

Model Checking a Software Architecture

P. Ciancarini, and C. Mascolo
Dipartimento di Scienze dell'Informazione
Università di Bologna
Mura Anteo Zamboni 7, I-40127 Bologna, Italy
phone: +39 51 354506
e-mail: {cianca,mascolo}@cs.unibo.it

ABSTRACT

Developing notations and tools for describing and analyzing software architectures is currently a main research issue in software engineering. Although there is no universally accepted definition, most researchers agree that an Architectural Description Language should allow to describe and analyze the structure of large software systems. This paper shows that PoliS, a coordination language, can be effectively used to design and analyze software architectures. We demonstrate that PoliS satisfies the requirements of ADLs and develop a model checking framework to reason on PoliS specifications.

KEYWORDS

Software Architecture, Coordination, Model Checking

1 INTRODUCTION

Software Architecture deals with: “*the structure of the components of a program/system, their interrelationships, and guidelines governing their design and evolution over time [7].*” Like this one, almost all other definitions are operational, that is, no one explains what exactly is a software architecture, but all suggest how an architectural view of software system can be useful during design.

Although there is no standard, universal definition, most share some basic concepts: components, connections or connectors, and constraints or configuration. More precisely, an *architectural language* is a formal notation that should be able to describe software systems as the composition of independent *components* via *connections* [9].

Components are system elements which process and store data. We can describe a component as a public interface defined by a set of ports which define points of access to services and a “*black box*” that performs computations, hiding internal implementation. The way data are manipulated classifies different types of components: filters, managers, controllers, memories etc. Defining the behavior of a component is not trivial task: in this case the descriptions based on module interfaces or on an object oriented style lack expressiveness be-

cause usually they cannot handle “*live*” entities like processes or agents. Connections are linking elements: data flow between connected components. Allen and Garland [1] study connection protocols like independent entities and call them *connectors*. They list and analyze different types of connections: pipe, procedure call, message passing, event notification etc., each one differing from the others in the way data are carried between linked components. In a real system some components are connected with some connections on the basis of constraints that specify which components are linked together and the connection between them. We call such a set of constraints a *configuration*. A system can have a static or a dynamic configuration. A configuration is static if its structure is invariant at run-time, otherwise it is dynamic.

Specifying dynamics is a specialty of coordination languages. A *coordination language* is “*the linguistic embodiment of a coordination model*” [2]. A coordination language should orthogonally combine two languages: one for coordination (the inter-agent actions) and one for (sequential) computation (the intra-agent actions). In this paper we show that a coordination language can be naturally used as an architectural description language.

Our paper is organized as follows: Section 2 introduces PoliS; Section 3 studies how PoliS can be used to specify a software architecture. Section 4 introduces model checking for PoliS and describes a PoliS Temporal Logic used for formal reasoning on specifications. In the last section we describe our current work.

2 OVERVIEW OF POLIS

PoliS is a coordination language based on nested tuple spaces. A tuple space, or *space* for short, includes both tuples and other spaces. 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.

A space is modified by reactions that transform multi-sets of tuples in multi-sets of tuples (this is multi-set rewriting, and is common to most coordination models based on *generative communication*). 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.

More precisely, a rule is made up of a *preactivation*, a *local computation*, and a *postactivation*. The preactivation is a multi-set of tuples to be found in its scope; the local computation is any sequential computation which does not modify the tuple space; the postactivation is made up of a multi-set of tuples to be produced in its scope and of a set of spaces to be created. Notice that this is a very general definition; actually rules need not to be made up of all the admitted components: a rule can have an empty preactivation, it can involve no local computation, it can produce no tuples and it can create no spaces. The preactivation can include *formal tuples*, that are tuples whose fields can be identifiers; moreover, it includes the primitive **ask**, that permits to check the values that are assigned to the identifiers of a formal tuple matched against a tuple in the space. The semantics of a program tuple PT is that a reaction takes place in a space if the space itself includes both PT and a multi-set of tuples matching the preactivation of PT. A *match* relation checks whether a multi-set of formal tuples M_{ft} can be instantiated by a multi-set M_{gt} of ground tuples. Consequently, such a *match* relation is defined between pairs of multi-sets of tuples and not between pairs of tuples. The tuples of the preactivation must be read or consumed in the rule scope. When a rule can be activated in a space, the reaction can takes place: the tuples to be consumed locally are removed from the space where the reaction takes place, the tuples to be consumed externally are removed from the parent space of the space where the reaction takes place, the local computation is performed, the tuples and the new spaces of the postactivation are created. A program tuple is a multi-set rewriting rule: preactivation and postactivation are multi-sets and the local

computation is written as annotation on the arrow between preactivation and postactivation. A tuple in the preactivation must be read if the symbol “?” is put in front of it and must be consumed otherwise; a read or consume operation involves the parent space if the symbol “ \uparrow ” is put in front of a tuple and involves the local space if the symbol is missing; a tuple in the postactivation must be produced in the parent space if the symbol “ \uparrow ” is put in front of it and must be produced locally otherwise.

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. Whenever disjoint multi-sets of tuples satisfy the activation preconditions of a set of rules, such rules can be executed independently and simultaneously: every rule modifies only the portion of space containing the tuples that must be read or consumed and therefore other rules can modify other tuples in the space or other spaces.

A simple example helps in explaining both syntax and semantics of PoliS. Let us consider a client-server system. A client emits requests and the server serves them. Such a system can be described by two distinct spaces both included in the main space representing the client and the server.

Table 1 contains the specification of the 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. The execution of a rule containing a **tsc**(M) operation in its postactivation causes the multi-set M to be added as a child space of the space where the rule was executed.

For instance, the rule *CREATE* of Table 1, contained in the main space, creates the spaces *Client* and *Server* that contain the tuples describing the client and the server respectively. *Client* is the client space and 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 associ-

ated to the request. The rule *GET* waits for an answer in the main space $\uparrow(\text{"answer"}, \text{answ}, i)$ where i corresponds to the number of the request sent (the rule checks if the tuple $(\text{"wait"}, i)$ is present). It emits a new state tuple with the number i increased by one by the function f on the arrow in the rule (specified in the **where** clause).

Server is the server space. It 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 (emitting the tuple $(\text{"idle"}, i)$ locally). and moving the answer tuple to the main space.

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. A PoliS invariant is a condition on the tuple space contents: it asserts that the space will never contain a given multi-set of tuples. Invariant rules can only read tuples locally (the tuples that must not belong to the tuple space) and produce tuples in the parent space. When the tuples to be read are in the space, the reaction specified by the invariant takes place in the usual way. Local computation and tuple production are used to communicate possible results to the parent space and then the space dies. Invariants are given by means of special program tuples whose names are replaced by the keyword **invariant**.

Going back to our example, if we want the Client space to terminate as soon as it receives the 10-th answer, we have to put an invariant rule like the one shown in Table 1 (*END*). The invariant fires when the client space contains a tuple $(\text{"wait"}, 10)$ and the main space contains the tuple $(\text{"answer"}, \text{answ}, 10)$. The tuple $(\text{"done"}, i)$ represents a termination signal sent by the consumer to the main space.

3 SOFTWARE ARCHITECTURE AND POLIS

Our purpose is to verify the architectural expressiveness of PoliS.

We first give the definition of components and connections that are the building blocks of architectures. The choice is quite simple for components: we represent them with spaces, or in the simplest cases, with simple tuples. The case of connections is more complex: we choose to describe a connection with a set of tuples contained in the same space containing the components to be connected.

According to our definition, we can build complex components starting from simpler ones, from multisets properties we can easily infer closure property for these operations. We can classify four methods to build up a component:

1. Building a space from a multiset of tuples:

$$t_1, t_2, \dots, t_n \mapsto \{\{t_1, t_2, \dots, t_n\}\}$$
2. Inclusion of a tuple in a space:

$$t_i, S \mapsto \{\{t_i\} \oplus S\}$$
3. Union of two spaces:

$$S_1, S_2 \mapsto S_1 \oplus S_2$$
4. Inclusion of a space in another space:

$$S, \{\{t_1, t_2, \dots, S_1, S_2, \dots\}\} \mapsto \{\{S, t_1, t_2, \dots, S_1, S_2, \dots\}\}$$

Method 1 is like a space *constructor*, while method 2 models an hereditary relation among spaces, in fact we can say that a new space, obtained from an old one by the introduction of a new tuple, inherits the characteristics of the former. Methods 3 and 4 describe composition rules, which define three ways to compose two spaces (say A and B):

- A and B are put in a new space C .
 We have this case by the application of method 4 for two times:
 $A, \{\{ \} \} \mapsto \{\{A\}\},$
 $B, \{\{A\}\} \mapsto \{\{A, B\}\}$
- A is put inside B or B is put inside A , by applying again method 3 in a special case.
- A and B are melted, by applying method 3

In the first case connections are contained in space C , whereas in the second one they are contained in the larger space, and in the last case they are contained in $A \cup B$.

4 MODEL CHECKING A SOFTWARE ARCHITECTURE

In a previous work [5] 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.

Model checking was introduced for hardware system verification [6]. Recently it has been used also for software systems, but we know only one previous example in which it has been used with a coordination model [3]. We explore its use for architectural analysis.

<i>StartContext</i>
$StartContext = \{ \{ ("create" : CREATE) \} \}$
$CREATE = \{ \{ ("create" : CREATE) \} \longrightarrow \{ \{ \mathbf{tsc}(Client), \mathbf{tsc}(Server) \} \}$
<i>Client</i>
$Client = \{ \{ ("idle", 0), ("req" : REQ), ("get" : GET), (\mathbf{invariant} : END) \} \}$
$REQ = \{ \{ ("idle", i) \} \longrightarrow \{ \{ \uparrow("request", i), ("wait", i) \} \}$
$GET = \{ \{ ("wait", i), \uparrow("answer", ans, i) \} \xrightarrow{(j)-f(i)} \{ \{ ("idle", j) \} \}$
where $f(x) = (x + 1)$
$END = \{ \{ ?("idle", 10) \} \longrightarrow \{ \{ \uparrow("done") \} \}$
<i>Server</i>
$Server = \{ \{ \uparrow("getreq" : GETREQ), ("idle"), ("serve" : SERVE), ("put" : PUT) \} \}$
$GETREQ = \{ \{ \uparrow("request", i), ("idle") \} \longrightarrow \{ \{ ("request", i) \} \}$
$SERVE = \{ \{ ("request", i) \} \longrightarrow \{ \{ ("answer", ans, i) \} \}$
$PUT = \{ \{ ("answer", ans, i) \} \longrightarrow \{ \{ \uparrow("answer", ans, i), ("idle") \} \}$

Table 1: Specification of the Client-Server System

The aim of model checking is to find an assignment (a *model*) for system variables that satisfies some formulae describing some system properties. Given an operational specification of a software system, a model checker builds a model and then it makes an exhaustive checking of variable values. This method could seem trivial and inefficient, but it is very powerful for systems with finite state models.

In order to specify the architectural properties to be proved, we introduce a temporal logic. The Polis Temporal Logic (PTL) is a CTL [6] dialect. The main differences between PTL and CTL depend on the definition of our model, that is based on multi-sets (spaces): all formulae are evaluated in a context (a space); we also assume that formulae without an explicit context are evaluated in the *StartContext*. An atomic proposition *atom* is a tuple; we say that proposition *atom* is true in a context C if it belongs to space denoted by constant C. We have also added the classical logic operators and some temporal operators to improve formulae representation and understanding.

- A *ptf* can be a *context*, a *temporal*, a *classic*, a parenthesized *ptf*, or an *atom*; a *ptf* can be universally or existentially quantified over some variables;
- a *context* is a PTL formula that has a pattern like:

$ptf \in C$ (space C), $ptf \in \star C$ (all C spaces), $ptf \in \& C$ (some C spaces), or $ptf \in \% C$ (exactly one C space), because a specification can include more than one instance of a space;

- a *temporal* is a CTL formula: the canonical operators **A** (for all paths) and **E** (at least a path does exist) for path quantification are described respectively by symbols \star and $\&$. **X** and **U** are PTL symbols for CTL operators Next and Until;
 - $\star \diamond ptf$ is defined as $\star(\mathbf{trueU} ptf)$: it means “for all paths *ptf* will eventually be true”;
 - $\& \diamond ptf$ is defined as $\&(\mathbf{trueU} ptf)$: it means “for at least a path *ptf* will eventually be true”;
 - $\star \square ptf$ is defined as $\neg \& \diamond \neg ptf$: it means that “for all paths *ptf* is always true”;
 - $\& \square ptf$ is defined as $\neg \star \diamond \neg ptf$: it means that “for at least a path *ptf* is always true”;
 - $ptf \rightsquigarrow ptf'$ is defined as $\star \square (ptf \Rightarrow \star \diamond ptf')$: it means that “for all paths it is always true that *ptf* implies that for all least a path *ptf'* will be eventually true”.
- a *classic* is a PTL formula with classical logic operators;
- an *atom* is simply a tuple.

PoliMC is a model checker for PoliS. The model checker gets two inputs: a system specification written in PoliS, and a set of properties to be verified written in PTL. PoliMC first parses the PoliS specification and builds up a model for it. Starting from the SOS formal operational semantics of PoliS we also defined a transition system. The graph obtained from unfolding a transition system of a real system is something quite similar to our model of the system.

The main difference between SOS unfolding and our model is that in SOS a unique monolithic graph is built to represent the system, while we have a graph for each space definition. Nodes show how spaces evolve and edges are labelled with tuples produced/consumed and tested in the parent spaces. PoliMC works recursively starting from the more nested spaces, going up to the root space (*StartContext*), using the information collected during the visit.

After having built the graph, the checker parses PTL formulae and builds syntax trees including only CTL operators, finally PoliMC can start performing model checking. Its algorithm follows the guidelines given in [6]: the checking is performed recursively starting from simpler sub-formulae (which are deeper in a syntactic tree), a difference to remark is that each formula is checked inside its context, that is the model checker make the checks using the graph of the space (context).

4.1 Testing the Client-Server Architecture

We have used the model checker to test some liveness properties on the Client/Server system shown in Tab.1. The model checker helped us in increasing our confidence in the specification document and in understanding some features of the specification.

For instance, we tried to prove the property:

$$\forall a, i(idle) \in Server \rightsquigarrow (answer, a, i) \in Server$$

That is, if the server is idle, then eventually it will serve a request. The model checker proved that this property is false. In fact, if the client has already requested ten services, the invariant rule is enabled in the client space, the client will not ask for other services, and the tuple (*idle*) will always remain in the server space.

5 CONCLUSIONS AND RELATED WORKS

We studied PoliS, a coordination model for formally describing software architectures. We showed how it is possible to map components and connections using PoliS spaces and tuples. We also applied a model checking technique to test the specification of a software architecture, using a simple temporal logic (PTL) to describe properties. This technique is not universally usable. PoliS is basically a rewriting multiset system, so we have

that architectures including a great number of different components result in a very large graph system, hard to build and check.

Modern software architectures often deal with mobile components and the diffusion of Internet based systems imply the need of formalization of new architectural patterns based on mobility paradigms. At the moment we are using PoliS to describe and study architectures including mobile agents [4]. We believe that PoliS can have an interesting application in the specification of mobility aspects: the coordination allows flexible moving of components and extensibility of the architecture.

We are also comparing PoliS to another multiset-rewriting based architectural language, namely the CHAM [8].

Acknowledgements: We would like to acknowledge the help and contribution of Francesco Franzè.

REFERENCES

- [1] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, June 1997.
- [2] N. Carriero and D. Gelernter. Coordination Languages and Their Significance. *Communications of the ACM*, 35(2):97–107, February 1992.
- [3] X. Chen, P. Inverardi, and C. Montangero. ESP-MC: An Experiment in the Use of Verification Tools. In K. Kanchanasut and J. Levy, editors, *Proc. Asian Computing Science Conference*, volume 1023 of *Lecture Notes in Computer Science*, pages 396–406, Thailand, Dec 1995. Springer-Verlag, Berlin.
- [4] P. Ciancarini, F. Franzè, and C. Mascolo. A Coordination Model to Specify Systems including Mobile Agents. In *Proc. 9th ACM-IEEE Int. Workshop on Software Specification and Design (IWSSD)*, page (to appear), Japan, 1998.
- [5] P. Ciancarini, M. Mazza, and L. Pazzaglia. A Logic for a Coordination Model with Multiple Spaces. *Science of Computer Programming*, (to appear), 1997.
- [6] E. Clarke, E. Emerson, and A. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.
- [7] D. Garlan and D. Perry. Introduction to the Special Issue on Software Architecture. *IEEE Transactions on Software Engineering*, 21(4):269–274, April 1995.
- [8] P. Inverardi and A. Wolf. Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering*, 21(4):373–386, April 1995.
- [9] M. Shaw and D. Garlan. *Software Architecture. Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.