Software architecture styles as graph grammars

Daniel Le Métayer
IRISA/INRIA
Campus de Beaulieu, 35042 Rennes, France
email: lemetayer@irisa.fr

Abstract

We present a formalism for the definition of software architectures in terms of graphs. Nodes represent the individual agents and edges define their interconnection. Individual agents can communicate only along the links specified by the architecture. The dynamic evolution of the overall architecture is defined independently by a 'coordinator'. An architecture style is a class of architectures characterised by a graph grammar. The rules of the coordinator are statically checked to ensure that they preserve the constraints imposed by the architecture style.

Keywords: coordination, graph rewriting, software architecture, static verification.

1 Motivation and approach

Software systems tend to grow in size and complexity; they are often developed through a long period of time and become extremely difficult to understand and to maintain. The cost incurred by this complexity is becoming a serious concern and a major challenge today is to provide ways of organising software in order to make large applications manageable and to favour the reuse of existing products. Several languages or systems have been proposed recently to tackle these problems: they are called software architecture languages [12], configuration languages [18] or coordination languages [6, 14]. Despite some differences of emphasis, these works share a common point of view: the definition of a software application should make a clear distinction between individual components and their interaction in the overall software organisation.

Several authors [1, 2, 16, 27] have emphasized the importance of a framework for the formal definition of software architectures. Not only is it a prerequisite for a rigorous analysis of architectures, but it also increases their usefulness and reusability by removing the sources of ambiguity which are unavoidable in informal descriptions. Another major requirement for a software architecture model is its ability to express standard software design choices in a natural way. The common practice of software engineers is to represent architectures informally as 'box and line' drawings [2, 17]. Starting from this observation, we propose to define software architectures formally in terms of graphs, which constitute the mathematical model closest to the intuition conveyed by 'box and line' drawings. The nodes of the graph represent the individual entities which can themselves be described in conventional programming languages. The edges correspond to the communication links between entities. An architecture style is a class (or set) of architectures exhibiting a common shape. For example:

$$\begin{tikzcd} e_1 \arrow[r, "S"] & e_2 \arrow[r, "S"] & e_3 & & e_a \arrow[r, "S"] & e_b \end{tikzcd}$$

are two architectures of a style 'pipeline' (with e_i representing entity names and S directed links between entities). Technically, architecture styles are defined as context-free graph grammars.

The architecture can be seen as the skeleton of an application. In order to be executable, it must be 'fleshed', or completed with a mapping from nodes to entities defined in a given language. This complete description is called an architecture instance. The specification of the computation of an architecture instance mirrors its hierarchical organisation:

- The evolution of the local states of the entities follows the rules of the operational semantics of their programming language.
- A 'coordinator' is used to pilot the overall application. The coordinator is in charge of managing the architecture itself (creating and removing entities and links).

The coordinator is expressed in terms of conditional graph rewriting in the spirit of [10, 28]. The conditions bear on the public variables of the entities and represent the only possible interactions between a coordinator and the individual components (apart from the creation and destruction of links and entities).

The standard way to describe distributed systems is to resort to traditional sequential programming languages enhanced with facilities for process creation and communication (possibly through operating system procedure calls). On the other hand, specification languages like CSP [13],

SIGSOFT '96 CA, USA
© 1996 ACM 0-89791-797-9/96/0010...$3.50

and the π-calculus [23] are inherently parallel languages providing powerful and integrated constructions for process creation and synchronisation. None of these approaches makes it easy to extract the underlying communication topology from the application. As an illustration, [22, 25] propose sophisticated analyses to derive information about the topology of CSP and CML programs. We believe that a better basis for understanding the structure of a system is to consider its topology as an explicit feature rather than trying to dig it up from the program *a posteriori*. In other words, we propose that the 'skeleton' of the application is specified independently, with the 'flesh' described in a separate way.

Among the benefits of our approach, let us mention the following:

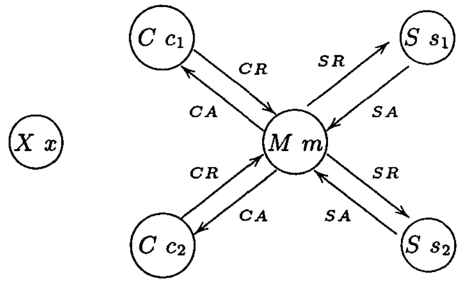
- It makes it possible to reconcile a dynamic view of the architecture, which is crucial for a large class of applications, with the possibility of static checking, ensuring that the evolution of the architecture conforms to its style. This verification can be seen as a form of static type checking of the coordinator (the type being the graph grammar defining the architecture style). In our framework, this amounts to a proof of convergence of graph rewrite rules.
- It provides a high-level view of software systems which is both intuitive and formally based. The clean separation between the computation of the individual entities and their coordination makes it easier to check global properties of the system. In particular, properties about the information flows in an application can be derived from the architecture style. This is of prime importance to be able to enforce the requirements imposed by a given security policy (confidentiality, integrity).

The presentation of the paper follows this two-level decomposition. In the next section, we introduce our view of architectures as graphs and architecture styles as graph grammars. Section 3 defines coordination as conditional graph rewriting and describes an algorithm for the static checking of a coordinator with respect to a given style. Sections 2 and 3 are independent of the definition of the atomic components of the architecture. In section 4, we complete the picture with a language for programming the individual entities. We provide a structural operational semantics of this language and we show how it cooperates with the semantics of coordination. In the conclusion, we relate our approach to previous work in this area and we suggest avenues for further research.

We use the 'client-server' model as a case study throughout the paper. Following the top-down presentation of the material, the client-server architecture style itself is introduced in section 2; a first version of the coordinator is presented in section 3 and the complete application is described in section 4 (Figure 1). A second example is presented in the appendix (a distributed hospital ward monitoring system inspired by [18] and [15]).

2 Architecture styles

Our notion of graphs is inspired by previous work on the chemical reaction model [3, 4, 10] and set-theoretic graph rewriting [28]. Formally, a graph is a multiset of relation tuples noted $R(e_1, \ldots, e_n)$ where R is a n-ary relation name and e_i are entity names (we assume appropriate countable sets of names). We consider only binary and unary relations here. In our context, a binary relation $L(e_1, e_2)$ represents a directed link of name L between e_1 and e_2. A unary relation $U(e)$ characterises the role of an entity e in the architecture. As an illustration, the following (unconnected) graph represents an example of a client-server architecture.



Unary relations are represented by circles and binary relations by arrows. C, S, M and X correspond respectively to client, server, manager and external entities. The external entity stands for the external world; it records requests for new clients wanting to be registered in the system. CR and CA correspond to client request links and client answer links respectively (SR and SA are the dual links for servers). The architecture represented by the above graph involves two clients c_1 and c_2, two servers s_1 and s_2, a manager m and x. It is formally defined as the multiset Δ:

$$\{X(x),\ M(m),\ C(c_1),\ C(c_2),\ S(s_1),\ S(s_2),$$
$$CR(c_1,m),\ CR(c_2,m), CA(m,c_1),\ CA(m,c_2),$$
$$SR(m,s_1),\ SR(m,s_2),\ SA(s_1,m),\ SA(s_2,m)\}$$

It should be clear that Δ is just one particular representative of a more general class of client-server architectures. Architectures belonging to this class must include values $X(x)$ and $M(m)$ and any number of servers and clients. Furthermore, they must follow the communication link pattern exhibited by Δ. We propose to specify such a class as a context-free graph grammar. Different notions of context-free graph grammars have been studied in the literature. They are defined either in terms of node replacement [9] or in terms of hyper-edge replacement [7]. Graph rewriting is also used in [8] as a model for distributed systems, but a dual approach is taken, with hyper-edges representing processes and nodes representing ports (also the process rewrite rules are essentially used to model synchronisation requirements). Our definition of graphs as multisets allows us to express hyper-edge replacement in a very natural way. A grammar is a four-tuple $[NT,\ T,\ PR,\ AX]$ where NT and T are sets of non-terminal and terminal symbols (each one with a given arity), PR is a finite set of production rules and AX is an axiom (the origin of the derivation). Terminal symbols correspond to the relations of the architecture. The production rules are pairs $(l,\ r)$ where l is a singleton $\{A(x_1, \ldots, x_i)\}$ (with A a non-terminal of arity i) and r is a multiset of terms $B(y_1, \ldots, y_j)$ with $B \in NT \cup T$.

Continuing our example, the client-server architecture style is defined as:

$$H_{CS} =$$
$$[\{CS, CS_1\},\ \{M,\ X,\ C,\ S,\ CR,\ CA,\ SR,\ SA\},\ R,\ CS]$$

with R the following set of rules (we use the concrete syntax $Left \Rightarrow Right$ to represent a pair $(\{Left\},\ \{Right\})$:

$$\begin{array}{lll} CS & \Rightarrow & CS_1(m) \\ CS_1(m) & \Rightarrow & CR(c,m),\ CA(m,c),\ C(c),\ CS_1(m) \\ CS_1(m) & \Rightarrow & SR(m,s),\ SA(s,m),\ S(s),\ CS_1(m) \\ CS_1(m) & \Rightarrow & M(m),\ X(x) \end{array}$$

Formally, a graph grammar $H = [NT, T, PR, AX]$ defines a rewrite system $\rightarrow_H$ between multisets:

$$M \rightarrow_H M' \Leftrightarrow M' = M - m_l + m_r$$

with $m_l \subseteq M$, $(N_e(m_r) - N_e(m_l)) \cap N_e(M) = \emptyset$, and $m_l = \sigma\ l$, $m_r = \sigma\ r$, with σ an injective substitution and $(l,\ r) \in PR$. The substitution σ maps variables to entity names. $N_e(M)$ is the set of entity names occurring in the multiset M. The second condition ensures that new variables occurring on the right-hand side of a rule are instantiated with entity names which are distinct from all other existing names. This constraint, which is usual in graph rewriting [28], is necessary to avoid unexpected variable sharing. It is crucial in our context to be able to state precisely the actual connections between entities.

The style defined by a grammar $H = [NT, T, PR, AX]$ is the set of all terminal graphs (graphs containing only terminal relation symbols) produced by $\rightarrow_H$ rewritings:

$$Class(H) = \{G \mid \{AX\} \overset{*}{\rightarrow}_H G \text{ and } G \text{ terminal}\}$$

For example, it is easy to check that the graph Δ defined above belongs to the client-server class: $\Delta \in Class(H_{CS})$.

3 Coordination

As mentioned in the introduction, it is often the case that the architecture of an application should be able to evolve dynamically. For instance, a client-server organisation must allow for the introduction of new clients or their departure, a pipeline may grow or shrink dynamically depending on the size of the data being processed, facilities for dealing with mobile computing may be required. In our framework, the evolution of the architecture is defined by a *coordinator*. The task of the coordinator is expressed by conditional graph rewrite rules in the style of [3, 4]. The semantics of the rules is similar to the definition used above for the rewrite system associated with graph grammars, except that we may have additional side conditions in coordination rules. These conditions bear on the local states of the individual entities; they do not play any role at this stage, so we do not consider them until the next section.

As an illustration, we introduce the following coordinator Coo_{CS} which applies to a client-server architecture:

$$X(x),\ M(m) \rightarrow X(x'),\ M(m),\ CR(c,m),\ CA(m,c),\ C(c)$$

$$CR(c,m),\ CA(m,c),\ C(c) \rightarrow \emptyset$$

The two rules describe respectively the introduction of a new client in the architecture and its departure. Note that these rules are completed with side conditions on the states of the entities in the complete version of the coordinator presented in the next section (Figure 1); otherwise, the coordinator could clearly lead to infinite behaviours.

The possibility of expressing architecture transformations is definitely a useful feature but it also raises a new question: is it possible to ensure that a coordinator does not break the constraints of a given architecture style? For example, had we forgotten, say $CR(c, m)$ in the right-hand side of the first rule, then the coordinator would have been able to transform a client-server architecture into an architecture which does not belong any longer to the client-server class defined by H_{CS}. What is needed is a static style checker which would be the counterpart for coordinators of the type checking algorithms of classical languages.

In order to define a checking algorithm for a given style $H = [NT,\ T,\ PR,\ AX]$, we first consider the graph rewrite system $\rightarrow_H^{-1}$ obtained by a right to left reading of the rules in PR (with the appropriate dual restrictions on variables appearing only on the right-hand side of a rule). Obviously, if G is a graph belonging to the style H, then $G \overset{*}{\rightarrow}_H^{-1} \{AX\}$. The coordinator Coo defines a second graph rewrite relation $\rightarrow_{Coo}$ and the checking algorithm must ensure that:

$$\forall\ G \text{ such that } G \overset{*}{\rightarrow}_H^{-1} \{AX\},$$

$$G \rightarrow_{Coo} G' \Rightarrow G' \overset{*}{\rightarrow}_H^{-1} \{AX\}$$

The checking algorithm proceeds in two stages.

1. First, all the Coo rules (l_i, r_i) are considered in turn. For each l_i, the set S_i of all its minimal contexts C_i^j with the associated non-terminal terms $N_i^j(x_1, \ldots, x_n)$ is computed. The pairs $(C_i^j,\ N_i^j(x_1, \ldots, x_n))$ satisfy the relation:

$$l_i + C_i^j \overset{*}{\rightarrow}_H^{-1} \{N_i^j(x_1, \ldots, x_n)\}.$$

   The minimal contexts C_i^j are the smallest multisets which have to be added to the l_i to reduce to a non-terminal (in other words, they are completely consumed by the reduction). They are computed by constructing all the possible superpositions (non empty intersections) of l_i with left-hand sides of $\rightarrow_H^{-1}$ rules and performing the corresponding reductions until a non-terminal is reached (or a term isomorphic to one of its ancestors). This iteration terminates because the reductions cannot increase the size of a term and the number of terms of a given size is finite (up to variable renaming). 'Impossible contexts' (contexts which cannot lead to a single non-terminal, and thus cannot lead to the axiom AX) are removed during the course of this process for a better precision of the analysis.

2. The second stage consists in applying $\rightarrow_H^{-1}$ rules, to show that all pairs (l_i, r_i) satisfy:

$$\forall (C_i^j,\ N_i^j(x_1, \ldots, x_n)) \in S_i,$$

$$r_i + C_i^j \overset{*}{\rightarrow}_H^{-1} \{N_i^j(x_1, \ldots, x_n)\}.$$

   The second stage terminates for the same reason as the first one. If the above property holds, then Coo is correct with respect to the style H.

The correctness of the algorithm is proven in [10]. As an illustration, let us apply it to the coordinator Coo_{CS} presented above. The first rule of Coo_{CS} is:

$$(\{X(x), M(m)\}, \{X(x'), M(m), CR(c,m), CA(m,c), C(c)\}).$$

There is only one superposition of $X(x)$, $M(m)$ with left-hand sides of $\rightarrow_{CS}^{-1}$, namely $X(x)$, $M(m)$ itself. The only possible rewriting to a non-terminal is

$$\{X(x),\ M(m)\} \rightarrow_{CS}^{-1} \{CS_1(m)\}.$$

Thus $S_1 = \{(\emptyset, CS_1(m))\}$ and we have to show that:

$$\{X(x'),\ M(m),\ CR(c,m),\ CA(m,c),\ C(c)\}$$

$$\overset{*}{\rightarrow}_{CS}^{-1} \{CS_1(m)\}$$

which is obtained in two rewrite steps.

The second rule of Coo_{CS} is treated in a similar way. The only successful reduction of its left-hand side is:

$$\{CR(c,m),\ CA(m,c),\ C(c),\ CS_1(m)\} \rightarrow_{CS}^{-1} \{CS_1(m)\}.$$

Thus $S_2 = \{(\{CS_1(m)\}, CS_1(m))\}$ and we obviously have

$$\{CS_1(m)\} \overset{*}{\rightarrow}_{CS}^{-1} \{CS_1(m)\}.$$

This concludes the verification that Coo_{CS} is a correct coordinator with respect to the client-server style CS.

4 Architecture instances

We have presented architecture styles and architecture transformations without any assumption on the individual entities so far. This section completes the picture by introducing a small language for entities. We provide its formal definition in terms of a structural operational semantics and we show how it interacts with the actions of the coordinator.

The syntax of the language of entities is better introduced through the complete version of the client-server application in Figure 1. First note that the relations defining the links of the architecture are typed (and so are the variables they bear on). The basic types are names of entities (*client, server, manager, external* here). For instance, the type attached to CR specifies a link from a *client* entity to a *manager* entity. Each entity defines public and private variables, output and input links and entity names. The public variables can be checked (but not assigned) by the coordinator. The public variable v of an entity a is denoted by $a.v$ in the definition of the coordinator. For example, the complete definition of Coo_{CS} creates a new client only if the boolean variable *newc* of the entity x is true. A new instance x' of the external entity is created in the same rule (which prevents the immediate re-application of the rule). Similarly, clients use a public variable *leave* to indicate their intention to leave the system. The output and input links of the entities must conform to the edges of the architecture (this can be checked statically).

The commands of the language are very much in the spirit of CSP except for the following generalisation: the semantics of input and output commands of the form

$$a \in L \ ?\ v \quad \text{and} \quad a \in L \ !\ E$$

correspond to the establishment of a rendez-vous with any entity a linked to the current entity through a link of name L. For example, the command $c \in CR\ ?\ r$ of the manager m is matched with the command $m \in CR\ !\ r$ in any of the clients c_i such that $CR(c_i, m)$ is an edge of the architecture. The effect of this communication is to assign c_i to $m.c$ in

Architecture style

$$\begin{array}{lcl} CS & \Rightarrow & CS_1(m) \\ CS_1(m) & \Rightarrow & CR(c,m),\ CA(m,c),\ C(c),\ CS_1(m) \\ CS_1(m) & \Rightarrow & SR(m,s),\ SA(s,m),\ S(s),\ CS_1(m) \\ CS_1(m) & \Rightarrow & M(m),\ X(x) \end{array}$$

Link types

C : *client*	S : *server*
CR : *client* $\times$ *manager*	CA : *manager* $\times$ *client*
SR : *manager* $\times$ *server*	SA : *server* $\times$ *manager*
M : *manager*	X : *external*

Coordinator

$$\begin{array}{lcl} Coo_{CS} = & & \\ \quad X(x),\ M(m),\ x.newc = true & \rightarrow & X(x'),\ M(m),\ CR(c,m), \\ & & CA(m,c),\ C(c) \\ \quad C(c),\ c.leave = true & \rightarrow & \emptyset \\ \quad CR(c,m),\ CA(m,c) & & \end{array}$$

Entities

$$\begin{array}{lll} client: & \textbf{pub} & leave : bool \\ & \textbf{priv} & r, a : int \\ & \textbf{out} & CR \\ & \textbf{in} & CA \\ & \textbf{ent} & m \\ & \textbf{body} & Init_c;\ leave := false; \\ & & *[Cond_1 \rightarrow C_1 \ \square \\ & & \quad Cond_2 \rightarrow m \in CR\ !\ r\ ;\ m : CA\ ?\ a\ ]; \\ & & leave := true \end{array}$$

$$\begin{array}{lll} server: & \textbf{priv} & r : int \\ & \textbf{out} & SA \\ & \textbf{in} & SR \\ & \textbf{ent} & m \\ & \textbf{body} & *[m \in SR\ ?\ r \ \rightarrow\ m : SA\ !\ f(r)\ ] \end{array}$$

$$\begin{array}{lll} manager: & \textbf{priv} & r, a : int \\ & \textbf{out} & SR, CA \\ & \textbf{in} & CR, SA \\ & \textbf{ent} & c, s \\ & \textbf{body} & *[c \in CR\ ?\ r \ \rightarrow s \in SR\ !\ r\ ; \\ & & \quad s : SA\ ?\ a\ ;\ \ c : CA\ !\ a] \end{array}$$

$$\begin{array}{lll} external: & \textbf{pub} & newc : bool \\ & \textbf{body} & Init_e;\ newc := false; \\ & & *[Cond_3 \ \rightarrow C_3]; \\ & & newc := true \end{array}$$

Figure 1: The complete client-server application

addition to the expected assignment of $c_i.r$ to $m.r$. This facility makes it possible for an entity to communicate with an unbounded number of other entities (without knowing their number or their existence), relying only on the topology of the architecture. Commands of the form

$$a : L \mathbin{?} v \quad \text{and} \quad a : L \mathbin{!} E$$

are closer to the standard CSP rendez-vous since the names of the partner entity is explicitly specified. This facility is necessary for an entity to realise a series of communications with the same partner: for instance, the manager must send the answer to the client which has issued the initial request.

The complete syntax of the commands of the language of entities is the following:

$$\begin{array}{lcl}
C & = & v := E \mid skip \mid C_1 \,;\, C_2 \mid Com \\
 & & [G \rightarrow C(\Box\, G \rightarrow C)^*] \mid \\
 & & *[G \rightarrow C(\Box\, G \rightarrow C)^*] \\
Com & = & H \mathbin{!} E \mid H \mathbin{?} v \\
H & = & a : L \mid a \in L \\
G & = & B \mid Com \mid (B,\ Com)
\end{array}$$

Symbols E and B denote respectively expressions (of any type) and boolean expressions, L is a link symbol declared in the **out** or **in** section, a is an entity variable and v any other (public or private) variable. As in CSP, a guard may be a combination of a boolean expression and a communication command. The semantics of the language is defined in the top part of Figure 2 as a labelled transition system on local configurations of the form $< C,\ S >$, with C a command and S a store. As usual, the label ϵ is used for silent transitions (transitions involving no communication). $Sem\ [E]\ S$ is the semantics of expression E in store S and $S[val/v]$ is the same as S except that variable v takes the value val. $\Re(Com, S, Com', S')$ is an intermediate relation associating the label Com' and the new store S' with the communication command Com and store S. $\aleph$ plays a similar role for guards. The label Com' and the new store S' are specified by the relation $\wp$, which formalises the above discussion on input and output commands. A repetitive command terminates when each guard includes a false boolean condition. Note that we do not follow the original CSP convention indicating the termination of the repetitive command when all processes addressed in the input/output guards have terminated [13]. This option would not make sense in our setting since, as explained above, a communication command may avoid to name the partner process explicitly and new processes and links can be added by the coordinator.

Let us now focus on the bottom part of Figure 2 which defines the semantics of the coordinator and show how it fits with the semantics of the underlying language of entities. Global configurations are triples $[Coo,\ G,\ Val]$ where Coo is the set of conditional rewrite rules defining the coordinator, G is the graph representing the architecture and Val is a function mapping entity names onto local configurations (pairs $< C,\ S >$). The three rules defining the semantics of coordination correspond to the following cases:

- The first rule simply propagates at the level of global configurations the silent transitions of local configurations.
- The second rule ensures proper matching of local transitions involving communications.

Semantics of the language of entities

$$< v := E\ ,\ S > \xrightarrow{\epsilon} < \emptyset\ ,\ S[(Sem[E]S)/v] >$$

$$< skip\ ,\ S > \xrightarrow{\epsilon} < \emptyset\ ,\ S >$$

$$\frac{< C_1\ ,\ S > \xrightarrow{\alpha} < \emptyset\ ,\ S' >}{< C_1\ ;\ C_2\ ,\ S > \xrightarrow{\alpha} < C_2\ ,\ S' >}$$

$$\frac{< C_1\ ,\ S > \xrightarrow{\alpha} < C_1'\ ,\ S' >}{< C_1\ ;\ C_2\ ,\ S > \xrightarrow{\alpha} < C_1' ; C_2\ ,\ S' >}$$

$$\frac{\Re(Com, S, Com', S')}{< Com, S > \xrightarrow{Com'} < \emptyset\ ,\ S' >}$$

$$\frac{\aleph(G_i, S, Com', S')}{< [\ldots \Box G_i \rightarrow C_i \Box \ldots]\ ,\ S > \xrightarrow{Com'} < C_i\ ,\ S' >}$$

$$\frac{\aleph(G_i, S, Com', S')}{< *[\ldots \Box G_i \rightarrow C_i \Box \ldots], S > \xrightarrow{Com'} < C_i ; *[\ldots \Box G_i \rightarrow C_i \Box \ldots], S' >}$$

$$\frac{\nexists\ i,\ Com',\ S'\ s.t.\ \aleph(G_i, S, Com', S')}{< *[\ldots \Box G_i \rightarrow C_i \Box \ldots]\ ,\ S > \xrightarrow{\epsilon} < \emptyset\ ,\ S >}$$

$$\frac{Sem\ [B]\ S\ =\ true}{\aleph(B, S, \epsilon, S)} \qquad \frac{\Re(Com, S, Com', S')}{\aleph(Com, S, Com', S')}$$

$$\frac{Sem\ [B]\ S\ =\ true \quad \Re(Com, S, Com', S')}{\aleph((B, Com), S, Com', S')}$$

$$\frac{\wp(H, S, H', S')}{\Re(H \mathbin{!} E\ ,\ S,\ H' \mathbin{!} (Sem\ [E]\ S)\ , S')}$$

$$\frac{\wp(H, S, H', S')}{\Re(H \mathbin{?} v\ ,\ S,\ H' \mathbin{?} val, S'[val/v])}$$

$$\wp(a \in L\ ,\ S,\ p : L\ ,\ S[p/a]) \qquad \wp(a : L\ ,\ S,\ S(a) : L\ ,\ S)$$

Semantics of coordination

$$\frac{Val(p) \xrightarrow{\epsilon} lc'}{[Coo,\ G,\ Val] \mapsto [Coo,\ G,\ Val[lc'/p]]}$$

$$\frac{L(p,q) \in G \quad Val(p) \xrightarrow{q:L!v} lcp' \quad Val(q) \xrightarrow{p:L?v} lcq'}{[Coo,\ G,\ Val] \mapsto [Coo,\ G,\ Val[lcp'/p][lcq'/q]]}$$

$$\frac{(l,r,c) \in Coo \quad \sigma\, l \subseteq G \quad V_\sigma(Val, c)\ =\ true}{[Coo,\ G,\ Val] \mapsto [Coo,\ G - \sigma l + \sigma r\ ,\ Val[< C_i, \bot > /p_i]]}$$

$$\text{for all } \ p_i \in N_e(\sigma r) - N_e(\sigma l) \ \text{ and } \ p_i \text{ of type } C_i$$

$$V_\sigma(Val, c)\ =\ Sem\ [c]\ [((Val(\sigma\ w)) \uparrow S)v/w.v]$$

Figure 2: Semantics of the language of entities and coordination

- The third case is the transformation of the architecture according to a rule (l, r, c) of the coordinator: l and r stand respectively for the left-hand side and the right-hand side of a rule and c is the condition. The value of c is evaluated with respect to the local states of the entities: in the definition of $V_\sigma(Val, c)$, σ is used to get the entity name associated with a variable w, *Val* returns the local configuration of the corresponding entity, and $\uparrow S$ extracts its store component. The names of $N_e(\sigma r) - N_e(\sigma l)$ correspond to new entities created by the rule: their original configuration is the pair $< C_i, \perp >$ where $\perp$ is the undefined store and C_i is the body of the entity which constitutes the type of the new variable (remember that the relations defining links and their variables are typed with entities). As an illustration, the type of C in the client-server application is *client* which means that the occurrence of a new variable c in the first rule of Coo_{CS} results in the creation of a new entity of type *client* and its initialisation with the undefined store.

An important observation concerning the process language described here is that it is very minimal indeed: it does not provide any facility for parallelism (no process creation, no parallel construct). This follows our original design choice of keeping a clear separation between the computation at the level of entities and the management of concurrency and communication at the level of the coordinator.

5 Conclusion

The need for specific languages and formal frameworks for describing the overall organisation of large software systems has triggered a considerable interest for software architectures and coordination languages during the last decade. Up-to-date surveys of formalisms and current trends can be found in [12, 29]. In order to relate our contribution to previous work in this area, let us focus on two complementary issues: the formal model used to describe software architectures and the features provided by specific software architecture or coordination languages.

- **Formal models:** Among the formal frameworks used to specify software architectures, let us mention the specification language Z [1], CSP [2], the chemical abstract machine [16], the π-calculus [21], partial ordered sets of events [20] and first-order logical theories [24]. These formalisms have been extensively studied and their respective advantages have been identified: Z is a widely used state-based specification language which allows for a clean decomposition of applications into collections of schemas (in the context of software architectures, schemas can be components, connectors, configurations [1]); CSP [13] and the π-calculus [23] are process algebra which highlight the concurrency and communication issues; the π-calculus includes powerful features for manipulating channels as first-class values which increases its potential for describing dynamic architectures; the chemical abstract machine [5] is based on the chemical reaction metaphor [4] which allows for a higher level of abstraction promoting parallelism as a basic computational model; the event-based structures of [20] are well suited to the explicit representation of timing properties; the logical theories used in [24] form the basis of a definition of a notion of architecture refinement.

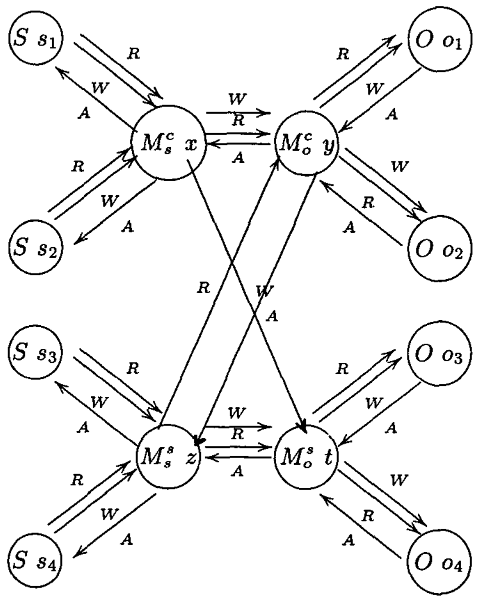
- **Software architecture languages:** there is a great variety of needs for software architectures [29] and this fact is reflected in the variety of papers published in this active area during the last few years. Aesop [11] provides facilities for the design and graphical visualisation of architectures following the rules prescribed by specific styles. Architecture styles are defined in a generic object model and include the specification of a vocabulary and constraints on the connections between elements. Unicon [30] supports a variety of components (such as 'shared data', 'filter', 'sequential file') and connectors (such as 'pipes', 'remote procedure calls') which have been implemented and used as a testbed for experimenting system construction mechanisms. Other proposals put more emphasis on the dynamic aspects of the system, introducing a separation between the sequential computation of individual agents and their coordination expressed in a specific languages [6, 14, 18]. In Linda [6], activities cooperate through a global tuple space using specific associative access primitives. The Conic environment provides a neat separation between individual tasks with explicit interfaces and a configuration level describing the overall application (which involves specifying the task components and establishing links between their ports). The Conic environment supports graphic tools for configuration programming and monitoring. Concoord [14] introduces a notion of coordinator which is in charge of a collection of processes. A coordinator has access to the state variables of its processes and can test them to trigger the creation (or deletion) of processes and the binding (and unbinding) of ports.

As far as formalisation is concerned, our approach to software architectures is in the spirit of previous proposals based on process calculi like CSP [2] or the π-calculus [23]. On the other hand, our computational model based on a clean separation between individual entities and a coordinator is inspired by [14] (but no formal model is provided for Concoord [14] and a number of technical choices differ from our own approach).

The main departure of our contribution with respect to the above process calculi based proposals is the emphasis put on the geometry of the architecture (following the 'box and line' drawing analogy), promoting it as an independent object. We believe that graph grammars provide a useful device for an intuitive and unambiguous understanding of the overall organisation of a system and form a suitable basis for various analyses. We observe, for example, that any powerful proof technique for parallel languages must include some form of analysis to obtain information about the communication topology of the program. The difficulty of this task is illustrated in various papers [22, 25]. Such analyses become much simpler in our framework because the topology is directly available from the specification of the architecture.

A significant benefit of our approach is that the direct information flows of a system are made explicit. This is because the coordinator cannot perform any assignment and the individual entities can only communicate through the links specified by the architecture. The latter property cannot be enforced by our model because it depends on the underlying language, but it can be ensured by a complementary *effect analysis*. The advantage is that such an analysis is easier to perform because all the parallelism issues are lifted at the level of the coordinator.

As an illustration of the relevance of our approach for the analysis of non-functional properties, let us consider a simplified version of the well-known 'Bell and LaPadula' security model [19]. The following graph represents in our setting a small platform with two levels of confidentiality (*Confidential* and *Secret* with *Confidential* < *Secret*).



$M_s^c(x)$ is a server for subjects of clearance *Confidential* and $M_s^s(z)$ is a server for subjects of clearance *Secret* (and similarly for $M_o^c(y)$ and $M_o^s(t)$ which are object servers). The links *W*, *R* and *A* represent respectively write requests, read requests and answers to read requests. The links specified by the architecture ensure that:

- No subject has read access to any object that has a classification greater than the clearance of the subject.
- No subject has write access to any object that has a classification less than the clearance of the subject.

Another important issue is the possibility of defining formally a notion of refinement between software architectures. We proposed in [10] a refinement relation which corresponds to class containment and which can be checked statically on the rules of the graph grammar. This notion of refinement was used in a different context (the transformation of parallel programs to optimise their implementation) and we are now exploring its applicability to software architectures. There does not seem to be a single answer to this problem because different usages may put different requirements on the notion of refinement. For instance, security-related properties may be preserved through refinements corresponding to multiset inclusion (because removing links or entities decrease the global information flow), but this form of refinement may not be acceptable for functional properties (because removing links or entities may alter the services provided by the system).

It should be clear that we have focused on specific aspects of software architectures in this paper and a number of important issues have not been considered. Our framework tackles the coordination problems, but it should be complemented with an appropriate interface to manage the data conversions required to support interoperability. One natural solution would be to rely on the interface definition language of a standard platform like CORBA. Also the types associated with the links in the architecture do not include a communication protocol dimension. We have assumed a rendez-vous mechanism here but a useful extension would be to associate links with user-defined communication protocols. One promising research direction is the notion of *regular processes* used in [26] to specify protocols for object behaviours.

Another issue deserving further work is the design of a user friendly interface supporting externally initiated changes to the architecture [18]. In our framework, these changes could be expressed, and formally controlled, through the 'external agents' as exemplified by the client-server case study.

Considering graph grammars themselves, a potentially interesting generalisation would be to consider context sensitive grammars. This would allow us to specify topologies like square grids which are out of reach of context free grammars. Further experience is necessary to assess the usefulness of this extension.

Acknowledgements

This work was partly supported by Esprit Basic Research project 9102 *Coordination*. Some of the inspiration for this paper emerged from previous work on Structured Gamma in collaboration with Pascal Fradet [10]. Jean-Pierre Banâtre, Alexandra Holzbacher and Valérie Issarny provided valuable feedback on an earlier version of this paper.

References

[1] G. Abowd, R. Allen and D. Garlan, *Using style to understand descriptions of software architecture*, Proc. Sigsoft'93: Foundations of Software Engineering, Software Engineering Notes, 18, 5, December 1993.

[2] R. Allen and D. Garlan, *Formalizing architectural connection*, Proc. 16th Int. Conf. Software Engineering, IEEE Computer Society, pp. 71-80, 1994.

[3] J.-P. Banâtre and D. Le Métayer, *Programming by multiset transformation*, Communications of the ACM, Vol. 36-1, pp. 98-111, January 1993.

[4] J.-P. Banâtre and D. Le Métayer, *Gamma and the chemical reaction model: ten years after*, Coordination programming: mechanisms, models and semantics, Imperial College Press, 1996, to appear.

[5] G. Berry and G. Boudol, *The chemical abstract machine*, Theoretical Computer Science, Vol. 96, pp. 217-248, 1992.

[6] N. Carriero and D. Gelernter, *Linda in context*, Communications of the ACM, Vol. 32-4, pp. 444-458, April 1989.

[7] B. Courcelle, *Graph rewriting: an algebraic and logic approach*, Handbook of Theoretical Computer Science, Chapter 5, J. van Leeuwen (ed.), Elsevier Science Publishers, 1990.

[8] P. Degano and U. Montanari, *A model for distributed systems based on graph rewritings*, Journal of the ACM, Vol. 34-2, pp. 411-449, April 1987.

[9] P. Della Vigna and C. Ghezzi, *Context-free graph grammars*, Information and Control, Vol. 37, pp. 207-233, 1978.

[10] P. Fradet and D. Le Métayer, *Structured Gamma*, Irisa Research Report PI-989, March 1996.

[11] D. Garlan, R. Allen and J. Ockerbloom, *Exploiting style in architectural design anvironment*, Proc. Sigsoft'94, Foundations of Software Engineering, pp. 175-188, 1994.

[12] D. Garlan and D. Perry, *Editor's Introduction*, IEEE Transactions on Software Engineering, Special Issue on Software Architectures, 1995.

[13] C. A. R. Hoare, *Communicating sequential processes*, Communications of the ACM, Vol. 21-8, pp. 666-677, August 1978.

[14] A. A. Holzbacher, *A software environment for concurrent coordinated programming*, Proc. First int. Conf. on Coordination Models, Languages and Applications, Springer Verlag, LNCS 1061, pp. 249-266, April 1996.

[15] A. A. Holzbacher, *Coordination of distributed and parallel programs in Concoord*, Coordination programming: mechanisms, models and semantics, Imperial College Press, 1996, to appear.

[16] P. Inverardi and A. Wolf, *Formal specification and analysis of software architectures using the chemical abstract machine model*, IEEE Transactions on Software Engineering, Vol. 21, No. 4, pp. 373-386, April 1995.

[17] R. Kazman, L. Bass, G. Abowd and M. Webb, *SAAM: A method for analysing the properties of software architectures*, Proc. 16th Int. Conf. Software Engineering, IEEE Computer Society, pp. 81-90, 1994.

[18] J. Kramer, *Configuration programming. A framework for the development of distributable systems*, Proc. COMPEURO'90, IEEE, pp. 374-384, 1990.

[19] C. E. Landwehr, *Formal models of computer security*, Computing Surveys, Vol. 13, No. 3, pp. 247-277, September 1981.

[20] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan and W. Mann, *Specification and analysis of system architecture using Rapide*, IEEE Transactions on Software Engineering, Vol. 21, No. 4, pp. 336-355, April 1995.

[21] J. Magee and J. Kramer, *Modelling distributed software architectures*, Proc. First int. workshop on Architectures for Software Systems, CMU Technical Report, CMU-CS-95-151, April 1995.

[22] N. Mercouroff, *An algorithm for analysing communicating processes*, 7th int. Conf. on Mathematical Foundations of Programming Semantics, pp. 312-325, March 1991.

[23] R. Milner, J. Parrow and D. Walker, *A calculus of mobile processes*, Journal of Information and Computation, Vol. 100, pp. 1-77, 1992.

[24] M. Moriconi, X. Qian and R. A. Riemenschneider, *Correct architecture refinement*, IEEE Transactions on Software Engineering, Vol. 21, No 4, pp. 356-372, April 1995.

[25] H. R. Nielson and F. Nielson, *Higher-order concurrent programs with finite communication topology*, Proc. 21st ACM Symp. on Principles of Programming Languages, pp. 84-97, January 1994.

[26] O. Nierstrasz, *Regular types for active objects*, Proc. OOPSLA'93, ACM Sigplan Notices, Vol. 28, No 10, pp. 1-15, October 1993.

[27] D. E. Perry and A. Wolf, *Foundations for the study of software architecture*, ACM Sigsoft, Software Engineering Notes, Vol. 17, No. 4, pp. 40-52, October 1992.

[28] J.-C. Raoult and F. Voisin. *Set-theoretic graph rewriting*, Proc. int. Workshop on Graph Transformations in Computer Science, Springer Verlag, LNCS 776, pp. 312-325, 1993.

[29] M. Shaw and D. Garlan, *Formulations and formalisms in software architecture*, Computer Science Today, Recent Trends and Developments, Springer Verlag, LNCS 1000, pp. 307-323, 1995.

[30] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young and G. Zelesnik, *Abstractions for software architecture and tools to support them*, IEEE Transactions on Software Engineering, Vol. 21, No. 4, pp. 314-335, April 1995.

Appendix: a hospital ward monitoring system

The distributed monitoring of a hospital ward system was used as a case study in [18] and [15]. The system consists of a number of nurses, beds and a secretary. Each nurse is associated with a collection of beds, some of them being occupied by a patient. Nurses check the state of their patients regularly and decide when they are allowed to leave the hospital. Patients can also send an alarm to their nurse when their state becomes critical. The following graph represents a possible configuration with two nurses n_1 and n_2. Nurse n_1 has two free beds f_1 and f_2 and one occupied bed b_1. Nurse n_2 has one occupied bed b_2 and one free bed f_3. Links C, R, L are used respectively for asking the state (e.g. temperature) of a patient, receiving his state and sending back the decision taken by the nurse (a boolean value indicating if the patient is allowed to leave the hospital). A is the alarm link and E is used to connect a nurse with her free beds.

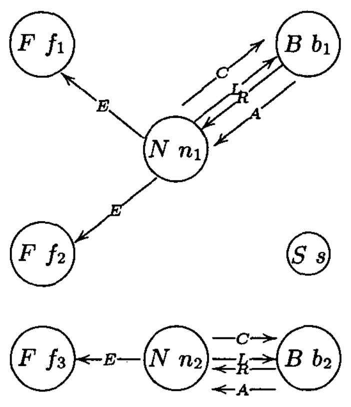


Figure 3 shows the architecture style of the application, the coordinator and the programs defining the individual entities. The secretary plays a role similar to the external entity in the client-server architecture. When its *newp* variable is set to *true*, the coordinator allocates a new patient to a nurse who has a bed available. Upon request, a nurse receives the temperature t of a patient and uses function $F(t)$ to decide whether the patient is allowed to leave the hospital. The patient receives the decision and stores it in the public variable *leave* which is used by the coordinator to modify the architecture according to this decision (replacing the occupied bed by a new instance of a free bed). Free beds are inactive entities and the associated program body is the *skip* command.

Architecture style

$$\begin{array}{lll}
H & \Rightarrow & H_1, S(s) \\
H_1 & \Rightarrow & N(n),\ H_2(n),\ H_3(n),\ H_1 \\
H_1 & \Rightarrow & \emptyset \\
H_2(n) & \Rightarrow & B(b),\ C(n,b),\ R(b,n),\ L(n,b),\ A(b,n),\ H_2(n) \\
H_2(n) & \Rightarrow & \emptyset \\
H_3(n) & \Rightarrow & F(f),\ E(n,f),\ H_3(n) \\
H_3(n) & \Rightarrow & \emptyset
\end{array}$$

Link types

N : *nurse*	C : *nurse* × *bed*	L : *nurse* × *bed*
B : *bed*	R : *bed* × *nurse*	A : *bed* × *nurse*
F : *free*	S : *secretary*	E : *nurse* × *free*

Coordinator

Coo_H =

$$\begin{array}{lcl}
N(n),\ S(s),\ s.newp = true & \rightarrow & N(n),\ S(s'),\ B(b), \\
 & & C(n,b),\ L(n,b) \\
F(f),\ E(n,f) & & R(b,n),\ A(b,n) \\
 & & \\
B(b),\ b.leave = true & \rightarrow & F(f),\ E(n,f) \\
C(n,b),\ L(n,b) & & \\
R(b,n),\ A(b,n) & &
\end{array}$$

Entities

```
nurse :  priv   $t : int$
         out    $C,\ L$
         in     $A,\ R$
         port   $b$
         body   $Init_n$;
                $*[b \in A\ ?\ t \rightarrow Action(b,t)\ \Box$
                  $b \in C\ !\ true \rightarrow b : R\ ?\ t\ ;\ b : L\ !\ F(t)]$

bed :  pub    $leave : bool$
       priv   $y, t : int$
       out    $A,\ R$
       in     $C,\ L$
       port   $n$
       body   $Init_b$; $leave := false$;
              $*[(\neg leave \wedge Cond_1) \rightarrow C_1\ \Box$
                $(\neg leave \wedge Cond_2) \rightarrow C_1\ \ n \in A\ !\ t\ \ \Box$
                $(\neg leave,\ n \in C\ ?\ y) \rightarrow n : R\ !\ t\ ;\ n : L\ ?\ leave\ ]$

secretary :  pub    $newp : bool$
             body   $Init_s$; $newp := false$;
                    $*[Cond_3 \rightarrow C_3]$;
                    $newp := true$

free :  body   $skip$
```

Figure 3: The hospital ward application