

Specification, Testing and Analysis of (Dynamic) Software Architecture with the Chemical Abstract Machine

Michel Wermelinger

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

ABSTRACT

We feel that the Chemical Abstract Machine (CHAM) is a useful formal description technique for static and dynamic software architectures that facilitates analysis and testing. Our position is based on work done so far and on the potential of rewriting approaches, of which the CHAM is a special case.

1 INTRODUCTION

The chemical model of computation was introduced by Banâtre and Le Métayer [2] and then refined by Berry and Boudol [3]. Inverardi and Wolf [7] were the first to apply it to Software Architecture but since then other papers have appeared [9, 8, 16, 15].

The next section briefly recapitulates the CHAM formalism for those readers not familiar with it. The third section surveys its application to Software Architecture and section 4 presents a simple example. The last two sections show some possible future research paths and make some concluding remarks.

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

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

where $m_{1\dots i}$ and $m'_{1\dots j}$ 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. A solution is *inert* when no reaction rule can be applied.

Any solution can be considered as a single molecule using the *membrane* operator $\{\cdot\}$. A solution within a membrane can thus be a subsolution of another solution or an argument

of a molecule operator. For example, if the molecule constructors are the constant 0 and the unary function s , then

$$s(\{0, s(0)\}), 0$$

is a solution containing two molecules. Membranes encapsulate molecules and thus force reactions to be local. In other words, the solution inside a membrane evolves independently of the solution outside the membrane. For example, if a CHAM included the rule

$$0 \rightarrow$$

then the above solution could be transformed into

$$s(\{s(0)\})$$

after two (possibly simultaneous) rewriting steps.

The *airlock* operator \triangleleft constructs a molecule $m \triangleleft \{S'\}$ from a solution $S = m \uplus S'$, where \uplus is the multiset union operator. In words, it picks a molecule from a solution and puts the rest of that solution within a membrane. The operator is reversible, which means that S can again be obtained from $m \triangleleft \{S'\}$. For example, using twice the airlock operator it is possible to transform the original solution into

$$0 \triangleleft \{s(\{s(0) \triangleleft \{0\}\})\}$$

3 SOFTWARE ARCHITECTURE AND THE CHAM

In [7] the CHAM was used to specify and analyze the computational behavior of static software architectures. That article uses the multi-phase compiler architecture as an example. For each component (phase) there is a molecule representing its state. Depending on their state, certain molecules may interact, exchanging data (characters, tokens, etc.). Two different architectures—a sequential and a concurrent one—were specified and it was proven that the latter terminates (for finite input) and subsumes the former. Using membranes to represent modular architectures, a third architecture was described with one membrane including the compiler's back-end while another contained its front-end. The correctness of

the module interfaces was also proven. Other kinds of analysis performed on the CHAM model are architecture refinement [9] and deadlock detection [8].

In [16, 15] we have proposed to apply the CHAM to dynamic software architectures, i.e., architectures that can be changed by the system itself (based on its current state or topology) or by the user (and thus in an unpredictable way). We call it programmed and ad-hoc reconfiguration, respectively, following [6]. To explicitly represent the topology and the user-requested changes we introduced molecules to represent connections and the four basic reconfigurations commands used by most languages ([6, 12, 1] among others), namely addition and removal of components and connections.

Related to programmed reconfiguration is the idea of self-organizing architectures [10]. The goal is to minimize the amount of explicit management. Ideally, the architecture “knows” what to do when a reconfiguration triggering event (like component failure, or addition of a new component) occurs. It is obvious that the CHAM is ideally suited for these architectures because the evolution of the solution (that represents such an architecture) depends only on the existing molecules, i.e., on the solution itself. This absence of an external control mechanism also means that the reactions will only stop when no reaction rule is applicable. Thus the CHAM is adequate to represent architectures with potentially infinite distinct configurations, something an approach like [1] cannot handle.

A previous proposal to deal with dynamic architectures [14] views an architecture as a graph, where nodes denote components and arcs represent connections, and an architectural style as a class of graphs. This means that 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.

This approach can be straightforwardly adapted to the chemical model because graph 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). We thus proposed [16] to specify architecture style and reconfiguration by two different CHAMs, the creation CHAM and the evolution CHAM. The former uses the initial solution to impose global system modification constraints (e.g., the maximal number of components) while the reaction rules enforce local component properties (e.g., the number of bindings [10]). The reaction rules of the evolution CHAM specify how the solution that describes the architecture can be transformed. 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.

In [15] we have also shown how a restricted form of code mobility can be easily achieved by encoding rules as molecules with a special operator symbol. Such molecules, like any other, can permeate through membranes to another part of the system where they are interpreted by rules whose left-hand sides check for molecules with the special syntax.

4 EXAMPLES

To illustrate the specification and analysis of architectural style and ad-hoc reconfiguration we will repeat the example presented in [16] which is an extension of the example used by Le Métayer [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. There must always be at least one server.

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

$$\begin{aligned} \text{Molecule} & := \text{Component} \mid \text{Link} \mid \text{Command} \\ \text{Component} & := \text{Id}:\text{Type} \\ \text{Type} & := \text{C} \mid \text{M} \mid \text{S} \\ \text{Link} & := \text{Id}-\text{Id} \\ \text{Command} & := \text{cc}(\text{Component}) \mid \text{rc}(\text{Id}) \end{aligned}$$

The CHAM that specifies the client-server style is

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

Assuming that the initial solution given by the user contains just the creation command $\text{cc}()$ 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.

Now we turn to the evolution of such an architecture. 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.

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

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

$$\begin{aligned} m1:M, m1-s1, s1:S, c1:C, c1-m1, rc(m1), cc(m2:M) \\ m2:M, m1-s1, s1:S, c1:C, c1-m1 \\ m2:M, m2-s1, s1:S, c1:C, c1-m1 \\ m2:M, m2-s1, s1:S, c1:C, c1-m2 \end{aligned}$$

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 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 will obey 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 will 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.

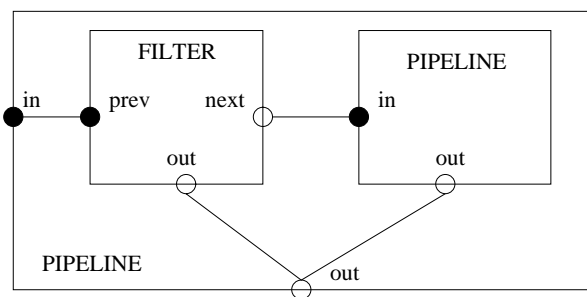
Sometimes it is necessary to prove that a reconfiguration will not “break” the style. For some properties this can be done inductively: prove that the initial solution of the reconfiguration CHAM satisfies the property and that each rule keeps it. The first part is usually not needed since it is assumed 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 will 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.

To illustrate programmed reconfiguration we show how the lazy pipeline described in [11] can be specified with a CHAM. The pipeline is a composite component with two communication ports called `in` and `out`. Internally, the pipeline consists of a filter and another pipeline. The architecture is thus recursive. A filter has three ports: an input port `prev` and two output ports `next` and `out`. The con-

nections to the enclosing and enclosed pipelines are shown in the following diagram.



The main characteristic of the example is that pipelines are only created on demand, in particular, when a message is sent to its in port. The example is used in [11] to illustrate the use of the `dyn` keyword of the Darwin architecture description language. We will show how it is possible to obtain the same effect with the CHAM.

First we define the syntax of molecules. We use positive integers in unary notation to uniquely identify components.

Molecule := *Component* | *Connection* | *Command*
Component := *Port* | *Number:Molecule*
Port := *PortName/Molecule* |
PortName= | *PortName=Message*
PortName := *Number.Id* | *Id*
Connection := *PortName—PortName*
Command := *cp(Number)*
Number := *1* | *1Number*

Next we need some general rules to describe communication between components. The first rule describes how a message is passed from a port p to a port q linked to p and ready to receive a message.

$$p=msg, p-q, q= \rightarrow p=, p-q, q=msg$$

The next two rules allow messages to cross membranes and thus permit communication between composite components. One rule makes a port visible to the environment, the other rule “internalizes” the port once it has received a message. After the component has processed the message, the former rule may be used again.

$$n:\{p= \triangleleft S\} \rightarrow n.p=, n:S$$

$$n.p=msg, n:S \rightarrow n:\{p=msg \triangleleft S\}$$

The last communication rule is the more important one for this example. It allows to attach an arbitrary solution to a port and add that solution to the system once the first message is received by or sent from that port. The solution may contain arbitrary reconfiguration commands, new components and connections. It thus generalizes Darwin’s `dyn` construct.

$$p=msg, p/\{S\} \rightarrow p=msg, S$$

For the example, only one reconfiguration command is necessary, namely to create a pipeline. We assume the initial solution contains the molecule $cp(1)$ to create the outermost pipeline according to the following rule:

$$cp(n) \rightarrow n:\{in=, out=, \\ 1n:\{prev=, next=, out=\}, \\ in-1n.prev, 1n.out-out, \\ 1n.next/\{cp(11n), \\ 1n.next-11n.in, 11n.out-out\}\}$$

When the filter sends its first message through port `next`, a new pipeline and its connections will be created. We have thus programmed reconfiguration and the recursive structure of the architecture is immediately apparent from the rule because the `cp()` command appears on both sides.

5 FUTURE WORK

Although work done so far already shows how the CHAM can be used to specify and analyze both static and dynamic architectures, it is possible to further explore this formalism.

On the one hand, a CHAM is a term rewriting system (TRS) with an associative and commutative operator (namely multiset union). We thus seek to use or adapt techniques developed for TRS to prove termination of reconfiguration and uniqueness of the resulting architecture.

On the other hand, the CHAM can be encoded in rewriting logic [13] and thus tools for that framework can be used to test architecture specifications written in CHAM. We will look at Maude [5] and ELAN [4]. The latter allows the separate specification of a computational system (in our case, the CHAM model), a particular instance (in our case, a CHAM representing a dynamic architecture style), and a query (a specific architecture with given reconfiguration commands). ELAN would then compute the resulting architecture, allowing the user to check whether the outcome is the expected one. The computation can be traced, which helps the user debug the specification.

6 CONCLUDING REMARKS

We perceive the following general advantages of the CHAM:

- It is a simple and operational programming model. There is a single data structure (multisets) 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.
- It is a general-purpose formalism. Its flexibility stems mainly from the fact that the definition of the molecules is left to the designer. This has a two fold advantage.

First, as much or as little detail can be included as necessary for the application at hand. Second, the most appropriate representation can be chosen for each element of the application.

- It is suitable for systems whose global evolution depends on local states (namely of those molecules that appear on the left-hand sides of rules). This makes it easier to build systems incrementally and to prove properties about them.

Of course, the CHAM is not a universal panacea, and therefore some architectures, especially those that rely on global state or negative properties may be hard or even impossible to specify. Also, since molecules are entirely built from the constants and operators defined by the user, architectures which need common data types (like booleans and integers) and operations (like sum and comparison) are a bit tedious and lengthy to specify, and not very readable.

However, all in all we feel that the CHAM is an adequate formalism for the specification, analysis and testing of certain classes of (possibly modular) static and dynamic software architectures due to its characteristics: it is simple, making proofs often straightforward; it is flexible, allowing the representation of components, connections, and re-configuration commands; it is an abstract formalism, subsuming other approaches like context-free grammars for architectural style specification; it is a term rewriting system, thus potentially making use of available theoretical results and practical tools.

ACKNOWLEDGEMENTS

The financial support of FCT through project PRAXIS XXI 2/2.1/TIT/1662/95 (SARA) is gratefully acknowledged.

REFERENCES

- [1] R. Allen, R. Douence, and D. Garlan. Specifying and analyzing dynamic software architectures. In *Fundamental Approaches to Software Engineering*, volume 1382 of *LNCS*, pages 21–37. Springer-Verlag, 1998.
- [2] J.-P. Banâtre and D. L. Métayer. Gamma and the chemical reaction model: Ten years after. In J.-M. Andreoli, C. Hankin, and D. L. Métayer, editors, *Coordination programming: mechanisms, models and semantics*, pages 3–41. Imperial College Press, 1996.
- [3] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, (96):217–248, 1992.
- [4] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. 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.
- [5] M. Clavel, S. Eker, P. Lincoln, and J. 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.
- [6] M. 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.
- [7] P. Inverardi and A. L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine. *IEEE Transactions on Software Engineering*, 21(4):373–386, Apr. 1995.
- [8] P. Inverardi, A. L. Wolf, and D. 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.
- [9] P. Inverardi and D. 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.
- [10] J. Kramer and J. Magee. Self organising software architectures. In *Joint Proceedings of the SIGSOFT'96 Workshops*, pages 35–38. ACM Press, 1996.
- [11] J. Magee and J. Kramer. Dynamic structure in software architectures. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 3–14. ACM Press, 1996.
- [12] N. Medvidovic. ADLs and dynamic architecture changes. In *Joint Proceedings of the SIGSOFT'96 Workshops*, pages 24–27. ACM Press, 1996.
- [13] J. Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *LNCS*, pages 331–372. Springer-Verlag, 1996.
- [14] D. L. Métayer. Software architecture styles as graph grammars. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 15–23. ACM Press, 1996.
- [15] M. Wermelinger. Software architecture evolution and the chemical abstract machine. In *International Workshop on the Principles of Software Evolution*, pages 93–97, Kyoto, Japan, Apr. 1998.
- [16] M. Wermelinger. Towards a chemical model for software architecture reconfiguration. In *Proceedings of the Fourth International Conference on Configurable Distributed Systems*, pages 111–118. IEEE Computer Society Press, 1998.