A Simple Description Language for Dynamic Architectures

Michel Wermelinger
Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
2825 Monte da Caparica, Portugal
mw@di.fct.unl.pt

ABSTRACT

The Chemical Abstract Machine (CHAM) is a very elegant and simple computational model that has been used for the specification of both static and dynamic architectures. However, so far those descriptions are ad-hoc and monolithic, thus making it difficult to perceive the elements and form of an architecture. In this paper we present some syntactic constructs and a methodology to specify an architecture in a principled, explicit, and modular way, thus proposing a very simple and compact Architecture Description Language (ADL) which is especially suited for dynamic architectures.

1 INTRODUCTION

One of the topics which is raising increased interest in the Software Architecture (SA) community is the ability to specify how a SA evolves over time, in particular at run-time, in order to adapt to new requirements or new environments. Indeed, the participants in the Architecture Description group of the last ISAW [14] felt that variability at the SA level was one of the important areas for future work because only few ADLs are able to express dynamism, as shown in [9]. As to be expected for any language or system, every ADL has its shortcomings. Darwin [8] only permits constrained dynamism: the initial architecture may depend on some parameters, and during run-time components may be replicated. To show the interaction between reconfiguration and the ongoing computations another formalism is used, Finite State Processes [7]. C2 also uses different languages and mechanisms to describe the architecture, the modifications, and the constraints on the modifications [11]. Wright’s formalism is a bit cumbersome since it requires all distinct configurations to be uniquely tagged [1]. ACME’s proposal is still preliminary and only allows for the specification of optional elements (i.e., components, connectors, and links) [10].

In previous work [13, 12] we proposed to use the CHAM [2] to specify dynamic SAs. The main advantage of the formalism is that it serves as a uniform framework to describe both the computations performed by the components, the changes done upon the architecture, and the constraints under which the reconfigurations can occur. Moreover, it allows different kinds of changes to be specified: programmed (i.e., the system itself triggers the reconfiguration process based upon its current state), ad-hoc (i.e., changes are triggered by the user), and self-organized (i.e., there are no explicit reconfiguration commands). However, the examples used in our works and in those by Inverardi and colleagues [3, 6, 4] are written in an ad-hoc fashion: each uses its own syntax which may be not very uniform, the topology is sometimes not explicit, components don’t have a recognizable interface, and all of the architecture is specified with a single CHAM. This last aspect was improved in [5], where the authors propose each component to be specified by its own CHAM. Each component specification uses meta-variables to stand for those components or links to which it will be connected. The architecture is then the union of all CHAMs together with functions mapping the meta-variables to the actual symbols. As the authors remark, the architecture cannot be changed, since the mappings are fixed by construction.

In summary, on the one hand there are (a few) ADLs which handle dynamism but with some restrictions, while on the other hand the CHAM is a generic theoretical framework for rewriting-based changes but it can’t be considered an ADL, as also noticed in [9]. In this paper we try to remedy this situation by presenting syntactic constructions for a concrete CHAM-based ADL for dynamic SAs and a method to specify architectures in a modular way. The appendix describes briefly the chemical model for those readers not familiar with
2 THE LANGUAGE

Our language is very simple for the moment. The rationale is to include further constructs only as necessary, when it is deemed essential for the description of dynamic architectures. Moreover, by committing just to a few design choices, it may serve as a least common ground between different existing ADLs.

There are only components and unidirectional links. The latter can be seen as wires: when there is a value on one end, it gets transferred to the other end. Unlike ACME, we do not require two kinds of elements (components and connectors) nor links only between different kinds of elements. This gives us freedom to describe architectures à la Darwin, which has no connectors, or C2, which allows connectors to be linked to connectors. The interface of a component is a set of ports, each being a name-value pair. The value of a port can obviously be changed by computation or communication. The interface itself may vary, i.e., ports may be added or deleted from a component.

To specify components we assume there is a context-free grammar describing the syntax of global types, like identifiers (Id), integers, and the messages communicated among components. A component is a molecule of the form name:type=[{state}], where name and type are two identifiers, and state is a solution describing the component’s current state. Notice that it is encapsulated within a membrane since the state is private to each component. A port is a molecule of the form name•value, where name is an identifier and value is of one of the previously declared message types. When ports are within the state solution they are not visible to the environment and therefore communication cannot occur through that port. It is up to the component, through its computations, to “export” ports to the outside and “import” them back again, thus signalling when it is ready to read or send a value and when it is processing or computing a port’s value, respectively.

Each component type is specified by a distinct CHAM consisting of three parts: a context-free grammar to describe the syntax of such component, a solution describing the initial state, and a set of rewriting rules describing the computations performed by the component. As the syntax of components is fixed, the grammar for type X only needs the rules:

\[
\begin{align*}
    \text{Port}_X & := \text{Id} \cdot \text{MsgType} | \ldots \\
    \text{State}_X & := \ldots
\end{align*}
\]

The first rule states the names of the ports and their types, and the second rule defines the syntax of molecules to describe additional state which will be needed for the component’s computations. These must be specified in such a way that whenever a port is exported, it is prefixed by the component’s name, so that later the port can be imported back again.

Notice that since each CHAM is self-contained, components are independently specified without any knowledge as to which components they will be linked. That is the purpose of the configuration manager. It describes the system’s architecture and how it changes. The manager is also specified by a CHAM: the grammar puts together the syntax of all component types, the solution describes the initial architecture, and the rewriting rules show how it may evolve. To be more precise, the grammar is the union of all component types’ grammars, except for the State rules, since the configuration manager has only access to ports of components. Moreover, some additional grammar rules define

- links of the form Port—Port;
- reconfiguration commands of the form cc(Id, Type), rc(Id), cl(Id, Id), and rl(Id, Id) which create and remove components and create and remove links, respectively;
- type declarations of the form X=Init which allow to create new components without repeating constantly their initial state.

As for the rules, the configuration manager must include the general communication rule

\[
p \rightarrow q, p\cdot M, q\cdot \rightarrow p \rightarrow q, p\cdot, q\cdot M
\]

stating that a message M is sent from port p to port q if they are connected, p has M, and q is ready to get it.

3 AN EXAMPLE

To illustrate the approach we use a very simple system with multiple clients and a single server.

3.1 The Client

A client has an output port “req” to send requests of the global type Query and an input port “rep” to get replies of type Answer. Initially, the client has generated a request internally.

\[
\begin{align*}
    \text{Port}_C & := \text{req} \cdot \text{Query} | \text{rep} \cdot \text{Answer} \\
    \text{Init}_C & := \{ \text{req} \cdot Q, \text{rep} \cdot A \}
\end{align*}
\]

The client interacts with its environment exporting the “req” port when it has a query and importing it again after the query has been sent.

\[
\begin{align*}
    c\cdot C=\text{req} \cdot Q \cdot S \rightarrow c\cdot \text{req} \cdot Q, c\cdot C=S \\
    c\cdot \text{req} \cdot, c\cdot C=S \rightarrow c\cdot C=\text{req} \cdot A \cdot S
\end{align*}
\]

The reply port is handled symmetrically.

\[
\begin{align*}
    c\cdot C=\text{rep} \cdot A \cdot S \rightarrow c\cdot \text{rep} \cdot, c\cdot C=S \\
    c\cdot \text{rep} \cdot A, c\cdot C=S \rightarrow c\cdot C=\text{rep} \cdot A \cdot S
\end{align*}
\]
A new query is generated based on the answer to the previous one.

\[ \text{rep} \cdot A, \text{req} \cdot \rightarrow \text{req} \cdot Q, \text{rep} \cdot \]

When the client wants to finish processing, it just processes the answer without generating a new query.

\[ \text{rep} \cdot A \rightarrow \text{rep} \cdot \]

### 3.2 The Server

As for the server, it has for each client \( c \) a pair of ports "req.\( c \)" and "rep.\( c \)" to get the requests and send the replies, respectively. These ports are always visible and they are added to (removed from) the server whenever \( c \) is added to (removed from) the architecture. Although it is not relevant for reconfiguration, for illustration purposes we assume the server caches the last query processed because it might help answering the next one. Initially the cache is empty.

\[
\begin{align*}
\text{Ports} & := \text{req.Id} \cdot \text{Query} \mid \text{rep.Id} \cdot \text{Answer} \\
\text{States} & := \text{cache(} \text{Query, Answer} \text{)} \\
\text{Init} & := \{ \text{cache(nil, nil)} \}
\end{align*}
\]

The server has a single rule to answer a query and update the cache.

\[
\begin{align*}
\text{s.req.c} \cdot Q, \text{s.rep.c} \cdot & \rightarrow \text{s.req.c} \cdot \text{s.rep.c} \cdot A, \\
\text{s:S=}[[\text{cache}(Q', A')] & \rightarrow \text{s:S=}[[\text{cache}(Q, A)]]
\end{align*}
\]

### 3.3 The Configuration Manager

The initial architecture has a single server and the client type declaration:

\[
\text{s:S=}[[\text{cache}(\text{nil, nil})]], \text{C} = \{ \text{req}\cdot Q, \text{rep}\cdot \}
\]

To add a client \( c \), simply add the molecule \( cc(c, C) \) to the solution and the following rule, based on the client type declaration and on the name of the server, will add the component and link it to two new ports of the server.

\[
\begin{align*}
\text{C=}I, \text{cc(c, C)}, \text{s:S=}S & \rightarrow \text{C=}I, \text{c:C=}I, \text{s:S=}S, \\
\text{c.req} & \rightarrow \text{s.req.c}, \text{s.req.c} \cdot, \\
\text{s.rep.c} & \rightarrow \text{c.rep}, \text{s.rep.c}
\end{align*}
\]

When removing a client, the links and the respective server ports are removed again. It is important however to do it in the right moment, when no request is pending. The configuration manager must thus look at the client’s interface and wait for the ports to have no messages and to be inside the client. Looking at the client’s computation rules it can be seen that in that state no new query can be generated. If the ports were outside, it would mean that a request had been sent but the reply hadn’t been received yet.

### 4 CONCLUDING REMARKS

We believe that the CHAM is a good model to specify changes to architectures due to its simplicity in the use of “if-then” rewriting rules, its flexibility to represent reconfiguration commands and components with encapsulated states, and its suitability to describe different kinds of changes [13, 12]. However, the CHAM lacked the syntactic constructs to serve as an ADL, i.e., to represent architectures in a modular way with explicit connections between component interfaces. This paper has shown that by imposing some constraints on the grammars used to describe the molecules’ syntax it is possible to achieve that goal without resorting to additional formalisms. Our position is thus that the CHAM can be used not only as a theoretical framework but also as a practical language, and the paper presented one such ADL. By including only a very small set of constructs we hope to contribute to a wider discussion on what are the minimal foundations for dynamic architectures and their description languages on which higher-level abstractions can be built.

### REFERENCES


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 $m_{1 \ldots j}$ 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. Thus rules of the form

$m_1, \ldots, m_i \rightarrow$

only remove molecules from the current solution. 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.

Any solution can be considered as a single molecule using the membrane operator $[[:]]$. 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 $\sigma$ constructs a molecule $m \sigma [:S:]$ 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 $m \sigma [:S']$. The operator is right-associative. For example, using twice the airlock operator it is possible to transform the original solution into

$0 \sigma [ [s(0) \sigma [:[[0]][[0]]]]]$

Molecules may permeate through membranes if there are explicit rules for that purpose. For example,


allows any “p” molecule to leave the membrane it is within.