Towards a Chemical Model for Software Architecture Reconfiguration

Michel Wermelinger

Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
2825 Monte da Caparica, Portugal
E-mail: mw@di.fct.unl.pt

Abstract

The Chemical Abstract Machine is a general-purpose, simple, and intuitive programming model. Among other domains, it has been used for the specification and analysis of the computational behaviour of software architectures. In this paper we explore the ability of the formalism to express the dynamics of the architecture itself and to unify different approaches to reconfiguration within a single framework.

1 Introduction

1.1 Motivation

Software Architecture has been an emerging discipline that focuses on the high-level design of complex systems. Usually systems have to be reconfigured in order to cope with new human needs (i.e., new requirements), new technology (e.g., new implementation), or a new environment (e.g., if one of the components fails). Hence the specification of the evolution of software architectures has been of concern [1]. There are several issues to be taken into account:

specification Changes can be specified implicitly or explicitly. Self-organising architectures [2] belong to the first case. Ideally, the architecture "knows" what to do when a reconfiguration triggering event (like component failure, or addition of a new component) occurs. The architecture designer would just provide the constraints on the architecture properties. However, most current approaches, including the ones to be mentioned next, have explicit *reconfiguration commands* to remove and add components and connections.

management Reconfiguration commands can be executed by the components themselves [3] or processed by some external configuration manager which coordinates the change process [4].

time Changes may be executed off-line, when the system is shut down, or on-line, while it is running. The latter is called dynamic reconfiguration. In this case

there are additional constraints on the evolution of the architecture: the configuration manager, given a change script (i.e., a sequence of reconfiguration commands), must execute it causing the least possible disruption and keeping the system always in a consistent state [4, 5, 6].

source Commands are given by the user or triggered by the system itself based on its current state. This is called, respectively, ad-hoc and programmed reconfiguration [7]. In the latter approach, a change script is executed when the state of the system satisfies some conditions. The fundamental problem is that, contrary to ad-hoc reconfiguration, the script is written together with the initial architecture, but it may be executed when the architecture has already changed. The solution has been to use queries [7] or path expressions [8] that assess the current architecture, and provide the actual components and connections to be used as arguments of the reconfiguration commands.

constraints A further problem in reconfiguration, whether ad-hoc or programmed, is that the changes to be performed might violate some structural properties of the architecture, and in that case system evolution has to be constrained. It can be done *a posteriori* or *a priori*. An example of the first approach is [8], where Prolog predicates check the architecture after each change. If some integrity property has been violated, the reconfiguration must be undone. The second method is typical of programmed reconfiguration approaches and is exemplified by [7]. Each change script has a pre-condition that checks whether the architecture in which the script will be executed satisfies some properties. If it does not, the reconfiguration will not be performed.

In this paper we explore the suitability of the chemical reaction model [9] for the specification and analysis of architecture reconfiguration, hoping to provide a first step towards a model that unifies the above mentioned approaches. The actual formalism to be used is the Chemical Abstract Machine (CHAM) [10].

1.2 The CHAM model

The chemical model views computation as a sequence of reactions between data elements, called *molecules*. The structure of molecules is defined by the designer. The system state is described by a multiset of molecules, the *solution*. The possible reactions are given by rules of the form

$$m_1, \ldots, m_i \rightarrow m'_1, \ldots, m'_j$$

where $m_{1\ldots i}$ and $m'_{1\ldots j}$ are molecules. If the current solution contains the molecules given on the left-hand side of a rule, that rule may be applied, replacing those molecules by the ones on the right-hand side. Usually a CHAM is presented using *rule schemata*, the actual rules being instances of those schemata. There is no control mechanism. At each moment, several rules may be applied, and the CHAM chooses one of them non-deterministically. Reactions on disjoint multisets can occur simultaneously, i.e., in parallel. The solution thus evolves by rewriting steps. A solution is *inert* when no reaction rule can be applied.

As a very simple example, consider a CHAM to build ring-shaped architectures. The initial solution is a multiset of components, each of the form —c— stating that it may be connected to two other components. A molecule of the form —c—c—. . . —c— may still grow. When the first component of a linear architecture is linked to the last

one, a ring is obtained: c—c—. . . —c. The molecules are thus built from the constant "c" and from the operator "—" as given by the grammar

$$\begin{array}{ll} \mathit{Ring} & := \mathrm{c}\text{—}\mathit{Ring} \mid \mathrm{c} \\ \mathit{Molecule} & := \text{—}\mathit{Ring}\text{—} \end{array}$$

and rings are formed according to the reaction rules

$$\begin{array}{c} \text{—}x\text{—}, \text{—}y\text{—} \rightarrow \text{—}x\text{—}y\text{—} \\ \text{—}x\text{—} \rightarrow x \end{array}$$

If the initial solution has four components, up to four rings can be built. For example, the following transformations lead to two rings: —c—, —c—, —c—, —c— —→ —c—c—, —c—, —c— —→ —c—c—, c, —c— —→ —c—c—c—, c —→ c—c—c, c. Notice that the second transformation (corresponding to an application of the second reaction rule) could occur in parallel with any of the other transformations.

1.3 Software Architecture and the CHAM

The CHAM was first used to specify and analyse software architectures by Inverardi and Wolf [11]. Other kinds of analysis performed on the CHAM model are architecture refinement [12] and deadlock detection [13]. These works describe and study only static architectures: the number and type of components and connections do not change.

Le Métayer's proposal to deal with dynamic architectures views an architecture as a graph, where nodes denote components and arcs represent connections, and an architectural style as a class of graphs [14]. Reconfigurations are specified by graph rewriting rules and architectural styles are given by context-free grammars. Moreover, [14] provides a general method to check whether the rewriting rules keep the architectural style or not, i.e., whether the resulting graph belongs to the same class as the original one.

Our inspiration for this paper came from two observations: 1) grammars and rewriting rules are basically CHAMs with one kind of molecules to represent non-terminal symbols and two kinds of molecules for terminal symbols (the graph's nodes and arcs); 2) the graph rewriting approach is suited for self-organizing architectures. Our preliminary proposal can be summarized as follows.

Architectural style and reconfiguration are specified by two different CHAMs, the creation CHAM and the evolution CHAM. The former uses the initial solution to impose global system integrity constraints (e.g., the maximal number of components) while the reaction rules enforce local component integrity properties (e.g., the number of bindings [2]).

The reaction rules of the evolution CHAM specify how the solution that describes the architecture can be transformed. The reconfigured architecture is obtained when the solution becomes inert. If a CHAM's reaction rules do not consume nor generate a reconfiguration command, then it is self-organization; if they only consume, it is ad-hoc change; but if they also produce commands on the right-hand sides, then it is programmed reconfiguration. Even if the creation CHAM is not context-free (i.e., at least one of its rules has more than one molecule on its left-hand side), it may be possible to prove straightforwardly that the evolution CHAM keeps the architectural style by showing (through inspection) that its rules are invariants regarding the style's

properties. The evolution CHAM is thus used to specify, analyse, and simulate architecture reconfiguration. The actual process is efficiently carried out by a configuration manager whose input script can be obtained from the CHAM specification.

The next three sections show how the CHAM can be used to describe the three forms of reconfiguration. We follow the Configuration Programming approach pioneered by Kramer and colleagues. The goal is to separate structural from computational and interaction aspects. Thus the evolution of the structure of a system should preferably be specified at the architectural level only, not at the component level. To this end we use different molecules to represent the architecture and the components' state. Moreover, we use explicit reconfiguration commands like those of [4, 7, 5, 6]. Since the reaction rules state how components are linked, the user only has to provide *create component* (cc) and *remove component* (rc) commands.

The CHAM is of course no universal panacea for architectural changes. Since each rule fires due to the actual presence of some molecules in the solution, reconfigurations which depend on "negative" or global conditions (e.g., "when no fast server of is available, connect all clients to a slow server"), instead of "positive" and local ones, are not the most suitable for our approach, although the use of counters may solve the problem in some cases, as the last example will show.

2 Ad-hoc Reconfiguration

To illustrate the specification and analysis of architectural style and ad-hoc reconfiguration we adopt Le Métayer's example [14]. It is a client-server system with a centralized manager. A client sends a request to the manager which forwards it to an available server. The server's reply is sent back to the correct client via the manager.

2.1 Specification

The structure of molecules is given by the following grammar, which leaves the precise syntax of component identifiers open.

$$
\begin{array}{ll}
\mathit{Molecule} & := \mathit{Component} \mid \mathit{Link} \mid \mathit{Command} \\
\mathit{Component} & := \mathit{Id}{:}\mathit{Type} \\
\mathit{Type} & := \mathrm{C} \mid \mathrm{M} \mid \mathrm{S} \\
\mathit{Link} & := \mathit{Id}\text{—}\mathit{Id} \\
\mathit{Command} & := \mathrm{cc}(\mathit{Component}) \mid \mathrm{rc}(\mathit{Id})
\end{array}
$$

To make the example more interesting we assume that there must always be at least one server. The CHAM that specifies the client-server architectural style is

$$
\begin{aligned}
\mathrm{cc}(m{:}\mathrm{M}) &\rightarrow c{:}\mathrm{C},\ c\text{—}m,\ \mathrm{cc}(m{:}\mathrm{M}) \\
\mathrm{cc}(m{:}\mathrm{M}) &\rightarrow s{:}\mathrm{S},\ m\text{—}s,\ \mathrm{cc}(m{:}\mathrm{M}) \\
s{:}\mathrm{S},\ \mathrm{cc}(m{:}\mathrm{M}) &\rightarrow s{:}\mathrm{S},\ m{:}\mathrm{M}
\end{aligned}
$$

Assuming that the initial solution given by the user contains just the creation command cc() with the manager's name, the first rule adds a client and its link, the second rule adds a server and its connection, and the last rule actually creates the manager, if at least one server has been previously created.

Although syntactically there is no difference between this creation CHAM and the evolution CHAM to be presented next, the distinction can be made just by looking at the rules. A creation CHAM is used to generate *all* architectures belonging to a

certain style. This entails two properties of creation CHAMs. First, there are no rc() commands, because components and links are only added, not removed. Second, components may be created on the right-hand side without a cc() command on the left-hand side, in order to allow an arbitrary number of components (and their connections) to be generated.

Now we turn to the evolution specification. Besides adding and deleting clients as in [14], we also deal with server and manager creation and removal. Each change must be explicitly invoked by an appropriate command, to be handled by (at least) one reaction rule of the reconfiguration CHAM.

$$\begin{aligned}
\text{cc}(c\text{:C}),\ m\text{:M} &\to c\text{:C},\ c\text{—}m,\ m\text{:M} \\
\text{cc}(s\text{:S}),\ m\text{:M} &\to s\text{:S},\ m\text{—}s,\ m\text{:M} \\
\text{rc}(c),\ c\text{:C},\ c\text{—}m &\to \\
s'\text{:S},\ \text{rc}(s),\ s\text{:S},\ m\text{—}s &\to s'\text{:S} \\
m\text{:M},\ \text{rc}(m),\ \text{cc}(m'\text{:M}) &\to m'\text{:M} \\
m\text{—}s,\ m'\text{:M} &\to m'\text{—}s,\ m'\text{:M} \\
c\text{—}m,\ m'\text{:M} &\to c\text{—}m',\ m'\text{:M}
\end{aligned}$$

The first four rules deal with client and server creation and removal, while the other rules handle manager substitution, which is indicated by a pair of creation/removal commands. The last two rules relink the existing clients and servers to the new manager. Notice that we assume different variables to be instantiated with different identifiers. Otherwise the right-hand sides would be instances of the left-hand sides. In other words, those two rules could be immediately reapplied (although provoking no change in the architecture) and the solution would never become inert.

This CHAM illustrates ad-hoc reconfiguration because the reconfiguration commands appear only on the left-hand sides of rules. In other words, the commands are only consumed by the CHAM and thus must have been put into the solution by the user. As an example of reconfiguration, let us assume that we have an architecture with a single server (and manager, of course), and we want to add a client and replace the manager. The initial solution for the evolution CHAM is

m1:M, m1—s1, s1:S, cc(c1:C), rc(m1), cc(m2:M)

and the states of the solution until it becomes inert are

1. m1:M, m1—s1, s1:S, c1:C, c1—m1, rc(m1), cc(m2:M)
2. m2:M, m1—s1, s1:S, c1:C, c1—m1
3. m2:M, m2—s1, s1:S, c1:C, c1—m1
4. m2:M, m2—s1, s1:S, c1:C, c1—m2

2.2 Analysis

In general, to make sure that the specification is correct, it is necessary to prove that a CHAM terminates, i.e., that an inert solution can be reached. Usually this involves some assumptions on the initial solution. For our example style we are assuming the initial solution contains just one molecule of the form cc(m:M). Then it is quite easy to prove that the style CHAM always terminates (assuming fairness in rule selection): the third rule consumes the cc() command that is necessary for any of the rules to be triggered. Hence the computation terminates.

Another issue is to prove that the architectures generated by the creation CHAM are really those that we intended. Towards that end it is necessary to write down the properties of the architectural style and then, given the initial solution, prove that any inert solution obeys those properties.

Returning to our example, the properties of the client-server style are:

- there is exactly one manager;
- there are $x \geq 0$ clients, each one linked to the manager;
- there are $y > 0$ servers, each one linked to the manager.

As an illustration, we just prove that the third proposition is true of the creation CHAM. If the solution is inert, then there is no cc(m:M) command because otherwise the first two reaction rules could be applied. Since there is such a command in the initial solution, it must have been consumed somehow. By inspection of the rules, this is only possible by the third reaction rule. However, that rule can only have been applied if there existed a server. Since no rule decreases the number of servers, it is proven that at least one server must have been created and that it has not been removed by the application of some other rule. As for the server links, the only rule that creates servers connects them to the component whose name is given by the cc() command. This completes the proof of the third property, assuming that while proving the first one it has been established that the manager's name is the one given by the cc() command.

Sometimes it is necessary to prove that a reconfiguration does not "break" the style. For some properties this can be done inductively: prove that the initial solution of the reconfiguration CHAM satisfies the property and that each rule keeps it. The first part is usually not needed since it is assumed that the initial solution is either an inert solution of the creation CHAM (and thus satisfies the properties as proven before) or it is the inert solution of a previous reconfiguration (and therefore satisfies the properties as it will be proven by inspection of the rules). It thus suffices to prove that for each rule $L \rightarrow R$, if it is applied to a solution S that satisfies the property, then $S - L \uplus R$ also satisfies it.

As an illustration we prove that the client-server reconfiguration CHAM keeps at least one server. Let y (resp. y') be the number of servers immediately *before* (resp. *after*) the application of a rule. One has to prove that $y > 0 \Rightarrow y' > 0$ for each rule. The second rule states that $y' = y + 1$, the fourth rule that $y \geq 2 \Rightarrow y' = y - 1$, and for the remaining rules $y' = y$. It is obvious that for each one the implication is true.

However, the second part of the third property, namely that each server is linked to the manager, cannot be proven in this way because it is not an invariant of the system. In fact, due to rule 5 of the evolution CHAM, the solution does not represent a graph temporarily: there are links m—s but there is no m! The connectivity property can thus only be established for inert solutions. The proof goes as follows. First show that there is always exactly one manager. Next prove that there is always exactly one connection m—s for each server s. Finally show that for inert solutions, if m:M is the manager and m'—s is a server connection, then $m = m'$. The first two statements can be proven inductively, the third results from the fact that in an inert solution the last two rules of the evolution CHAM cannot be applied.

2.3 Dynamic Reconfiguration

Since a CHAM does not have any control mechanism, the exact order in which the reactions take place is unknown and cannot be predicted. This is no problem if the

reconfiguration takes place when the system is shutdown. However, in dynamic reconfiguration the changes occur while the system is running. In that case it is of paramount importance to execute the reconfiguration actions in such a way that the system is kept consistent and that disruption is minimized. That has been the object of the work of several researchers [4, 5, 6]. Their goal is to provide an algorithm for the *configuration manager* to execute the set of reconfiguration commands provided by the user in the correct order. We adopt the four primitive commands used in the works mentioned to create and remove components and links: create(), delete(), link(), and unlink(). Notice that our cc() and rc() commands are high-level create() and delete() commands, respectively, that also deal with the links "automatically".

We separate concerns by using the CHAM just to specify *what* to do, letting the configuration manager decide *how* to do it. To that end we let the CHAM "trace" its execution, creating a "log" of the changes performed. That log corresponds to the change script that a user would input directly to the configuration manager. In other words, the CHAM can be seen as a "compiler" of high-level reconfiguration commands into low-level ones to be executed by the "run-time system", i.e., the configuration manager.

The molecule syntax is extended with

$$\mathit{Command} := \text{create}(\mathit{Component}) \mid \text{delete}(\mathit{Id}) \mid \text{link}(\mathit{Id}, \mathit{Id}) \mid \text{unlink}(\mathit{Id}, \mathit{Id})$$

and the reconfiguration CHAM becomes

$$\begin{aligned}
\text{cc}(c{:}\text{C}),\ m{:}\text{M} &\rightarrow c{:}\text{C},\ c\text{—}m,\ m{:}\text{M},\ \text{create}(c{:}\text{C}),\ \text{link}(c, m)\\
\text{cc}(s{:}\text{S}),\ m{:}\text{M} &\rightarrow s{:}\text{S},\ m\text{—}s,\ m{:}\text{M},\ \text{create}(s{:}\text{S}),\ \text{link}(m, s)\\
\text{rc}(c),\ c{:}\text{C},\ c\text{—}m &\rightarrow \text{delete}(c),\ \text{unlink}(c, m)\\
s'{:}\text{S},\ \text{rc}(s),\ s{:}\text{S},\ m\text{—}s &\rightarrow s'{:}\text{S},\ \text{delete}(s),\ \text{unlink}(m, s)\\
m{:}\text{M},\ \text{rc}(m),\ \text{cc}(m'{:}\text{M}) &\rightarrow m'{:}\text{M},\ \text{delete}(m),\ \text{create}(m'{:}\text{M})\\
m\text{—}s,\ m'{:}\text{M} &\rightarrow m'\text{—}s,\ m'{:}\text{M},\ \text{unlink}(m, s),\ \text{link}(m', s)\\
c\text{—}m,\ m'{:}\text{M} &\rightarrow c\text{—}m',\ m'{:}\text{M},\ \text{unlink}(c, m),\ \text{link}(c, m')
\end{aligned}$$

For the reconfiguration example shown in Section 2.1, the generated change commands are:

create(c1:C), link(c1, m1), delete(m1), create(m2:M),
unlink(c1, m1), link(c1, m2), unlink(m1, s1), link(m2, s1)

As shown in the example, if the commands are executed in this order then temporarily some arcs do not point to any existing component. Moreover, two commands cancel out. Looking at such a script, it is easy for the configuration manager to optimize and reorder the commands, based on such simple rules as "a component to be removed must have no connections" and "a component must be created before it can be connected" [6]. In this case, the configuration manager may execute the following sequence of commands (other sequences are possible):

unlink(m1, s1), create(c1:C), delete(m1), create(m2:M), link(c1, m2), link(m2, s1)

3 Self-Organizing Architectures

The chemical model is well suited to describe self-organizing architectures where external explicit management is kept to a minimum [2]. In fact, the very essence of the

CHAM model is that molecules react freely with each other until the solution "stabilizes", i.e., becomes inert. Once that state is reached, new reactions may be triggered by adding new molecules, but the reaction process itself is purely an "internal affair". The evolution of the solution proceeds without any intervention from the outside.

Our client-server example illustrates this. Once the commands to substitute the manager are given, the architecture reorganizes itself to maintain the right connections, without needing any further commands from the user. In this section we provide a more elaborate example of self-organization: a *n*-ary tree architecture where components can be removed without destroying the properties of the tree. Such a topology might be useful for divide-and-conquer problems, each component splitting the data it gets from its parent and combining the results produced by its children.

The example is a generalization of the binary tree architecture presented in [8]. Following the Configuration Programming approach, [8] only considers structural properties. In this case, the *structural integrity* to be kept by the reconfiguration action is the binary tree shape. Such integrity constraints are divided into *node integrity* and *system integrity* properties. The former are local, the latter global. In this example, being a tree is a system constraint because no single node can ensure that the graph is acyclic. On the other hand, it is enough for each node to restrict the number of children to at most two in order to have a binary tree. Other examples of system integrity constraints are the number of components in the tree and that the system consists of only one tree.

The approach taken in [8] to handle structural integrity properties is verification. The properties are expressed as Prolog clauses used to check the architecture after each change. If the reconfiguration violates at least one of the constraints, it must be undone. We follow the self-organization approach of [2]: when a change occurs, the system components reorganize themselves in order to satisfy the structural constraints.

But first let us specify the *n*-ary tree architectural style. The reaction rules generate trees with a maximum branching factor given by the initial solution. A node is represented by a molecule n_c^a where n is the node's name, a is the number of its ancestors (i.e., its depth), and c is the number of children the node has. A root node has no ancestors and therefore its depth is zero. Natural numbers are represented as usual, using the constant zero and the successor function (written as a postfix +). Numbers are compared using substring prefix matching: $n \geq m$ if n is of the form mx, and $n > m$ if n is of the form $m + x$, in both cases x being matched by a sequence of zero or more $+$.

$$\begin{aligned} \mathit{Molecule} &:= \mathit{Node}\begin{matrix}\mathit{Nat}\\ \mathit{Nat}\end{matrix} \mid \mathit{Node}\text{—}\mathit{Node} \mid \mathit{Nat} \\ \mathit{Nat} &:= \mathit{Nat}+ \mid 0 \end{aligned}$$

The initial solution contains just the root node and the maximal branching factor: r_0^0, $0++$. The creation CHAM is

$$\begin{matrix} x_c^a,\ c{+}w \rightarrow x_{c+}^a,\ x\text{—}y,\ y_0^{a+},\ c{+}w \\ b+ \rightarrow \end{matrix}$$

The first rule creates a new node and attaches it to an existing node that has not yet exceeded the children limit. The second rule allows the solution to become inert by eliminating the branch factor.

The initial solution of the evolution CHAM contains the branching factor again and the current architecture. When a node fails, a command to remove it is added to the solution, and the architecture reconfigures itself according to the rules

$$\begin{aligned}
\mathrm{rc}(x),\ x\text{—}y,\ x^{0}_{c+},\ y^{0+}_{c'} &\to \mathrm{rc}(x),\ y\text{—}x,\ x^{0+}_{c},\ y^{0}_{c'+}\\
\mathrm{rc}(x),\ y\text{—}x,\ x\text{—}z,\ y^{a}_{c'},\ x^{a+}_{c+},\ z^{a++}_{c''} &\to \mathrm{rc}(x),\ y\text{—}x,\ y\text{—}z,\ y^{a}_{c'+},\ x^{a+}_{c},\ z^{a+}_{c''}\\
\mathrm{rc}(x),\ y\text{—}x,\ y^{a}_{c+},\ x^{a+}_{0} &\to y^{a}_{c}\\
b,\ x\text{—}y,\ x\text{—}z,\ x^{a}_{b+w},\ y^{a+}_{c} &\to b,\ x\text{—}y,\ y\text{—}z,\ x^{a}_{bw},\ y^{a+}_{c+}\\
x\text{—}y,\ x^{a}_{c},\ y^{a}_{c'} &\to x\text{—}y,\ x^{a}_{c},\ y^{a+}_{c'}\\
x\text{—}y,\ x^{a}_{c},\ y^{a++}_{c'} &\to x\text{—}y,\ x^{a}_{c},\ y^{a+}_{c'}
\end{aligned}$$

The first rule handles the case of the root node: it is swapped with one of its children, which thus becomes the new root. The former root node now is a middle node or a leaf node and the second or third rule applies, respectively. The second rule links the children of the middle node directly to its parent. The node hence becomes a leaf node and we get to the third rule which effectively removes the node from the tree, updating the children counter of the parent. During this process some nodes may have more than b children, where b is the branching factor. The fourth rule ensures the correct number by demoting the exceeding children to grandchildren. The last two rules propagate the correct depth to children of nodes that have been promoted or demoted. The simplicity and conceptual elegance of these rules should be compared with the parameterized recursive rule of [8] which uses four different kinds of path expressions and a marking command.

Some comments about this example are in order. The notation has been chosen to be as compact as possible, showing only the relevant data. Furthermore, using superscripts for depth and subscripts for children makes both of them stand out. Thus it is easier to see how a node changes from the left to the right side of a rule. A more conventional notation is obtained by translating every molecule of the form x^a_c into two molecules "depth(x, a), children(x, c)", and by transforming x—y into "linked(x, y)".

The specification can be simplified by omitting the depth of each node, as in the original example [8]. In fact, all that is necessary is to be able to distinguish the root node from the others. This can be done just with an extra molecule "root(x)". The first rule becomes

$$\mathrm{rc}(x),\ \mathrm{root}(x),\ x\text{—}y,\ x_{c+},\ y_{c'} \to \mathrm{rc}(x),\ \mathrm{root}(y),\ y\text{—}x,\ x_{c},\ y_{c'+}$$

the last two rules are not necessary any more, and the superscripts of the remaining ones disappear. However, we chose to have this additional difficulty because it introduces further self-organization.

Besides illustrating self-organization and allowing comparison of our approach with a previous one, the example shows how structural and cardinality constraints on architectures can be specified. This is also useful for other kinds of topologies.

4 Programmed Reconfiguration

This section shows how a single CHAM may combine the specification of the computational behaviour with the specification of the architectural evolution in order to describe programmed reconfiguration.

The chosen example is a client-server system where the server is a printing system with a set of printers. Each client makes a single request to print a document. Printers may break down while printing. If no printer is working then any new client requests are immediately rejected.

The chosen architecture consists of zero or more clients, one name server, one "inhibitor" and one composite component containing zero or more printers. If there are no working printers, the name server is linked to the "inhibitor", otherwise it is linked to the printing system. Whenever a new client arrives, it is automatically linked to the name server whose name is known and whose job is to relink the client to the component the name server is linked to. The "inhibitor" just rejects any client request. The printing system accepts any incoming request if there is an available printer. Otherwise the request stays on hold until a printer becomes available. During that time no further request can be sent to the printing system. Clients automatically leave the system after getting a positive or negative reply. When the last working printer breaks down, the name server is unlinked from the printing system and linked to the "inhibitor". Inversely, if no printer is working but one of them starts working again (e.g., because a user has reset it), the name server is linked back to the printing system.

We now present the CHAM. The syntax of molecules is given by the following grammar, whose meaning will become clear from the explanation of the reaction rules.

Molecule	:= *Component* | *Connection* | *Data* | *Command*
Connection	:= *Id*—*Id*
Component	:= *Printer* | *Port*
Data	:= working(*Number*)
Command	:= cl(*Connection*) | rl(*Connection*)
Printer	:= idle | toprint(*Number*) | broken
Number	:= 1*Number* |
Port	:= *Id* | *Id*=*Message*
Id	:= *Letter Id* | *Letter*
Message	:= *Number* | print(*Number*) | accepted | rejected |

A possible initial solution is: p=, idle, working(1), i=, n—p. This states that the printing system has a port called "p" to receive requests and send replies, and there is a single working printer, which is initially idle. The "inhibitor" is represented as a single port "i", initially without messages. The name server is denoted by port "n" linked to port "p" of the printing system. There is no molecule "n=" because the name server does not send or receive messages.

Initially, a client is of the form *c*=*n*, where *n* is the number—in unary notation—of pages to print, and *c* the name of the port. Each client may use a different name. The following three rules describe the behaviour of a client. The first rule links it to the name server (thereby changing the state of the port in order to avoid duplicate connections) and the other two remove the client and its connection after getting a reply.

$$c{=}n \rightarrow c{=}\text{print}(n),\ c\text{—n}$$
$$c{=}\text{accepted},\ c\text{—}s \rightarrow$$
$$c{=}\text{rejected},\ c\text{—}s \rightarrow$$

The name server simply relinks a client to the current server (either the "inhibitor" or the printing system).

$$c\text{—n},\ \text{n—}s \rightarrow c\text{—}s,\ \text{n—}s$$

The "inhibitor" replies to any incoming request with a rejection. As for the printing system, it accepts any request provided there is an available printer.

$$\text{i=print}(n) \rightarrow \text{i=rejected}$$
$$\text{p=print}(n),\ \text{idle} \rightarrow \text{p=accepted},\ \text{toprint}(n)$$

Each printer outputs a document one page at a time. When reaching the end, it becomes available again. However, while printing a page, the printer may break down. If it is the last working printer, then the name server must get linked to the "inhibitor". To that end, the corresponding reconfiguration commands are generated to remove the old link and create the new one.

$$\begin{aligned}
\text{toprint}(1n) &\rightarrow \text{toprint}(n)\\
\text{toprint}() &\rightarrow \text{idle}\\
\text{toprint}(1n), \text{working}(11p) &\rightarrow \text{broken}, \text{working}(1p)\\
\text{toprint}(1n), \text{working}(1), &\rightarrow \text{broken}, \text{working}(), \text{rl(n—p)}, \text{cl(n—i)}
\end{aligned}$$

A broken printer can become idle again if it is reset or repaired. If there were no working printers, the connection between the name server and the "inhibitor" must be removed and a link to the printing system must be created.

$$\begin{aligned}
\text{broken}, \text{working}(1p) &\rightarrow \text{idle}, \text{working}(11p)\\
\text{broken}, \text{working}() &\rightarrow \text{idle}, \text{working}(1), \text{rl(n—i)}, \text{cl(n—p)}
\end{aligned}$$

In order for all this to work, general rules to remove and create connections and to exchanges messages between connected ports are necessary.

$$\begin{aligned}
\text{rl}(a), a &\rightarrow\\
\text{cl}(a) &\rightarrow a\\
p{=}msg, p\text{—}q, q{=}msg' &\rightarrow p{=}msg', p\text{—}q, q{=}msg
\end{aligned}$$

Consider the following scenario. Initially, there is only one printer, and a client requesting to print 2 pages. The printing system accepts the request and the client goes away. The printer prints the first page and then breaks down. The name server is relinked to the inhibitor and thus a new client arriving will get its request rejected.

1. c=11, n–p, p=, idle, working(1), i=
2. c=print(11), c–n, n–p, p=, idle, working(1), i=
3. c=print(11), c–p, n–p, p=, idle, working(1), i=
4. c=, c–p, n–p, p=print(11), idle, working(1), i=
5. c=, c–p, n–p, p=accepted, toprint(11), working(1), i=
6. c=accepted, c–p, n–p, p=, toprint(1), working(1), i=
7. n–p, p=, broken, working(), i=, rl(n–p), cl(n–i)
8. newc=1, p=, broken, working(), i=, cl(n–i)
9. newc=print(1), newc–n, n–i, p=, broken, working(), i=
10. newc=print(1), newc–i, n–i, p=, broken, working(), i=
11. newc=, newc–i, n–i, p=, broken, working(), i=print(1)
12. newc=, newc–i, n–i, p=, broken, working(), i=rejected
13. newc=rejected, newc–i, n–i, p=, broken, working(), i=
14. n–i, p=, broken, working(), i=

5 Concluding Remarks

Inverardi and others [11, 12, 13] have shown that the chemical reaction model [9], and in particular the Chemical Abstract Machine [10], is a useful tool to describe and study the computational behaviour of a system with a static architecture. In this paper we extended the work in two directions: first, to handle not a single architecture but whole classes of architectures, by specifying a style as a non-deterministic self-organized reconfiguration process starting with an "empty" architecture; second, to cope with dynamic architectures. In particular, we dealt with self-organized, ad-hoc, and programmed reconfiguration, and global and local structural constraints. Based on our preliminary exploration, we conclude that the CHAM may be used for the specification and analysis of software architecture style and reconfiguration due to following characteristics of the chemical model.

simplicity There is a single data structure (multiset of terms) and a single programming construct (rewrite rules), both of which are familiar and intuitive concepts. The specifications tend thus to be rather compact and easy to write and read. Moreover, proving properties about architectures or their reconfigurations is normally straightforward (although tedious), often based on induction over the structure of molecules and rules.

suitability The model's view of "computation as the global evolution of a collection of atomic values interacting freely" [9] is naturally suited to describe the evolution of self-organising architectures, which is the most general case of reconfiguration. Also, the combination of reaction rules (the interactions) and initial solution of molecules (the atomic values) can be used to specify both system and node integrity properties [8].

flexibility The definition of the molecules is left to the designer. Hence, a molecule can represent an element of the architecture (i.e., a component or a connection), a reconfiguration command or rule, or auxiliary data (like counters). As for components, it is possible to represent their structural and computational aspects. The former include the number of connections a component has, the latter describe the component's state. As for reconfiguration commands, they may be high-level (like the "cc" command which also creates new links) or low-level ones (like the "create" command). With all these options one can encode and compare several models of reconfiguration described in the literature within a uniform framework. With the CHAM, the diverse approaches are easily recognizable according to where molecules that represent reconfigurations (like "rc" and "cc") appear in the reaction rules: in programmed reconfiguration they appear on both sides, in ad-hoc reconfiguration they appear only in left-hand sides, and in self-organization they do not appear at all.

However, flexibility has some disadvantages too. As seen in the previous sections, each CHAM introduces its own syntax and assumptions. This requires additional explanations and does not allow reuse of specifications. The solution is to have a fixed representation for the most important concepts (like component and connection). Towards that goal we have made a first attempt at a minimal CHAM-based architecture description language [15].

In future work we plan to investigate the means to provide support for partial mechanical analysis of complex properties and architectures. There are at least three possible independent lines of action.

- A CHAM is a term rewriting system (TRS) with an associative and commutative operator: multiset union. It might be possible to use or adapt techniques developed for TRS to prove termination of reconfiguration and uniqueness of the resulting architecture.
- The CHAM can be encoded in rewriting logic [16] and thus tools for that framework, like Maude [17] and ELAN [18], could be used to test architecture specifications written in CHAM. Another possibility is to adapt and expand sequential and parallel implementations of Gamma, the original chemical model [19, 20].
- If a given creation CHAM is context-free, the algorithm of [14] can be applied to check whether a given evolution CHAM keeps the style of an architecture generated by the creation CHAM. It remains to be seen if the method can be adapted to the general CHAM model or if a new algorithm can be developed.

We are of course aware that the chemical model is not suited for every kind of style or reconfiguration. Since a reaction depends on the presence of some molecules, reconfigurations that depend on "negative" or global conditions (e.g., "if every client is not connected to the printing system, then…") may be impossible or very hard to specify, leading to CHAMs that are cumbersome to write and hard to understand. We also expect several global integrity constraints to be not as easy to express as the branching factor or the depth of a tree. However, we hope that our exploration has reinforced the suggestion that "the CHAM model might be one useful tool in the software architect's chest of useful tools" [11].

Acknowledgements

We thank Daniel Le Métayer for helpful comments on an early draft. We are also indebted to an anonymous reviewer for pointing out several aspects which needed further clarification.

References

[1] Alexander L. Wolf. Succeedings of the Second International Software Architecture Workshop. *ACM SIGSOFT Software Engineering Notes*, 22(1):42–56, January 1997.

[2] Jeff Kramer and Jeff Magee. Self organising software architectures. In *Joint Proceedings of the SIGSOFT'96 Workshops*, pages 35–38. ACM Press, 1996.

[3] B. Agnew, C. Hofmeister, and J. Purtilo. Planning for change: a reconfiguration language for distributed systems. *Distributed Systems Engineering*, 1(5):313–322, September 1994.

[4] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, November 1990.

[5] Kaveh Moazami Goudarzi and Jeff Kramer. Maintaining node consistency in the face of dynamic change. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, pages 62–69. IEEE Computer Society Press, 1996.

[6] Michel Wermelinger. A hierarchic architecture model for dynamic reconfiguration. In *Proceedings of the Second International Workshop on Software Engineering for Parallel and Distributed Systems*, pages 243–254. IEEE Computer Society Press, 1997.

[7] Markus Endler. A language for implementing generic dynamic reconfigurations of distributed programs. In *Proceedings of the 12th Brazilian Symposium on Computer Networks*, pages 175–187, 1994.

[8] A. J. Young and J. N. Magee. A flexible approach to evolution of reconfigurable systems. In *Proceedings of the First International Workshop on Configurable Distributed Systems*, pages 152–163. IEE, 1992.

[9] Jean-Pierre Banâtre and Daniel Le Métayer. Gamma and the chemical reaction model: Ten years after. In Jean-Marc Andreoli, Chris Hankin, and Daniel Le Métayer, editors, *Coordination programming: mechanisms, models and semantics*, pages 3–41. Imperial College Press, 1996.

[10] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, (96):217–248, 1992.

[11] Paola Inverardi and Alexander L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine. *IEEE Transactions on Software Engineering*, 21(4):373–386, April 1995.

[12] Paola Inverardi and Daniel Yankelevich. Relating CHAM descriptions of software architectures. In *Proceedings of the 8th International Workshop on Software Specification and Design*, pages 66–74. IEEE Computer Society Press, 1996.

[13] Paola Inverardi, Alexander L. Wolf, and Daniel Yankelevich. Checking assumptions in component dynamics at the architecture level. In *Coordination Languages and Models*, volume 1282 of *LNCS*, pages 46–63. Springer-Verlag, 1997.

[14] Daniel Le Métayer. Describing software architecture styles using graph grammars. *IEEE Transactions on Software Engineering*, 24(7):521–553, July 1998.

[15] Michel Wermelinger. A simple description language for dynamic architectures. In *Proceedings of the Third International Software Architecture Workshop*. ACM Press, 1998. To appear.

[16] José Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In *Proceedings of the 7th International Conference on Concurrency Theory*, volume 1119 of *LNCS*, pages 331–372. Springer-Verlag, 1996.

[17] Manuel Clavel, Steven Eker, Patrick Lincoln, and José Meseguer. Principles of Maude. In *Proceedings of the First International Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*, pages 65–89. Elsevier, 1996.

[18] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. ELAN: A logical framework based on computational systems. In *Proceedings of the First International Workshop on Rewriting Logic*, volume 4 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1996.

[19] C. Creveuil. *Techniques d'analyse et de mise en œuvre des programmes Gamma*. PhD thesis, University of Rennes, 1991.

[20] Jean-Pierre Banâtre, A. Coutant, and Daniel Le Métayer. A parallel machine for multiset transformation and its programming style. *Future Generation Systems*, pages 133–144, 1988.