Specification and Analysis of Component Based Software Architectures

P. Ciancarini, and C. Mascolo
Dip. di Scienze dell'Informazione, University of Bologna
Mura Anteo Zamboni, 7, I-40127 Bologna, Italy
e-mail: {ciancarini,mascolo}@cs.unibo.it

Abstract

We present a coordination language for the specification of software architectures. A model checker built on this language has been used to perform analysis on configuration of components. We reason on the assumptions that components make on their contexts and analyze how different assumptions can match and how components can be interconnected. We can predict which context allows a component to behave exploiting all its functions.. The coordination model we adopt fits this kind of reasoning.

1 INTRODUCTION

The growing complexity of modern software systems increases the need of rigorous formalization of structural and behavioral issues on the basis of their design.

In this paper we show that a coordination language can be naturally used as an architectural description language. The coordination model allows the study of the behavior of single components, of their assumptions about the environment, and of the possible components interconnections. In particular we perform some analysis on single components and on configurations using a model checker we have built for our language. The structure of the model checker allows the verification of properties on isolated parts of the architecture and the consequent verification of hypothetical configurations composed of multiple components.

Modern software architectures often contain mobile components and reconfiguration is a frequent event. The analysis of components as isolated items could help in the reconfiguration phase: the assumptions of every component about the extern environment are taken into account and matched in order to obtain acceptable configurations.

2 OVERVIEW OF POLIS

PoliS is a coordination language based on nested tuple spaces. We now briefly introduce the language but more details can be found in [3]. A PoliS specification is hierarchically structured: it denotes a tree of nested spaces whose structure evolves dynamically in time.

A space can contain both other spaces and tuples of two types: *ordinary tuples*, which are ordered sequences of values, and *program tuples*, which contain the coordination rules that manage local activities inside the space they belong to. The execution of a program tuple is an action, which can modify a space tree removing tuples and adding tuples and spaces. However, an action can only handle the tuples in the space it belongs to or in its parent space. This constraint defines both the "input" and the "output" environment of any action. Every program tuple ("r" : R) refers to a rule R specified below the description of the space containing the program tuple. The *rule* is the construct that defines which reactions can take place. A rule can act on the tuples of the space in which it resides and in the tuples of the parent space of this space: we will call this spaces the rule scope. A rule defines a reaction that reads and consumes tuples in its scope, performs a sequential computation, produces new tuples in its scope and creates new subspaces. Rules are first class entities in PoliS: in fact, they are themselves part of spaces as (program) tuples that can be read, consumed or produced just like ordinary tuples. A program tuple has the form (*rule_id*: *rule*) where *rule_id* is a rule identifier and *rule* is a PoliS rule. A program tuple has an identifier which simplifies reading or consuming program tuples.

Table 1 contains the specification of a client-server system. The $StartContext$ space is the main space, that contains the program tuple ("*create*" : $CREATE$). The program tuple indicates that the rule $CREATE$, specified below in Table 1, is contained in the main space. A key feature in PoliS is that a space tree can evolve dynamically: a new space is created by the primitive **tsc** (for *tuple space create*) and any space can be removed because of the execution of a special rule named *invariant* that terminates the space where it is executed. For instance, the rule $CREATE$ of Table 1, contained in the main space, creates the spaces $Client$ and $Server$. $Client$ space contains the tuple ("$idle$", i) that indicates the state of the client, the program tuple ("req" : REQ), and the program tuple ("put" : PUT) that refer respectively to the rule REQ and PUT specified below. The rule REQ emits a new request (tuple) in the main space: $\uparrow$ ("$request$", i), and changes the state of the client from ("$idle$", i) to ("$wait$", i) where i is the number associated to the request. The rule GET waits for an answer in the main space $\uparrow$("$answer$", $answ, i$) where i corresponds to the number of the request sent (the rule checks if the tuple ("$wait$", i) is present). $Server$ contains a tuple indicating the state and three rules: the rule $GETREQ$ checks if the state is idle and a request is present in the main space, then moves the request in the local space. The rule $SERVE$

generates an answer to the request. The rule PUT resets the state of the server to idle and moves the answer tuple to the main space.

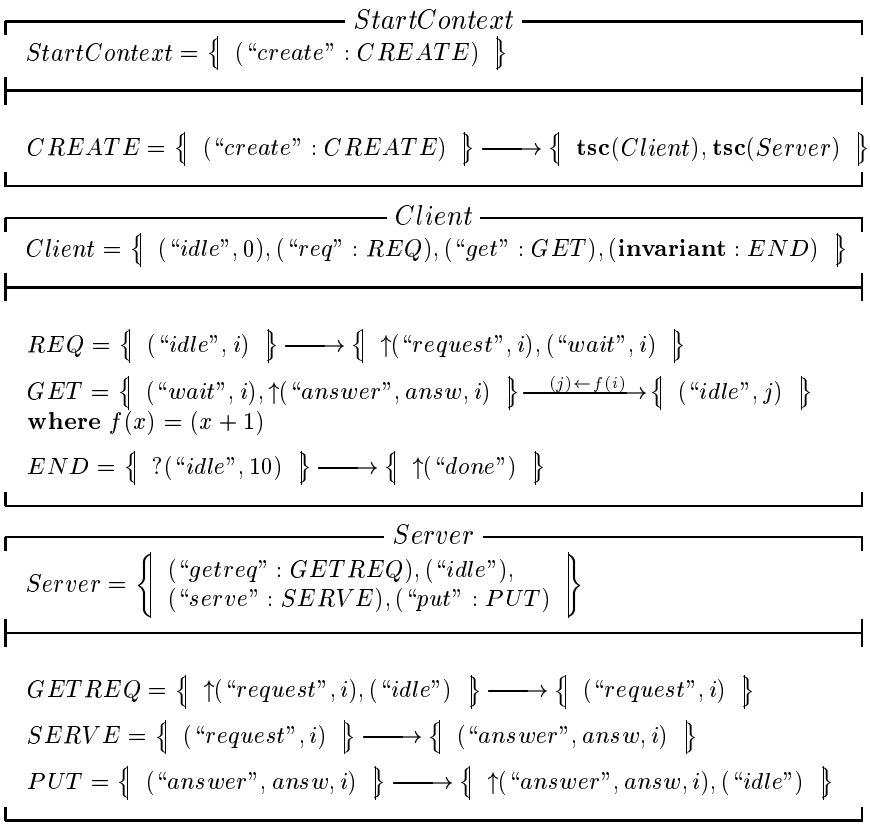

Table 1 Specification of the Client-Server System

In order to partially constrain activities inside a tuple space we can define one or more *invariants*, namely constraints that must hold for all the tuple space lifetime. Whenever an invariant is violated, the tuple space terminates and disappears. For instance the rule END in our example is an invariant.

3 POLIS AND SOFTWARE ARCHITECTURES

Research in the field of software architecture has led to the definition of several environments and languages for the definition and the design of architecture of software systems. Some works face the problem of defining a general-purpose language for architectural description, supporting system design by correct

combination of given interacting subsystems. Other works aim to characterize systems design according to defined style constraints, developing style specific environment to guide the building of specific systems [5]. Other architectural description languages have been developed exploiting well-known formalism as CSP [1] or π -calculus [8] providing also tools for animation and monitoring [7].

We now show how we use PoliS for the specification of software architectures putting emphasis on the interaction among components. The basic entity of the PoliS language is the *Tuple-Space*: an architectural component is specified using a space. When necessary, a component can be seen as composition of different sub-components. We specify this kind of compositionality in PoliS exploiting the multiple tuple spaces structure: each composed component is specified with a PoliS space containing other sub-spaces. For instance, a server component can be seen as a single space or as composed of different entities (i.e. sub-spaces) handling different kind of requests or providing different services.

The coordination model is a good framework to abstract from communication details. At the architectural level we would like to have an abstract view of the system: the tuple-based communication mechanism let the focus be put on the structure. On the other side, if the specification of the connection is important, it is possible to associate with the connector a space in order to define its particular behavior. For instance, in the example shown in Table 1 the client and the server communicate through the tuple space using this coordination abstraction. We could modify the model adding an entity, with the function of connector (i.e. a Buffer or a Router) in order to specify its particular behavior. This connector can also be composed of different sub-components, for instance a Layered Router: the nested space model fits the specification of this layered structure. The PoliS spaces model allows the specification of context-free components as independent spaces with their active rules. The PoliS mechanism of active rules scoping helps in the definition of the components assumptions on the external environment. For instance, consider a generic rule enabled only when a particular tuple is present in the parent space ($\uparrow(tuple)$): the component containing that rule should be put in a configuration that will eventually provide that tuple, otherwise parts of the component behavior will be unexploited (with consequences that can lead to the deadlock of the system). In this way we can reason on the assumptions that components make on their contexts and analyze how different assumptions can match and how components can be interconnected. We can predict which context allows a component to behave exploiting all its functions. These kinds of reasoning could help in the organization of the architectural configuration. Furthermore, the help of automatic tools for the testing of these properties could be devised. In this direction we propose the use of our PoliS model checker: we introduce this topic in the next section.

4 MODEL CHECKING SINGLE COMPONENTS AND ARCHITECTURES

In this section we outline a technique to check the behavior of components as isolated from the context. We can also make interesting proofs on the properties of composed architectures, where the components analyzed before are put in relation and interact. The configuration matching can be performed on multiple components.

This sort of analysis is possible as the model checker works bottom-up on the spaces, building graphs for the innermost ones and then going on recursively. Other key issues in this sort of compositionality analysis are the assumptions that a space (i.e. component) makes on the environment. The PoliS language provides a particular scoping mechanism: the reactions contained in a space can make assumptions on the external space (i.e. the parent of the local space) using the $\uparrow$ operator and formal tuples (not instanced) (see Section 2 for details).

A component (i.e a space) that is put in a context (i.e. an other space) uses pattern matching mechanism to match the assumptions contained in its rules (i.e. the tuples with "$\uparrow$") with the actual tuples contained in the environment. In this way we can easily state when a component will be able to have an useful behavior exploiting its functionalities and when not. If the environment does not provide the tuples that the component needs, the behavior of the component will be constrained and its capabilities will not be completely exploited. In a previous work [4] a mapping between PoliS operational semantics and TLA (Temporal Logic of Action) has been studied. This allowed us to use a theorem prover for formal reasoning on PoliS specifications. In this work instead we exploit a model checking technique to perform architectural analysis on PoliS specification documents.

4.1 Analysis of Software Architectures

We show how the model checker can be used for the verification of properties on software architectures. We first analyze single components out of their context, considering their interactions with the environment. Then we will be able to analyze configurations and saying if they are feasible and convenient. The study of components as isolated entities is useful when dealing with complex architectures where components are not elementary objects but they are composed of many parts.

We now show how a single component can be analyzed out of its context. Consider the Server in the Client-Server example (1): the Server makes only one assumption on the external context, that is, it remains idle till a request is present in its context (i.e. the father space) $(\uparrow(\text{"}request\text{"}, i))$, then a $GETREQ$ reaction can take place and after some steps an answer is generated in the

environment ($\uparrow(answer, answ, i)$). The model checker can be used to prove this property:

$$\forall i, a, C((\text{“}request\text{”}, i), Server) \in C \rightsquigarrow (\text{“}answer\text{”}, a, i) \in C \tag{1}$$

That is, if the context C of the Server guarantees the arrival of a request, then the answer to the request will be provided. The Client can emit a request without checking the context C, however it blocks if the context does not provide an answer (rule GET). Then, if an answer is provided the Client can go on making requests till the number of requested services is ten.

$$\begin{aligned} &\forall i, a, C(Client, (\text{“}answer\text{”}, a, i)) \in C \rightsquigarrow \\ &(((\text{“}request\text{”}, i+1) \in C) \vee (\text{“}done\text{”}) \in C) \end{aligned} \tag{2}$$

We can put together the assumptions of the two components and try to check if our Client-Server configuration is feasible.

$$(Client, Server, (\text{“}request\text{”}, i) \wedge (i < 10)) \in C \rightsquigarrow \tag{3}$$

$$(\text{“}answer\text{”}, a, i) \in C \rightsquigarrow \tag{4}$$

$$((\text{“}request\text{”}, i+1) \in C \vee (\text{“}done\text{”}) \in C) \tag{5}$$

We can trivially reach a state satisfying (3) in fact the Client can emit a request (with $i < 10$). The first “leads to” ($\rightsquigarrow$) property is satisfied by (1) as just shown, and the second “leads to” property is satisfied by (2). Hence, we conclude that the two components form a feasible configuration and that the corresponding assumptions match.

The Client-Server is a simple example without reconfiguration problems due to mobility of components. The introduced approach of analysis can be very useful to know if a mobile component could be introduced or not in a particular sub-architecture. For instance, if we introduce an agent in our Client site (space) and want to send it to the Server site in order to avoid heavy communication due to exchanging of requests-replies messages, we could analyze the Agent space and its assumptions on the environment and see if they match with the Server space contents.

5 RELATED WORK AND CONCLUSIONS

We have presented PoliS, a coordination language, based on multiple tuple spaces, for the specification of software architectures. Components are formalized as spaces and connectors can be specified either using simple abstract tuple production/consumption mechanism, or building more sophisticated spaces for connector entities and specifying their behavior.

We use a model checking technique for PoliS to perform verification on the compatibility of components. Every component makes assumptions on its context. When put together the components assumptions have to match in order to obtain a meaningful software architectural configuration.

The Cham model [2] has been exploited to perform similar checks in [6]: we think PoliS offers an immediate abstraction (the rule scope) for these kinds of checks.

Modern software architectures often deal with mobile components and the diffusion of Internet based systems imply the need of formalization of architectural patterns based on mobility paradigms. At the moment we are using PoliS to describe and study architectures including mobile agents [3]. We are studying mobility from the architectural point of view: the study of assumptions matching of mobile architectural components in the context of reconfigurable architectures seems to be a very interesting research field. We are studying these aspects that could be analyzed also in terms of security.

REFERENCES

[1] R. Allen and D. Garlan. Formalizing Architectural Connection. In *Proc. 16th IEEE Int. Conf. on Sw Eng.*, pages 71–80, Sorrento, Italy, 1994.

[2] G. Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96:217–248, 1992.

[3] P. Ciancarini, F. Franzé, and C. Mascolo. A Coordination Model to Specify Systems including Mobile Agents. In *Proc. 9th IEEE Int. Workshop on Software Specification and Design (IWSSD)*, pages 96–105, Japan, 1998.

[4] P. Ciancarini, M. Mazza, and L. Pazzaglia. A Logic for a Coordination Model with Multiple Spaces. *Science of Computer Programming*, 31(2/3):231–262, July 1998.

[5] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting Style in Architectural Design Environments. In D. Wile, editor, *Proc. 2nd ACM SIGSOFT Symp. on Foundations of Software Engineering*, volume 19:5 of *ACM SIGSOFT Software Engineering Notes*, pages 175–188, New Orleans, USA, December 1994.

[6] P. Inverardi, A. Wolf, and D. Yankelevich. Checking assumptions in components dynamics at the architectural level. In D. Garlan and D. LeMetayer, editors, *Proc. 2nd Int. Conf. on Coordination Models and Languages*, volume 1282 of *Lecture Notes in Computer Science*, pages 46–63, Berlin, Germany, September 1997. Springer-Verlag, Berlin.

[7] D. Luckham et al. Specification and Analysis of System Architecture using RAPIDE. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[8] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schafer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, volume 989 of *Lecture Notes in Computer Science*, pages 137–153, Sitges, Spain, September 1995. Springer-Verlag, Berlin.