical Language (HOCL) [1] and Fraglets [14], among others. Potentiality of the paradigm has been shown by its application to solve problems as diverse as page ranking of biochemical databases [12], coordination of services [3], or protocol resilience [15].

A 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. Autonomicity is an important property in VOs, since coalition members should act autonomously in order to achieve the VO goals. The chemical programming paradigm is very relevant to the programming of autonomic systems as it captures the intuition of a collection of cooperative components which evolve freely according to some predefined constraints (reaction rules). Security is also an important concern in VOs, since such a coalition may include unknown organisations that are untrusted by other VO partners.

We introduce here a method for modelling autonomic and secure VOs in HOCL using aspect-oriented techniques. We have selected HOCL as the language representative of the chemical paradigm, due mainly to its generality, its implicit autonomic property — HOCL is based on the Gamma calculus [1], which is also the foundation of other chemical frameworks such as Fraglets — and its potential application to emerging computing paradigms such as service computing [3]. We define first a set of patterns for HOCL programs, representing security and autonomic properties. Each property is modelled then as an aspect, defined using the patterns, which is weaved following a code pre-processing technique.

The structure of the paper is the following. Section 2 introduces HOCL. Section 3 discusses the autonomic properties of HOCL and describes its application to VOs, exemplified by a system for the generation of digital products. Section 4 presents security patterns for chemical programs. Section 5 describes the use of aspect-oriented techniques in HOCL. Next, section 6 shows how to apply the security patterns by using aspect-oriented programming. Section 7 relates our work with others. Finally, section 8 concludes the paper and highlights future work.

2 The Higher-Order Chemical Language

In this section we introduce the main features of HOCL, referring the reader to [2] for a more complete presentation. 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.

In HOCL, a chemical solution is represented by a multiset and reaction rules specify multiset rewritings. 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:

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

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. For example, consider the program of Figure 1 that computes the prime numbers lower than 10 using a chemical version of the Eratosthenes' sieve.

$$\textbf{let } \textit{sieve} = \textbf{replace } x, y \textbf{ by } x \textbf{ if } x \texttt{ div } y \textbf{ in}$$
$$\langle \textit{sieve},\ 2, 3, 4, 5, 6, 7, 8, 9, 10 \rangle$$

Fig. 1. Chemical prime numbers program.

The rule *sieve* reacts with two integers x and y such that x divides y, and returns x (i.e. removes y). Initially several reactions are possible, for example $\textit{sieve}, 2, 8$ (replaced by $\textit{sieve}, 2$) or $\textit{sieve}, 3, 9$ (replaced by $\textit{sieve}, 3$) or $\textit{sieve}, 2, 10$, etc. The solution becomes inert when the rule *sieve* cannot react with any couple of integers in the solution, that is to say, when the solution contains only prime numbers. The result of the computation in our example is $\langle \textit{sieve},\ 2, 3, 5, 7 \rangle$.

An important feature of HOCL is the notion of *multiplets*. 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 exponentiation of constants or patterns, indicating that the size of a multiplet becomes dynamic.

3 Virtual Organisations in HOCL

3.1 Autonomicity in HOCL

Autonomic computing provides a vision in which systems manage themselves according to some predefined goals. The essence of autonomic computing is self-organisation. Like biological systems, autonomic systems maintain and adjust their operation in the face of changing components, workloads, demands and external conditions, such as hardware or software failures, either innocent or malicious. The autonomic system might continually monitor its own use and check for component upgrades. HOCL is very appropriate as a programming model to express programs with autonomic behaviours. The reason is twofold. First, HOCL is intrinsically dynamic: rules are executed until an inert state is reached. When the multiset is modified, then reactions rules are executed to achieve again the inertness. Secondly, the high-order promoted by HOCL allows some policies to be replaced at runtime by new ones. Policies can be expressed by a set of rules that are stored in the multiset and thus can be replaced thanks to the execution of some other rules (high-order). An autonomic system is implemented using control loops that monitor the system and executes a set of operations to keep its parameters within a desired scope. A control loop has four basic steps: *monitor*, *analyse*, *plan* and *execute*. All these steps can be mapped onto chemical objects. *Monitor* and *execute* can be represented by external input/output operations into the multiset by generating molecules whereas *analyse* and *plan* are a set of chemical rules that express the autonomic behavior. A simple autonomic mail system [2] has been developed as an example of programming self-organisation with HOCL.

3.2 Programming Autonomic Virtual Organisations in HOCL

We model here a VO with the goal of generating products resulting from the collaboration of several dispersed organisations, which possesses 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 2. 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*.

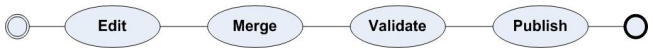

Fig. 2. Workflow process for the VO supporting the generation of a product.

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 3 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

$$
\begin{array}{l}
\textbf{let } \mathit{publish} = \textbf{replace } \mathit{GlobalProduct}, \texttt{ACCEPTING}^{x}, \texttt{GENERATE}^{y} \\
\qquad\qquad\qquad\quad \textbf{by } \texttt{PUBLISHED:}\mathit{GlobalProduct} \\
\qquad\qquad\qquad\quad \textbf{if } x \geq \mathit{MinApproval}(k) \wedge y = \mathit{NumMerges}(k) \\
\textbf{in} \\
\textbf{let } \mathit{accept} = \textbf{replace } S\colon\langle \texttt{VALIDATING:}\mathit{Product}\rangle \\
\qquad\qquad\qquad\quad \textbf{by } S\colon\langle \texttt{VALIDATED}\rangle, \texttt{ACCEPTING} \\
\qquad\qquad\qquad\quad \textbf{if } \mathit{AgreeProduct}(\mathit{Product}) \\
\textbf{in} \\
\textbf{let } \mathit{valid} = \textbf{replace } S\colon\langle \texttt{EDITED}\rangle, \mathit{GlobalProduct}, \texttt{GENERATE}^{y} \\
\qquad\qquad\qquad\quad \textbf{by } S\colon\langle \texttt{VALIDATING:}\mathit{GlobalProduct}\rangle, \mathit{GlobalProduct}, \texttt{GENERATE}^{y} \\
\qquad\qquad\qquad\quad \textbf{if } y = \mathit{NumMerges}(k) \\
\textbf{in} \\
\textbf{let } \mathit{merge} = \textbf{replace } S\colon\langle \texttt{EDITING:}\mathit{Product}\rangle, \mathit{GlobalProduct} \\
\qquad\qquad\qquad\quad \textbf{by } S\colon\langle \texttt{EDITED}\rangle, \mathit{Merge}(\mathit{Product}, \mathit{GlobalProduct}), \texttt{GENERATE} \\
\qquad\qquad\qquad\quad \textbf{if } \mathit{FinishProduct}(\mathit{Product}) \\
\textbf{in} \\
\textbf{let } \mathit{edit} = \textbf{replace } S\colon\langle\rangle, \mathit{GlobalProduct} \\
\qquad\qquad\qquad\quad \textbf{by } S\colon\langle \texttt{EDITING:}\mathit{GlobalProduct}\rangle, \mathit{GlobalProduct} \\
\textbf{in} \\
\langle S_1\colon\langle\rangle, \cdots, S_{\mathrm{k}}\colon\langle\rangle,\ \mathit{GlobalProduct},\ \mathit{edit},\ \mathit{merge},\ \mathit{valid},\ \mathit{accept},\ \mathit{publish}\rangle
\end{array}
$$

Fig. 3. HOCL Program for collaborative generation of a digital product

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. Function $MinApproval(k)$ is an abstraction of the protocol used to decide when to publish a product; for instance, if it is equal to $ceil(k/2)$, we would be using a majority vote protocol.

4 Patterns for Chemical Programming

A composition pattern is a design model that specifies the design of a cross-cutting requirement independently of any design it may potentially cross-cut, and how that design may be re-used wherever it may be required [6]. In this section we define composition patterns for HOCL programs. These patterns serve as templates that guide the definition of aspects by instantiating them with domain-specific information. We define patterns for important security properties, namely *Authorisation* and *Security Logs*; as well as patterns for autonomic properties such as *Self-Protection* and *Self-Healing*.

Authorisation Pattern. Authorisation is concerned 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 4, indicates that whenever a solution S reacts using reaction R, the authorisation condition $Authorised(S, R)$ holds.

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

Fig. 4. 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.

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.

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

Fig. 5. HOCL Pattern for Security Logging

Self-Protection Pattern. Self-protection refers to the ability of anticipating problems, and taking steps to avoid or mitigate them. It can be decomposed in two phases: a detection phase and a reaction phase [9]. The detection phase consists mainly in filtering data (pattern matching). The reaction phase consists in preventing offensive data from spreading and sometimes also in counter-attacking. This mechanism can easily be expressed with the condition-reaction

scheme of the chemical programming. Figure 6 shows the self-protection pattern. Function $Filter$ rule out undesirable data; on the other hand, function $Protect$ represents the application of a protection mechanism to the rest of the data.

$$selfprot(S,R) \widehat{=} \textbf{let } R = \textbf{replace } P,Q \textbf{ by } Protect(Q) \textbf{ if } Filter(P) \textbf{ in } S{:}\langle\omega, R\rangle$$

Fig. 6. HOCL Pattern for Self-Protection

Self-Healing Pattern. Another important autonomic property is self-healing, which refers to the automatic discovery and correction of faults in a system. We define a pattern for the case in which a partner in a VO — represented as a solution— fails by replacing it by a *back-up partner*. The back-up partner offers his own resources while the original partner cannot contribute to the VO objective. Functions $Failure(S)$ and $Recover(S)$ are associated to the system functionality capable of detecting whether a system has failed or recovered from a previous problem.

$$\begin{aligned} fail(S) &\widehat{=} \textbf{replace } S{:}\langle\omega\rangle \textbf{ by } S_{backup}{:}\langle\omega\rangle \textbf{ if } Failure(S) \\ repair(S) &\widehat{=} \textbf{replace } S_{backup}{:}\langle\omega\rangle \textbf{ by } S{:}\langle\omega\rangle \textbf{ if } Recover(S) \end{aligned}$$

Fig. 7. HOCL Pattern for Self-Healing

5 Aspects for Chemical Programming

Aspect-oriented programming (AOP) is a 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 [16].

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.

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 Defining Aspects for Chemical Programming

In this work 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: Ψ_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. Upper part of Figure 8 shows the definition of the transformation function Ψ_{RBAC}. Let us assume that the *merge* reaction in the VO system presented in Figure 3 can react when the solution containing it is playing the *Editor* role. Lower part of Figure 8 shows the result of applying the transformation function Ψ_{RBAC} to *merge*.

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

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

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

Fig. 8. Weaving an aspect: applying the RBAC aspect to reaction *merge*.

6 Applying Patterns and AOP to 'Chemical' VOs

In general, our approach for applying AOP techniques to chemical programs comprises the following steps. First, requirements for the system under construction are defined. Second, the requirements are modelled as aspect functions, following the 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 to the VO for product generation introduced in section 3.2.

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.
6. The system must check automatically that any product to be merged is free of virus.
7. The VO member assigned to location 1, i.e. the member identified as S_1, is considered critical one and must be replaced by a back up member in case of failure.

Requirements 1 to 5 are classical security requirements; requirement 6 is a self-protection one; and requirement 7 is a self-healing requirement.

Aspect Transformation Functions. Figure 9 shows the aspect functions defined for our VO to deal with the security requirements presented above, and Figure 10 illustrates the aspect functions defined for self-protection and self-healing requirements.

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

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

$$\Psi_{TRUST}\colon Reaction \times SolutionName \times \Re \rightarrow Reaction$$

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

$$\Psi_{LOG}\colon Reaction \rightarrow Reaction$$

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

Fig. 9. Aspect functions for securing the VO for product generation.

In Figure 9, function Ψ_{RBAC} models role-based authorisation, following the authorisation pattern introduced in sub-section 4. 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, function Ψ_{TRUST} models authorisation based on trust values, following also the pattern from sub-section 4. 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.

In Figure 10, function $\Psi_{NOVIRUS}$ models self-protection according to the pattern presented in subsection 4. Here, we are assuming there is a system function called $NoVirus$ in charge of checking there is not virus in a digital product. On the other hand, functions Ψ_{FAIL} and $\Psi_{RECOVER}$ model self-healing according to the pattern presented previously.

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

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, without altering the business logic

$$\Psi_{NOVIRUS}\colon Reaction \rightarrow Reaction$$

$$\forall R\colon Reaction$$
$$R = \textbf{ replace } S\colon \langle \texttt{EDITING}\colon P \rangle, \omega \textbf{ by } S\colon \langle \texttt{EDITED} \rangle, M \textbf{ if } C \quad \rightarrow$$
$$\Psi_{NOVIRUS}(R) = R = \textbf{ replace } S\colon \langle \texttt{EDITING}\colon P \rangle, \omega \textbf{ by } S\colon \langle \texttt{EDITED} \rangle, M \textbf{ if } C \wedge NoVirus(P)$$

$$\Psi_{FAIL}\colon SolutionName \times SolutionName \rightarrow Reaction$$
$$\Psi_{RECOVER}\colon SolutionName \times SolutionName \rightarrow Reaction$$

$$\forall S, S_{backup}\colon SolutionName$$
$$\Psi_{FAIL}(S, S_{backup}) = fail = \textbf{ replace } S\colon \langle \omega \rangle \textbf{ by } S_{backup}\colon \langle \omega \rangle \textbf{ if } Failure(S)$$
$$\Psi_{RECOVER}(S, S_{backup}) = recover = \textbf{ replace } S_{backup}\colon \langle \omega \rangle \textbf{ by } S\colon \langle \omega \rangle \textbf{ if } Recover(S)$$

Fig. 10. Aspect functions for self-protection and self-healing in the VO for product generation.

Requirement	Aspect	Requirement	Aspect
1, 2	$\Psi_{RBAC}(edit,\ S,\ Editor)$	5	$\Psi_{TRUST}(accept,\ S,\ 0.5)$
1, 2	$\Psi_{RBAC}(merge,\ S,\ Editor)$	6	$\Psi_{NOVIRUS}(merge)$
1, 3	$\Psi_{RBAC}(valid,\ S,\ Validator)$	7	$\Psi_{FAIL}(S_1, S_{1_{backup}})$
1, 3	$\Psi_{RBAC}(accept,\ S,\ Validator)$	7	$\Psi_{RECOVER}(S_1, S_{1_{backup}})$
4	$\Psi_{LOG}(accept)$		

Table 2. Join points to apply aspect functions to the product generation VO

of the program. 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 the corresponding rule in Table 2.

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 11. For instance, comparing reaction *merge* with the original version presented in Figure 3, we can notice that the condition of the rule has been strengthened restricting the execution only to solutions playing the role *Editor* and when the product to be generated is free of any virus.

7 Related Work

The work presented here has been inspired by Viega, Bloch and Chandra's work on applying aspect-oriented programming to security [16]. 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 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

Fig. 11. HOCL program for the VO system for product generation after weaving aspects

security patterns. Previous work on the application of aspect-oriented techniques to chemical programming include [10, 11]. In [10], 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. Comparing with our work, they also used a transformational approach, weaving at compilation time a Gamma program to produce an automaton; however, they do not represent cross-cutting concerns as patterns. The work by Mousavi *et al* [11] centred on extending Gamma with aspect-oriented concepts, including aspects for 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. Our work has the advantage that there is not need of changing the underlying semantic model (all our aspects are in HOCL) and exploiting the existence of composition patterns.

8 Conclusion and Future Work

This paper has described an approach to program autonomic and secure Virtual Organisations (VOs) when using the Higher-Order Chemical Language (HOCL). Our approach is based on composition patterns and aspect-oriented techniques. 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 program in order to generate a new program that include the concerns.

Our working example has been a VO for the production of digital product, and the cross-cutting concerns have been security properties such as attribute-based authorisation and security logs, as well as autonomic properties such as self-protection and self-healing. The approach comprises the following steps. First, security requirements for the system under construction are defined. Second, the requirements are modelled as transformational aspect functions following a library of compositional patterns. Third, it is defined the join points where the aspects functions should be applied. Finally, aspects are weaved producing a new chemical program.

There are several avenues to follow as future work. Firstly, we are currently studying the weaving of several aspects on the same reaction, analysing conditions that guarantee properties such as commutativity and associativity of aspects. Secondly, we plan to investigate patterns for weaving aspects at run-time, exploiting the high-order potentiality of HOCL. Thirdly, we are interested in evaluating the effectiveness of our approach to improve modularisation of cross-cutting concerns in HOCL; an initial step is to adapt quantitative methods to evaluate AOP [8]. Finally, there are several similarities between chemical programming and other evolutionary approaches such as genetic programming [4]; we plan to investigate how our approach to secure and autonomic cooperations can be applied when using genetic programming.

Acknowledgments

This work has been partially funded by the EU CoreGRID (IST FP6 No 004265) and GridTrust (IST FP6 No 033817) projects. We would like to thank Yann Radenac and Benjamin Aziz for comments to early drafts of this paper.

References

1. J-P. Banâtre, P. Fradet, and D. Le Métayer. Gamma and the Chemical Reaction Model: Fifteen Years After. In *Multiset Processing*, volume 2235 of *Lecture Notes in Computer Science*, pages 17–44. Springer-Verlag, 2001.
2. J-P. Banâtre, P. Fradet, and Y. Radenac. Chemical Specification of Autonomic Systems. In *Proceedings of the 13th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE'04)*, July 2004.
3. J-P. Banâtre, T. Priol, and Y. Radenac. Service Orchestration Using the Chemical Metaphor. In Springer, editor, *Software Technologies for Embedded and Ubiquitous Systems*, volume 5287 of *Lecture Notes in Computer Science*, pages 79–89, 2008.
4. W. Banzhaf, J.R. Koza, C. Ryan, L. Spector, and C. Jacob. Genetic Programming. *Intelligent Systems and their Applications, IEEE*, 15(3):74–84, May/Jun 2000.
5. L. M. Camarihna-Matos and H. Afsarmanesh, editors. *Collaborative Networked Organisations — A Research Agenda for Emerging Business Models*. Kluwer, 2004.
6. S. Clarke and R. J. Walker. Composition Patterns: An Approach to Designing Reusable Aspects. *International Conference on Software Engineering*, 2001.
7. I. Foster, C. Kesselman, and S. Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *Int. Journal of Supercomputer Applications*, 15(3), 2001.
8. A. Garcia, C. Sant Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa. Modularizing Design Patterns with Aspects: A Quantitative Study. In *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*, pages 36–74. Springer, 2006.
9. J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, 2003.
10. D. Mentré, D. Le Métayer, and T. Priol. Formalization and Verification of Coherence Protocols with the Gamma Framework. In *Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 2000)*, pages 105–113, 2000.
11. 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.
12. M. Muskulus. Application of Page Ranking in P Systems. In *9th Workshop on Membrane Computing*. IEEE Computer Society, 2008.
13. Gheorghe Păun. *Membrane Computing. An Introduction*. Springer-Verlag, Berlin, 2002.
14. C. Tschudin. Fraglets - a Metabolistic Execution Model for Communication Protocols. In *2nd Annual Symposium on Autonomous Intelligent Networks and Systems (AINS)*. IEEE Computer Society, 2003.
15. C. Tschudin and L. Yamamoto. A Metabolic Approach to Protocol Resilience. In *WAC 2004, 1st Workshop on Autonomic Communication*, volume 3457 of *Lecture Notes in Computer Science*, pages 191–206. Springer, 2004.
16. J. Viega, J. T. Bloch, and P. Chandra. Applying Aspect-Oriented Programming to Security. *Cutter IT Journal*, 14(2):31–39, 2001.