Graph Grammars and Constraint Solving for Software Architecture Styles

Dan Hirsch[1] Paola Inverardi[2] Ugo Montanari[3]

[1] Dep. de Computación
Universidad de Buenos Aires
Ciudad Universitaria, Pab.I, (1428)
Buenos Aires, Argentina
dhirsch@dc.uba.ar

[2] Dipartamento Di Matematica
Universita' dell'Aquila
Via Vetoio, Localita' Coppito
L'Aquila, Italia
inverard@univaq.it

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

Abstract

The description of a software architecture style must include the structural model of the components and their interactions, the laws governing the dynamic changes in the architecture, and the communication pattern. In our work we represent a system as a graph where 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 propose to use techniques of constraint solving. From this approach we obtain an intuitive way to model systems with nice characteristics for the description of dynamic architectures and reconfiguration and, a unique language to describe the style, model the evolution of the system and prove properties.

1 Introduction

A software architecture style is a class of architectures exhibiting a common pattern [1]. 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 graph grammars to describe styles. So a grammar will generate all possible instances of that style. This approach has first been proposed in [2].

In this position paper we propose to represent a system as a graph where edges (or hyperedges) are components and nodes are ports of communication. To model the construction and dynamic evolution of the style we need to choose a way of selecting which components will evolve and communicate. For these we use a technique already applied in [3] 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*) [4].

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. The solution of the rule-matching problem is implemented considering it as a finite domain constraint problem [5].

In the case of software architectures we use constraint rules to coordinate the dynamic evolution of the system. This is done using constraints (conditions) on ports to represent communications between components and (if necessary) to control changes in the configuration of the system.

In [2], 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*.

The main difference between the two approaches is that in our work we give a uniform description of the *complete style* with grammars. Also, we don't have a global coordinator of evolution. Instead, each type of component defines its own evolution making it easier to describe dynamic reconfiguration and specially for self organising architectures that do not use a global coordinator [6]. In this way, a clearer view of the system is obtained while a separation of configuration and evolution is achieved (a desirable property of software architecture description languages) [7]. Another important point is that the evolution and communication pattern can be obtained directly by the rewriting sequences on the graphs, analogously to what happens in the CHAM description of software architectures [8]. And this is fundamental for the verification of properties of the system.

2 Basic Notions

An edge-labeled hypergraph is defined as in [9], as a tuple that contains a set of nodes, a set of edges and labeling functions for nodes and edges. Each edge can be connected to a list of nodes (hyperedge) and a set of distinguished external nodes is given.

In our approach we will just consider *context-free productions* of the form $L \to R$, where L is the (graph containing only the) hyperedge to be rewritten and R is the graph to be generated. A production $L \to R$ can be applied to a graph G yielding H if there is an *occurrence* of L in G. The result of applying the production to G is a graph H which is obtained from G by removing the occurrence of L and adding R.

To model coordinated rewriting, it is necessary to add some conditions to the nodes in the left side of productions. In this way, each rewrite of an edge must match conditions with its adjacent edges which should in turn evolve as well. These conditions will be used to coordinate interactions among components (communications, synchronization on shared data, etc.). 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 the shared node (a and b). If $a \neq b$, then the edges cannot rewrite together (using those rules). If $a = b$, then they can move to a new state.

3 Modeling 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 mean that there is a communication link between the two components.

A software architecture style is described by a hyperedge context-free grammar. The productions of a grammar are grouped in three sets. The construction of an instance of a style (graph grammar) begins with the application of the first set of productions to obtain a desired initial configuration of the system. After this, the last two sets of rules can be applied to model the evolution of the architecture. The second set represents the rules for the dynamic evolution of the configuration and are used to create and remove components dynamically. If necessary, some of these rules can be constrained. For example, rules of this type can be used to coordinate simultaneous or ordered creation and destruction of components, and to model local and coordinated termination.

The third set of rules describes the communication pattern of the style. These rules are constrained productions that during rewriting will synchronize to model the evolution of the system. Because of the use of context-free productions and constraints we obtain independent specifications of the different types of components and the only relation between them is by the communication coordination using constraints.

Software architectures may require complex interactions among components. The explicit and independent specification of connectors, help to achieve a higher level of reusability. 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.

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

3.1 Client-Server

The example is a client-server case study based on the one used in [2].

The software architecture style is represented as a hyperedge context-free grammar. Edge labels are drawn as boxes and have two parts. One is the component name and the other is the status of the component that represents its different states during evolution. One difference with [3], is that in our approach, we will rely on two basic types of communication paradigms: point-to-point and broadcast communications. This allows to represent both types of communication at the same time. Broadcast ports are drawn 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 $\overline{a}$ 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.

The example has three types of components: 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 case all nodes are point-to-point ports.

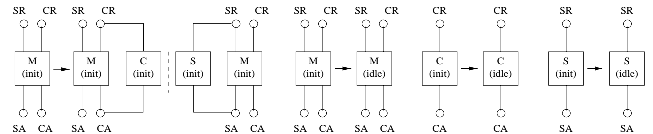

Figure 1: Client-Server: Static Productions

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.

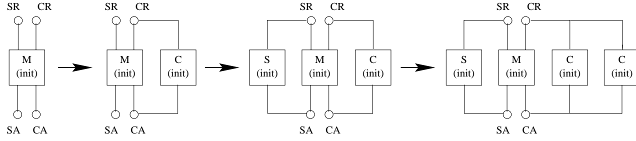

Figure 2: Client-Server: An instance of the architecture style generated by the static productions

After the desired initial configuration is obtained, then $(init) \rightarrow (idle)$ 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.

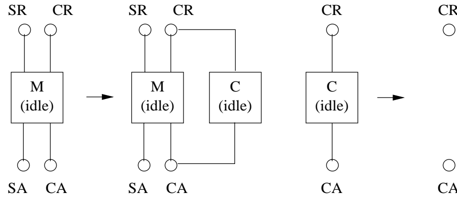

Figure 3: Client-Server: Dynamic Productions

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.

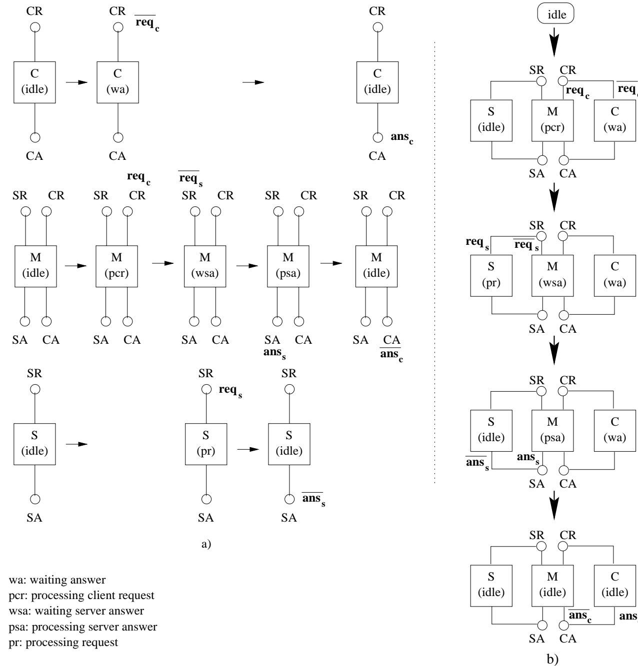

Figure 4: Client-Server: Communication Pattern Productions

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).

It is worthwhile mentioning that the generality of the approach allows to choose the level of abstraction for the description of the communication pattern, and with this to obtain different levels of detail. 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*).

Summarizing, we present a language for the description of software architecture styles. This includes the description of the communication pattern and the dynamic changes in the topology of systems. The use of context-free productions and constraint solving allows a separation of coordination and configuration and is well suited for the explicit modeling of dynamic reconfiguration. Also note that by analyzing the derivation tree of the grammar it is possible to have all the computations of the system permitting the verification of properties of the architecture, like for example, deadlock [10, 11].

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, that a simple description of systems with a unique language is obtained; the use of constraints to model coordination of

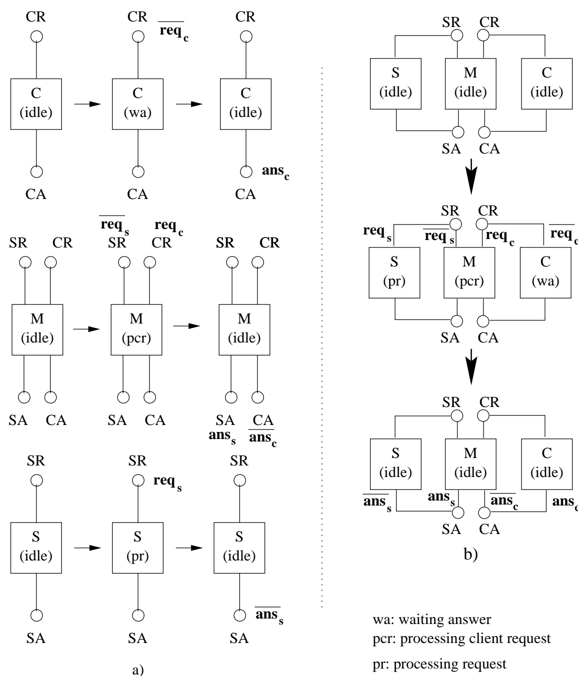

Figure 5: Client-Server: Communication Pattern Productions. An alternative

components allows a clear description of component interactions and controlled dynamics; and the inheritance of the distributed solutions for the constraint-solving problem. Also, context-free rules are a natural way for modeling the behavior of components independently of each other, allowing a distributed implementation.

Finally, the productions we use are all rewriting rules, but an interesting extension is to incorporate refinement rules where the history of the system is remembered. 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 [12].

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* COORDINA *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* FOMEC Project 376, Contract 164.

References

[1] Shaw, M. and Garlan, D., *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

[2] Le Métayer, D., "Describing software architecture styles using graph grammars," *IEEE Transactions on Software Engineering*, to appear.

[3] Montanari, U. and Rossi, F., "Graph rewriting, constraint solving and tiles for coordinating distributed systems," 1997, To appear in Applied Category Theory.

[4] Corradini, A., Degano, P., and Montanari, U., "Specifying highly concurrent data structure manipulation," in *COMPUTING 85:A Broad Perspective of Concurrent Developments*, Bucci, G. and Valle, G., Eds. Elsevier Science, 1985.

[5] A.K. Mackworth, *Encyclopedia of IA*, chapter Constraint Satisfaction, Springer Verlag, 1988.

[6] Magee, J. and Kramer, J., "Self organising software architectures," in *Proceedings of the Second International Software Architecture Workshop*, 1996.

[7] Medvidovic, N., "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, 1997.

[8] Inverardi, P. and Wolf, A., "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, Special Issue on Software Architectures.

[9] Drewes, F., Kreowski, H.-J., and Habel, A., "Foundations," in *Handbook of Graph Grammars and Computing by Graph Transformation*, G. Rozenberg, Ed., vol. I, chapter 2. World Scientific, 1996.

[10] Degano, P. and Montanari, U., "A model for distributed systems based on graph rewriting," *Journal of the Association for Computing Machinery*, vol. 34, no. 2, April 1987.

[11] Compare, D., Inverardi, P., and Wolf, A., "Uncovering architectural mismatch in dynamic behavior," To appear.

[12] Magee, J. and Kramer, J., "Dynamic structure in software architectures," in *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1996, ACM Software Engineering Notes.