Modeling Software Architectures and Styles with Graph Grammars and Constraint Solving

Dan Hirsch, Paola Inverardi and Ugo Montanari

Departamento de Computaión, Universidad de Buenos Aires, Ciudad Universitaria, Pab.I, (1428), Buenos Aires, Argentina, dhirsch@dc.uba.ar

Dip. Di Mat. Pura ed Applicata, Universita' dell'Aquila, Via Vetoio, Localita' Coppito, L'Aquila, Italia, inverard@univaq.it

Dipartimento di Informatica, Universita' di Pisa, Corso Italia 40, (56125), Pisa, Italia, ugo@di.unipi.it

Key words: Architectural descriptions, graph rewriting, styles, dynamic architectures, reconfiguration

Abstract: A software architecture style is a class of architectures exhibiting a common pattern. The description of a style must include the structure model of the components and their interactions (i.e. structural topology), the laws governing the dynamic changes in the architecture, and the communication pattern. A simple and natural way to describe a system is by using graphs, and as an extension of this, by using grammars to describe styles. So a grammar will generate all possible instances of that style. In our work we represent a system as a graph where edges (or hyperedges) are components and nodes are ports of communication. The construction and dynamic evolution of the style will be represented as context-free productions and graph rewriting. To model the evolution of the system we need to choose a way of selecting which components will evolve and communicate. For this we propose to use techniques of constraint solving already applied in the representation of distributed systems. From this approach we obtain: an intuitive way to model systems; a unique language to describe the style, but still a clear separation between coordination and configuration, and with these a direct way of following the evolution of the system and prove properties.

1. INTRODUCTION

A software architecture style is a class of architectures exhibiting a common pattern (Shaw, M. and Garlan, D., 1996). The description of a style must include the structure model of the components and their interactions (i.e. structural topology), the laws governing the dynamic changes in the architecture, and the communication pattern. In the following we refer to all these aspects as a *complete style* description. A simple and natural way to describe a system architecture is by using graphs, and as an extension of this, by using grammars to describe styles. So a grammar will generate all possible instances of that style. This approach has first been proposed in (Le M'etayer, D., 1998).

In our work we represent a system as a graph where edges (or hyperedges) (Drewes, F. et al., 1996) are components and nodes are ports of communication. The construction and dynamic evolution of the style will be represented as *context-free* productions and graph rewriting. The productions that represent the style will be grouped in three sets. The first one contains the productions that correspond to the construction of the initial static configuration of the system. The second set contains the rules that model dynamic changes in the configuration of the system (create and remove components) and the third set contains the rules that model the communication pattern.

To model the evolution of the system we need to choose a way of selecting which components will evolve and communicate. For this we propose a technique already applied in (Montanari, U. and Rossi, F., 1997) and (Montanari, U. and Rossi, F., 1996) to represent distributed systems with graph rewriting and constraint solving. A graph represents a distributed system, where edges represent processes and nodes represent shared data. In order to evolve, one process may need to synchronize with adjacent processes on some conditions on the shared data. If they agree on these conditions, then all of them can evolve. This is modeled by a two phased approach where, context-free process productions are specified (a set for each process) with synchronization conditions for each of the possible moves. After that, context-sensitive subsystem rewriting rules are obtained by combining some context-free productions (this is called *the rule-matching problem*) (Corradini, A. et al., 1985).

Applying one of these context-sensitive rules, allows for the evolution of a subpart of the system consisting of several processes (each with one of its

context-free productions) that agree on the conditions imposed on the shared data.

Applying the rule means making all such processes (and not a proper subset of them) evolve, each with one of its context-free productions.

The solution of the rule-matching problem is implemented considering it as a finite domain constraint problem (Mackworth, A., 1988). In this paper we will not describe these techniques the interested reader can refer to the references.

In the case of software architectures we use constraint rules to coordinate the dynamic evolution of the system. This is done by using constraints on ports to represent communications between components and (if necessary) to control changes in the configuration of the system. One difference with (Montanari, U. and Rossi, F., 1997), is that in our approach, we will rely on two basic types of communication paradigms: point-to-point and broadcast communications. These will be represented with two types of nodes. With point-to-point communication the rule-matching problem is easier, it has to choose only two rules (for each sender, one receiver). In the case of broadcast the solution is the same as in (Montanari, U. and Rossi, F., 1997). This allows to represent both types of communication at the same time.

The use of hyperedge rewriting grammars and constraints to represent styles and model evolution gives us an intuitive way to model systems and a unique language to capture the style, but still with a clear separation between coordination and configuration. Besides we have a direct way of following the evolution of the system and prove properties and the inheritance of the distributed solutions for the rule-matching problem. Moreover, context-free hyperedge rewriting is natural for modeling the behavior of components independently of each other, and its generality can be used (if one wants to) to incorporate descriptions of more complex connector elements in the specification of a system (you just represent connectors with edges and productions for their evolution).

A related work that uses graph grammars is (Le M'etayer, D., 1998). There, a dual approach is taken and architectural styles are represented as context-free graph grammars were nodes represent components and edges their communication links. But, in this case the grammar only specifies the static configuration of the system (referred as style). The dynamic evolution (create and remove components) is defined independently by a *coordinator*, and the rules of the coordinator are checked to preserve the constraints imposed by the grammar that defined the style. Also a CSP-like language for the individual entities is given to fit with the coordinator semantics.

The main difference between the two approaches is that in our work we give a uniform description of the *complete style* with grammars (but still

maintaining an independent description of components behavior). Also, we don't have a global coordinator of evolution, but instead, each component defines its own evolution (Magee, J. and Kramer, J., 1996a).

In (Le M'etayer, D., 1998), communication links are represented as edges, and components as nodes. We chose a dual approach, because we want the evolution of the style (including the communication pattern) to be modeled with the rewriting steps of the graphs. So, in this way hyperedges (and their associated nodes) are used only to represent components and the ports that they will share and use to communicate among them. A graph with this representation gives a simple view of the structure of an instance of architecture at a given state, separated from the application of the rewriting rules that shows the evolution between states. In this way, a clearer representation of the system is obtained while a separation of configuration and evolution is achieved, which is a desirable property of software architecture description languages (Medvidovic, N., 1997).

Another important point is that the evolution and communication pattern can be followed directly by the rewriting sequences on the graphs, analogously to what happens in the CHAM description of software architectures (Inverardi, P. and Wolf, A., 1995). And this allows the verification of properties of the architecture, like for example deadlock (Degano, P. and Montanari, U., 1987; Compare, D. et al., n.d.).

In section 2 basic notions on graph rewriting and constraint rules are introduced, then in section 3 we apply these notions to software architectures using some examples, and finally in section 4 we draw the conclusions and describe our future work.

2. BACKGROUND

In this section we introduce basic notions on hypergraphs, hypergraph rewriting and constraint productions.

2.1 Graphs and Graph Rewriting

DEFINITION [HYPERGRAPHS]

We define an edge-labeled hypergraph, or simply a graph as a tuple $G = < N, E, c, ext, lab_{LN}, lab_{LE} >$, where:

1. N is a set of nodes.
2. E is a set of edges.
3. $c: E \rightarrow N^*$ is the connection function (each edge can be connected to a list of nodes).
4. $ext \in N^*$ is a set of external nodes.

5. lab_{LE}: $E \rightarrow LE$ is the labeling function of edges.
6. lab_{LN}: $N \rightarrow LN$ is the labeling function of nodes.

A *graph production* rewrites a graph into another graph, deleting some elements (nodes and edges), generating new ones, and preserving others. In this paper we will just consider context-free productions, which rewrite a graph containing a single hyperedge, into an arbitrary graph, while preserving the (*external*) nodes connected by the rewritten hyperedge. Therefore, in a context-free production, no nodes are deleted.

DEFINITION [GRAPH PRODUCTIONS]

Given a set of external nodes *EN*, a graph production *p* is a pair $< L , R >$, where:

1. *L* is a graph containing only an hyperedge.
2. *R* is a graph.
3. The external nodes of *L* and *R* are exactly those in *EN*.

Context-free graph productions will be written as $L \rightarrow R$, where *L* is the (graph containing only the) hyperedge to be rewritten and *R* is the graph to be generated. A production $p = (L \rightarrow R)$ can be applied to a graph *G* yielding H ($G \Rightarrow_p H$) if there is an *occurrence* of *L* in *G*. The result of applying *p* to *G* is a graph *H* which is obtained from *G* by removing the occurrence of *L* and adding *R*.

DEFINITION [GRAPH REWRITING SYSTEM]

A graph rewriting system is a pair $GRS = < G_0 , P >$, where:

1. G_0 is a graph.
2. *P* is a set of graph productions.

A derivation for *GRS* is a finite sequence of direct derivation steps of the form $G_0 \Rightarrow_{p1} G_1 \Rightarrow_{p2} \ldots \Rightarrow_{pn} G_n = H$, where $p_1, \ldots, p_n$ are in *P*.

To model coordinated rewriting, it is necessary to add some labels to the nodes in the left member of productions. Assuming to have an alphabet of requirements A, then we need a partial function f: $nodes(L) \rightarrow A$ which associates conditions (or actions) to some of the nodes. In this way, each rewrite of an edge must match conditions with its adjacent edges and they have to move as well. For example, consider two edges which share one node, such that no other edge is attached to that node, and let us take one production for each of these edges. Each of these productions have a condition on that node (a and b). If $a \neq b$, then the edges cannot rewrite together (using that rules). If $a = b$, then they can move, via the context-

sensitive rule obtained from merging the two context-free rules (*rule-matching problem*).

3. GRAPH REWRITING FOR SOFTWARE ARCHITECTURE STYLES

Now we will apply the notions introduced in the previous section to the description of software architectures. Software architectures are represented as hyperedge graphs where edges are components and nodes are communication ports. Two edges sharing a node means that there is a communication link between the two components. As we mentioned in the introduction we have two types of nodes, point-to-point and broadcast communications.

A software architecture style is described by a hyperedge context-free grammar. The productions of a grammar are grouped in three sets. The first set represents the construction of all possible initial configurations of the class of architectures modeled by the style. The second set represents the rules for the dynamic evolution of the configuration, this means create and remove components. The third set contains the rules that model the communication pattern of the architecture. This set contains productions to model the communication evolution for each type of component. These rules are constrained productions that during rewriting will coordinate for the evolution of the system. Also, some of the rules in the second set can be (if necessary) constrained. This can be used to model coordinated changes in the configuration. We will show this in the second example.

Edge labels have two parts. One is the component name and the other is the status of the component that represents its different states during evolution. Edges are drawn as boxes, broadcast ports as full circles and point-to-point ports as empty circles. Nodes are labeled with port names (port names are local to rules and external nodes have to be matched when a production is applied). Constraints decorate nodes in bold letters, and appear on the right part of a production. For point-to-point we have a CCS like notation for the constraints, where a node labeled as $\bar{a}$ means that the component is the sender of a message a and a node labeled a is its receiver. For broadcast, all nodes that have to coordinate are labeled with the constraint representing the message.

Now we present three simple examples to show how a style is modeled.

3.1 Client-Server

The first example is a client-server case study based on the one used in (Le M'etayer, D., 1998). We have clients, servers and a manager. An instance of the style can have an initial configuration with any number of clients, any number of servers and one manager. Clients and servers communicate through the manager. Clients and manager are connected via the *CR* (client request) and *CA* (client answer) ports. Servers and manager are connected via the *SR* (server request) and *SA* (server answer) ports. In this example all nodes are point-to-point ports.

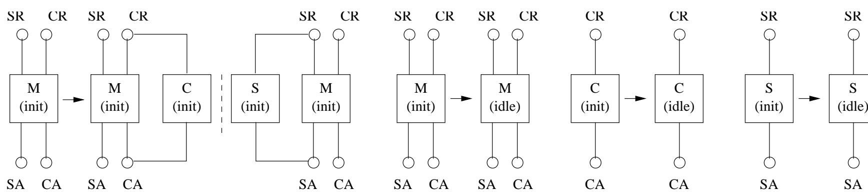

Figure 1. Client-Server: Static Productions

As we said at the beginning of this section we grouped productions in three sets. The first set represents the construction of all possible initial configurations of the class of architectures modeled by the style. For the client-server example these are the productions in figure 1. This figure shows that all instances start with the manager and then clients and servers are attached to it. This is done by the application to the manager of the first and second rules in figure 1 (the dash line is a shortcut to describe two productions for the manager). Note that the status of all components at this level is *(init)*, indicating that they are in a construction (or initialization) phase. Figure 2 shows an instance with two clients and one server generated by these productions.

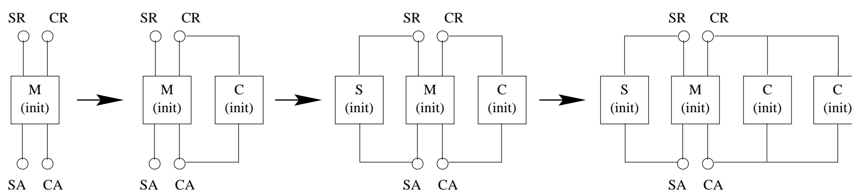

Figure 2. Client-Server: An instance of the architecture style generated by the static productions

After the desired initial configuration is obtained, then (init) → (idle) rules are applied (last three in figure 1). These rules mean that the construction phase is over and that the system is ready to start to work. Now, you can apply the last two sets of rules for the evolution of the architecture.

Figure 3 shows the dynamic rules. In this example we have two simple rules. The first one states that the manager accepts the incorporation of a new client in the system, and the second one is for clients that want to leave the system.

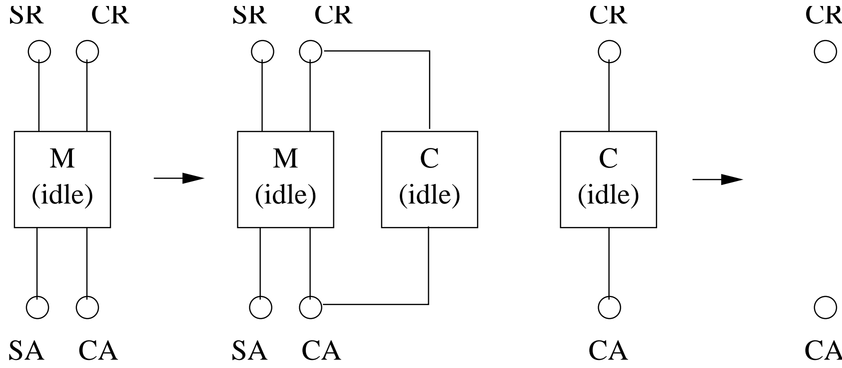

Figure 3. Client-Server: Dynamic Productions

Figure 4*a* shows the rules corresponding to the communication pattern. Note that all component specifications are independent from each other and that the only relation between them is by the communication coordination. This is important for a better understanding and analysis of the system behavior.

In this example all ports are point-to-point so, the manager will have to choose among the clients that want to make a request (obviously this is handled by the constraint resolution algorithms). In a broadcast communication all rules that want to rewrite and share nodes have to agree on the conditions imposed by the constraints.

In figure 4*b* you can see how the constrained rules work with a client that sends a request, the manager, and a server that returns the answer. These components can be part of a bigger graph but we assume that they were already chosen by the constraint solving algorithm at each rewriting step. The three components start from an *idle* state. Then the manager and the client rewrite respectively to the *pcr* and *wa* states after having coordinated on the client request. The second rewriting is between the manager and the server (to *wsa* and *pr* states, respectively) when the manager forwards the request the server. The last two steps are from the server to the manager (to *idle* and *psa* states, respectively) delivering the answer, and from the

manager to the client returning the answer of its request. At the end of the sequence they return to an *idle* state (the server already after returning the answer), where new communications can be performed or any of the dynamic productions can be applied. Note that the dynamic productions in figure 3 can be applied only when components are in an *idle* status (they cannot be in the middle of a communication).

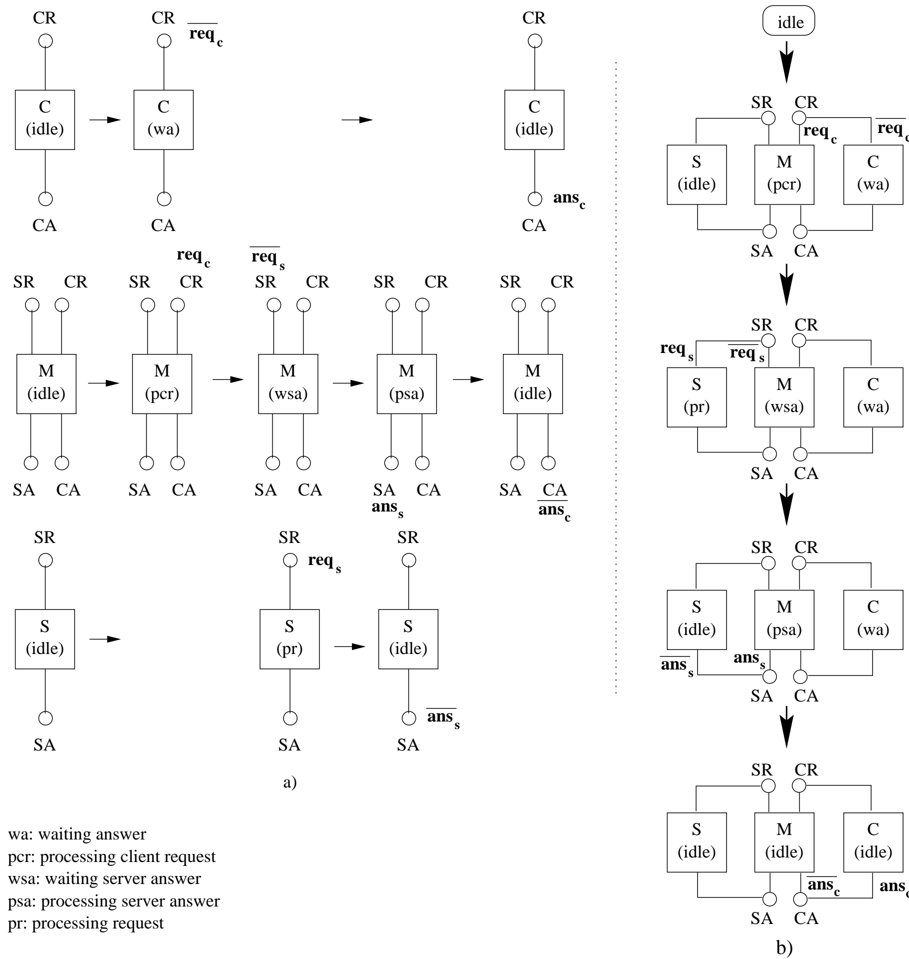

Figure 4. Client-Server: Communication Pattern Productions

It is worthwhile mentioning that we choose the level of abstraction for the description of the communication pattern. For example, figure 5*a* is an alternative set of rules for the communication pattern, where there are two

rewrites instead of four, one that sends the request from the client to the server (via the manager), and the other that returns the answer to the client (figure 5*b*).

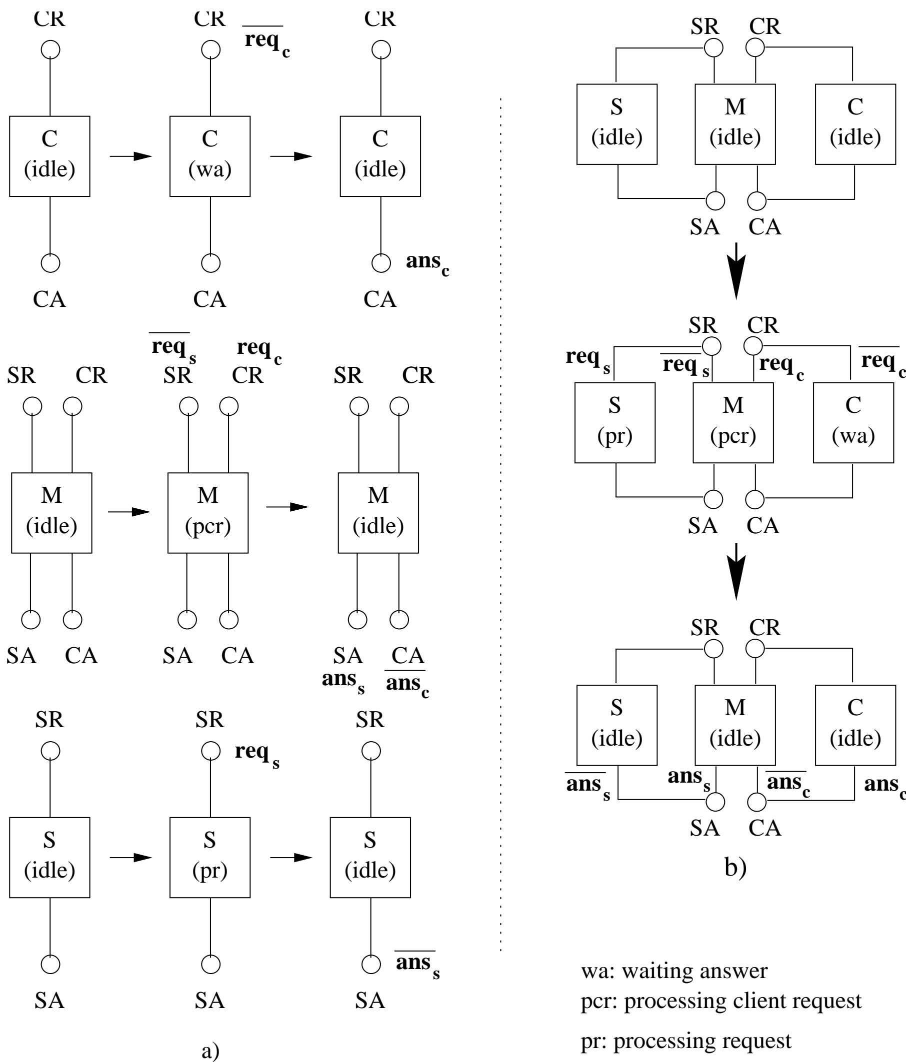

Figure 5. Client-Server: Communication Pattern Productions. An alternative

With this grammar we obtained a complete characterization of the style in a unique language and a clear identification of the steps that every architecture instance gives during its evolution. Also note that by analyzing the derivation tree it is possible to have all the computations of the system

allowing the verification of properties of the architecture, like for example, deadlock (Degano, P. and Montanari, U., 1987).

3.2 Remote Medical Care System

This example is a simplification of a case study presented in (Balsamo, S. et al., 1998) for performance evaluation of a software architecture. We present here only a partial specification of the style, to show how constraints can be used to control an ordered evolution in the configuration of the system.

This system is part of a project carried out by University of L'Aquila at Parco Scientifico e Tecnologico d'Abruzzo, a regional consortium of public and private research institutions and manufacturing industries.

The Teleservices and Remote Medical Care System (TRMCS) provides and guarantees assistance services to users with specific needs, like disabled or elderly people. It is composed of a set of *Users*, which are connected to a *Router* which interacts with a *Server*. An external component, the *Timer* allows the modeling of time.

The four types of units operate as follows:

- **User** sends either alarm (i.e. help requests) or check signals (i.e. control messages about the subsystem user state or the user's health state, respectively).

- **Router** accepts signals (control or alarm) from the users. It forwards the alarm requests to the Server and checks the behavior of the subsystem user though the control messages.

- **Server** dispatches the help requests.

- **Timer** sends a clock signal for each time unit.

There is only one server in the system. A variable number of routers are connected to the server and a variable number of users are connected to each router. The timer controls all routers. Figure 6*a* shows the static productions and figure 6*b* shows an instance of the system with two routers, and one user attached to the first one and two others to the second router.

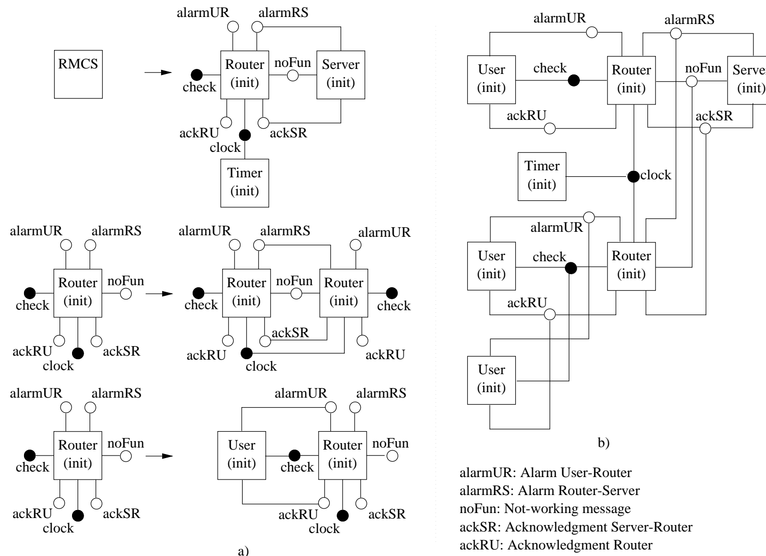

Figure 6. TRMCS: Static Productions

In the Client-Server example presented above, clients can leave the system independently of the other components. The only restriction, as it is modeled in the productions, is that they cannot leave the system if they are in the middle of a communication. In the TRMCS system, users have a similar behavior to the clients, but for routers the situation is different. In the case of a router, it is allowed to leave the system, but it cannot disappear without checking if there are users still connected to it. One possible action for the router if there are users connected to it, is to wait until all of them leave and then, when there are no users connected, it can leave too. These actions are described with the productions in figure 7.

These productions are part of the set of dynamic productions for the TRMCS. The first rule is for the user and it allows it to leave the system independently (i.e. it is not constrained). The second rule is for the router and it is constrained. The condition **noUSER** is imposed on the *check* port. A router and its users are connected to this port. This is a broadcast type of port, so a condition in it means that for this rule to be applied it must coordinate with all other edges connected to that port (i. e. the users). So, if all neighbors agree on the condition, then everybody can rewrite. But in this case, the only one with this condition is the router and it cannot leave the system while users are attached to it. When all users connected to the router leave the system then the production with condition **noUSER** is satisfied and then it can be applied to the router. The rule can be applied because there are no neighbors, so the router is the only one that has to agree on the constraint.

We can mention, that after the router leaves the system, the three isolated nodes that remain can be eliminated with a special action called *end* that function as a type of garbage collection.

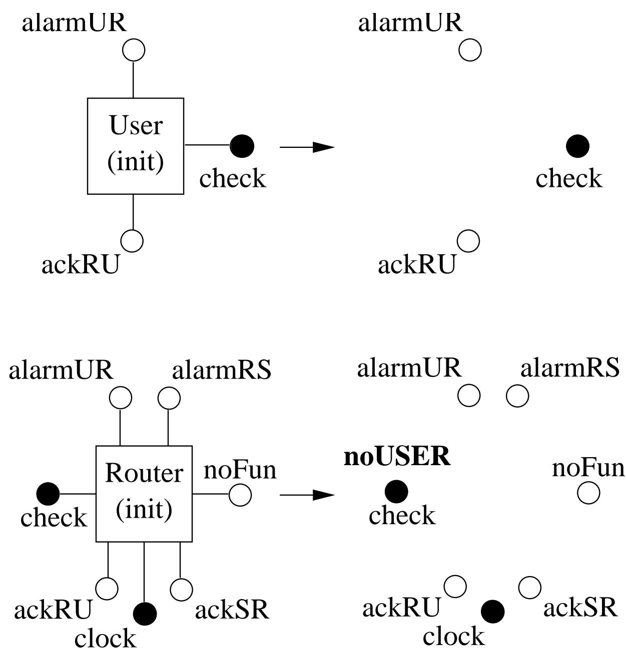

Figure 7. TRMCS: Dynamic Productions

This is an example of coordinated evolution, where constraints are used to control and coordinate the dynamics of a system.

3.3 Connectors: Parallel Point-to-point

Software architectures may require complex interactions among components. Usually, connectors may be defined as architectural building blocks to help model and specify these interactions. The modeling of connectors explicitly and independently, helps to achieve a higher level of reusability allowing to use already specified connectors in different styles and to create new connector types as the composition of basic ones.

So, in this direction we propose to use the generality of the model we are presenting to obtain independent connector descriptions. Using the same language to specify connectors based on more basic ones, allows to

incorporate them to the primitive set of communication types and reuse them successively in different style descriptions.

In all the examples presented we use two basic types of communication: broadcast and point-to-point communication. In the next example we use the broadcast port as a basic type and specify with constrained productions the parallel point-to-point communication. The specification of a parallel point-to-point port allows for a set of adjacent components to perform parallel communications between pairs. This means that in a given port (the one we are specifying), for each sender a receiver (if there are available) is selected to accept the communication and if there are more than one pair willing to communicate, simultaneous interactions are allowed.

Figure 8 shows the specification for a connector (edge *C*) from two to four components. In this figure all ports are broadcast ports. For two components broadcast and point-to-point are the same (figure 8*a*). For three components, figure 8*b* shows the three possible alternatives (with three components there are no parallel communications) and figure 8*c* shows the nine possible interactions that can take place between four components. In a similar way this specification can be generalized for *n* components. In this case, point-to-point communication is associative and commutative so once we have the connector specification we can abstract from it and use the new connector as a new type of port. Also, we can mention that repeatedly composing the connector specification for three components and the corresponding one for four components (only considering the rules for a single pair communication) in a sequential pattern, we obtain the simple point-to-point communication.

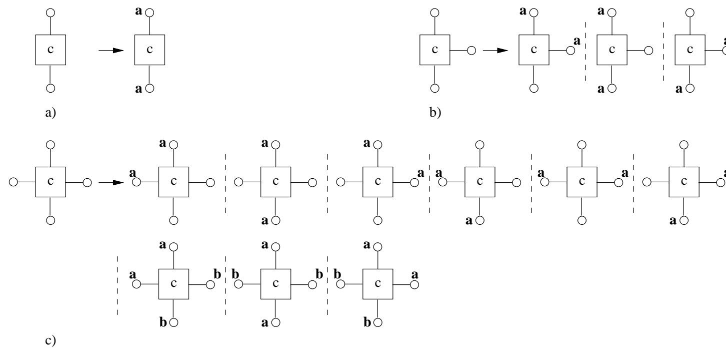

Figure 8. Parallel Point-to-point connector

In this way, an independent specification of a new connector is obtained and it can be reused in the description of other software architecture styles.

4. CONCLUSIONS AND FUTURE WORK

In this work we have presented a specification method for software architecture styles using hyperedge context-free graph grammars. Based on the rewriting system specified by the grammars we describe the style as a set of productions that model the initial structural topology of the architecture, the laws governing the dynamic changes, and its communication pattern.

Among the benefits of this approach we can mention:

A simple description of systems with a unique language is obtained; the use of constraints to model coordination of components allows a clear description of component interactions and controlled dynamics and the inheritance of the distributed solutions for the rule-matching problem. As we said, we propose to use a technique already applied in (Montanari, U. and Rossi, F., 1997) and (Montanari, U. and Rossi, F., 1996) to represent distributed systems with graph rewriting and constraint solving. This is modeled by a two phased approach where, context-free process productions are specified (a set for each process) with synchronization requests for each of the possible moves. After that, context-sensitive subsystem rewriting rules are obtained by combining some context-free productions.

The solution of the rule-matching problem is implemented considering it as a finite domain constraint problem (Mackworth, A., 1988), where variables are associated to processes and constraints to ports. The domain of a variable is then the set of all context-free productions for the corresponding process, and each constraint is satisfied by the tuples of context-free productions (one for each adjacent process) whose synchronization requirements agree on the considered port. In this kind of constraint problem, a solution is thus a choice of a context-free production for each process, such that all synchronization requirements are satisfied. Usually, finite domain constraint problems are solved by a backtracking search over a tree of the possible alternatives for each variable. To deal with this type of problems many efficient techniques have been proposed (constraint propagation or local consistency algorithms) (Mackworth, A., 1988), (Dechter, R. and Pearl, J., 1988). As in (Montanari, U. and Rossi, F., 1997), this kind of graph rewriting can be lifted to a general framework, called *the tile model* (Gadducci, F. and Montanari, U., 1996), which permits a clear separation between sequential rewriting and synchronization.

Also, context-free rules are a natural way for modeling the behavior of components independently of each other allowing a distributed implementation, and as we saw in the client-server example, constrained rules allows different levels of detail for the description of transactions (Bruni, R. and Montanari, U., 1997). This is a convenient property to model

architectures in which components are required to configure themselves (Magee, J. and Kramer, J., 1996a).

In this paper we model ports just as connections between components but as was shown in the examples the generality of the method can be used to incorporate descriptions of more complex connector elements in the specification of a system. If it is necessary complex connectors can be incorporated as a new type of edge.

Another thing to mention, is that in the examples presented we did not include termination rules. Constraints and productions can be used to model local and coordinated termination and this will be important for the verification of properties on the derivation tree.

We agree that the use of context-free rules limit the type of architecture styles that can be described, but we consider this as a first step on our work. With this type of rules, two separate edges already created cannot be bound later, so for example, an architecture instance that has a pipeline style cannot be converted, after its creation, into a ring. This is a great restriction that can easily be modeled in languages like π-calculus. But work like (Montanari, U. and Pistore, M., 1995), shows that this type of calculus can be represented with graph rewriting (not context-free).

Finally, the productions that we use are all rewriting rules (one thing is replaced by another), but an interesting extension is to incorporate refinement rules where the history of the system is remembered. It is worth mention that in the original paper (Degano, P. and Montanari, U., 1987) the partial ordering is generated with the past history of the derivation. This can be useful in the description of a bigger class of software architectures, specially those in which the organization of components and connectors may change during system execution (Magee, J. and Kramer, J., 1996b).

In spite of the fact that context-free productions limit the classes of systems that can be described, it is clear that the description language proposed has very good properties for modeling reconfiguration and self organising architectures. It is our intention to continue the research in this direction for a deeper analysis of the subject.

ACKNOWLEDGMENTS

The third author was partially supported by CNR Integrated Project Sistemi Eterogenei Connessi mediante Reti di Comunicazione, Esprit Working Group COOR-DINA and Italian Ministry of Research Tecniche Formali per Sistemi Software. The first author was partially supported by ARTE Project, PIC 11-00000-01856, ANPCyT and FOMECProject 376, Contract 164.

REFERENCES

Balsamo, S., Inverardi, P., Mangano, C. and Russo, F. (1998). Performance evaluation of a software architecture: A case study, *Proceedings of the Ninth International Workshop on Software Specification and Design*.

Bruni, R. and Montanari, U. (1997). Zero-safe nets, or transaction synchronization made simple, *EXPRESS'97, Electronic Notes in Theoretical Computer Science* **7**.

Compare, D., Inverardi, P. and Wolf, A. (n.d.). Uncovering architectural mismatch in dynamic behavior. To appear.

Corradini, A., Degano, P. and Montanari, U. (1985). Specifying highly concurrent data structure manipulation, in Bucci, G. and Valle, G. (eds), *COMPUTING 85:A Broad Perspective of Concurrent Developments*, Elsevier Science.

Dechter, R. and Pearl, J. (1988). Network-based heuristics for constraint satisfaction problems, in Kanal and Kumar (eds), *Search in Artificial Intelligence*, Springer Verlag.

Degano, P. and Montanari, U. (1987). A model for distributed systems based on graph rewriting, *Journal of the Association for Computing Machinery* **34**(2).

Drewes, F., Kreowski, H.-J. and Habel, A. (1996). Foundations, in G. Rozenberg (ed.), *Handbook of Graph Grammars and Computing by Graph Transformation*, Vol. I, World Scientific, chapter 2.

Gadducci, F. and Montanari, U. (1996). The tile model, *Technical Report TR-96-27*, Department of Computer Science, University of Pisa.

Inverardi, P. and Wolf, A. (1995). Formal specification and analysis of software architectures using the chemical abstract machine model, *IEEE Transactions on Software Engineering* **21**(4): 373-386. Special Issue on Software Architectures.

Le M'etayer, D. (1998). Describing software architecture styles using graph grammars, *IEEE Transactions on Software Engineering* . to appear.

Mackworth, A. (1988). *Encyclopedia of IA*, Springer Verlag, chapter Constraint Satisfaction.

Magee, J. and Kramer, J. (1996a). Dynamic structure in software architectures, *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM Software Engineering Notes.

Magee, J. and Kramer, J. (1996b). Self organising software architectures, *Proceedings of the Second International Software Architecture Workshop*.

Medvidovic, N. (1997). A classification and comparison framework for software architecture description languages, *Technical Report ICS-TR-97- 02*, University of California, Irvine, Department of Information and Computer Science.

Montanari, U. and Pistore, M. (1995). Concurrent semantics for the π-calculus, *Electronic Notes in Theoretical Computer Science* **1**.

Montanari, U. and Rossi, F. (1996). Graph rewriting and constraint solving for modelling distributed systems with synchronization, *Lecture Notes in Computer Science* **1061**.

Montanari, U. and Rossi, F. (1997). Graph rewriting, constraint solving and tiles for coordinating distributed systems. To appear in Applied Category Theory.

Shaw, M. and Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall.