Process Algebra–Guided Design of Java Mobile Network Applications.

Marco Carbone, Matteo Coccia, Gianluigi Ferrari, Sergio Maffeis

Dipartimento di Informatica
Università di Pisa, Italy

Highly distributed networks have now become a common platform for wide-area applications which use network facilities to access remote resources and services. WEB applications distinguish themselves from traditional distributed applications mainly because they have to deal with *dynamic* and *unpredictable* changes of network environment, e.g. availability of network connectivity, lack of resources, dynamic service creation, network reconfiguration, and so on. Mechanisms to control how WEB applications can be dynamically assembled, extended and reconfigured are therefore the key programming abstractions.

We propose a programming model and a design discipline for developing Java-based (JINI–like) applications which exploit a programmable coordination language amenable to formal verification. Designers are forced to develop applications by clearly separating the computational modules from the coordination ones. The distinguishing feature of our approach is that the coordination language takes the shape of a distributed process calculus which provides mechanisms for mobility of coordinating agents and for explicit distribution of modules and their dynamic assembling over wide-are networks. The separation between computation and coordination makes the calculus amenable to be effectively analyzed with formal techniques. Hence behavioural properties can be stated and possibly established, enforcing correctness for the programmable coordination policies.

Our coordination/scripting language ED_π is based on a variant of Hennessy and Riely's Distributed π-calculus (D_π) [3], and extends it with new mechanisms to manage locations, to cope with IP name spaces, and with a new communication primitive allowing more powerful interactions (e.g. negotiation of services). ED_π is endowed with a formal LTS semantics and a structural congruence relation amenable to formal proof techniques such as bisimulation-based equivalences, control-flow analyses and many others. Related work includes several scripting/architecture-description languages: in particular Piccola [1], but also Darwin [5] and Nomadic Pict [6], and the application of formal methods to Java (here we mention [4, 2]).

Follows an example explaining some of the features of the language and its intended use: the definition of a service-fetching abstract operation $FIND$, meant to retrieve on behalf of some client sitting at location *Client*, a service *Service* starting the search from a given server location *Server*.

$\mathsf{rec}\ \mathrm{FIND}(Client, Server, Service, Result).$
$\mathsf{go}\ Server.lookUp(Client^{\uparrow}, Service^{\updownarrow}, R^{\downarrow})?$
$\quad(\mathsf{go}\ Client.\overline{Result}(R^{\uparrow}).$
$\quad\ lookUp(Client^{\uparrow},\ (\mathtt{addr},\ Service)^{\updownarrow},\ NewAddr^{\downarrow})?$
$\quad\quad(\mathrm{FIND}(Client, NewAddr, Service, Result),$
$\quad\quad\ \mathsf{go}\ Client.\overline{Result}(\mathtt{FAIL}^{\uparrow}).0))$

This process first moves ($\mathsf{go}\ Server.$), to the default server site. Then, by exploiting a conventional public channel ($lookUp$) it checks whether the service is available: $lookUp(Client^{\uparrow}, Service^{\updownarrow}, R^{\downarrow})?$ .Here, communication is a bidirectional synchronization via typed tuples and pattern matching. In the example at hand this feature allows us to disclose the client's identity if and only if the server offers the required service. In this case, the network reference to the service is returned as a result. The arrows specify for each argument of the typed tuple whether it is an input, output or synchronization parameter. If the request for the service succeeds, the process goes back to the original location and communicates the result through the channel $Result$. Otherwise it asks the server for an alternative address for the service and recursively begins the fetch protocol. Notice that typed tuples provide the mean to negotiate service requests by explicitly stating a specific range of values, e.g. the minimum level of service that components are willing to accept and the maximum level that they are able to use.

The Java implementation consists in a package providing classes for each of the process-algebra constructs, so to stress on the programmer's side the connection with the underlying formal model. Work in progress includes a security type system for the language, static analyzers and an extensive revision of the implementation.

References

1. F. Achermann, S. Kneubuehl, O. Nierstrasz. Scripting Coordination Styles. In Proc. Proc. Coordination'01, LNCS 1906, 2001.
2. S. Drossopoulou. Towards an abstract model of Java dynamic linking and verification. In Proc. TIC'2000, CMU Technical Reports, CMU-CS-00-161, 2000.
3. M. Hennessy, J. Riely. Type-safe execution of mobile agents in anonymous networks. In J. Vitek, C. Jensen, Eds. *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, LNCS State-Of-The-Art-Survey, NCS 1603, Springer, 1999.
4. S. N. Freund and J. C. Mitchell. A formal framework for the Java Bytecode Language and Verifier. In *ACM OOPSLA '99*, pp. 147–166.
5. Jeff Magee, Naranker Dulay, Susan Eisenbach, Jeff Kramer Specifying Distributed Software Architectures In. Proc. of 5th European Software Engineering Conference (ESEC '95), LNCS 989, (Springer-Verlag), 1995.
6. Sewell, P., Wojciechowski, P.Nomadic Pict: Language and Infrastructure Design for Mobile Agents. IEEE Concurrency, 2000.