Agent-based Configuration Management

Matthias Radestock and Susan Eisenbach

Department of Computing
Imperial College of Science, Technology and Medicine
180 Queen's Gate, London SW7 2BZ, United Kingdom
E-mail: {M.Radestock,S.Eisenbach}@ic.ac.uk

Abstract. The application of agent technology in the context of distributed systems provides the basis for a new, flexible approach to the management of such systems. In this paper we illustrate how agents are used in *configuration management*. The approach views a distributed system as a collection of distributed agents with a clear separation between various concerns that have to be addressed in distributed systems. The configuration aspect is described by defining *configuration scripts* on agents using the language *Evolution*, thus defining a *configuration management interface*. The implementation of the system is based on a high-performance interpreter for an actor-based language. Reconfiguration is not just confined to system structure but can also affect the agent state and hence the behaviour of an agent. Agents can be (re-)defined at run-time, which enables the dynamic definition of new agent types and the addition/modification of configuration scripts. The construction of complex management structures can be carried out by defining *management agents*. The configuration management in our system can be carried out via a simple WWW interface, thus enabling *remote support*. The architecture enables the provision of advanced tool support for configuration management and its integration with other aspects of system management.

1 Introduction

We can view a distributed system as a collection of distributed agents[1] that interact with each other [MDK93, GS93, PW92, GP94]. The concerns of a distributed system can be separated into four parts (cf. Fig. 1):

- The *communication part* defines *how* agents communicate with each other.
- The *computation part* defines the implementation of the *behaviour* of individual agents. It thus determines *what* is being communicated.
- The *configuration part* defines the *interaction structure*, or *configuration*. It states which agents exist in the system and which agents can communicate with each other, as well as the method of communication. Basically it is a description of *where* information comes from and *where* it is sent to.
- The *coordination part* defines patterns of interaction, ie. it determines *when* certain communications take place.

The communication part is the only part that is totally application independent, and thus features in the design and implementation as something that is being *used*, rather than defined or altered. The computation, configuration and coordination parts all include application dependent elements. However, each of them also has its own set of general, application independent requirements. In addition to this, inter-part dependencies yield a layered system structure. The coordination layer depends on the configuration layer because it requires information about the interaction structure in order to determine possible communications. The configuration layer depends on the computation layer since it needs to know which kinds of agents have been defined in order to be able to create new agents and to establish with which other agents an agent can communicate. The computation layer depends on the communication layer for the exchange of information.

[1] We use the term *agents* in a somewhat lax way as denoting an entity that 'does something'.

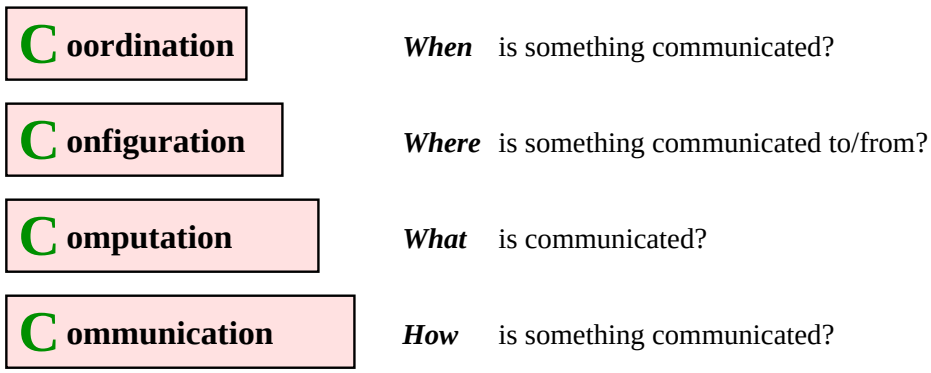

Fig. 1. The four layers of a specification of a distributed system

The separation of concerns into four layers permits reuse of elements in different context, eg. the behaviour description of an agent can be reused by another agent in another system. Each of the layers contributes to the overall system behaviour. The higher layers are of a more abstract nature than lower layers since they correspond to issues of *programming in the large* whereas the lower layers are concerned with *programming in the small*. Configuration management therefore affects the system behaviour on a rather abstract level. It is on this level that the majority of system management tasks are performed. According to the roles of the various layers we can define the term *configuration management*[CDF+95] as follows

> *Configuration management* is the management of the configuration aspect of a system. It enables the system manager to dynamically configure a system by specifying which agents are part of the system and which agents can communicate with each other.

2 Configuration Languages

Each of the layers can employ a different language. This emphasises and enforces the separation of the layers. More importantly the use of targeted languages results in more clarity and compactness and reflect the various aspects more intuitively. For the definition of the configuration layer of a system we employ a *configuration language*. The following are some key concepts that are commonly found in configuration languages. They are a direct result of the nature of the language domain, ie. configurations, and in this combination they only apply to the configuration layer:

Interfaces. Agents have *interfaces* that symbolise provided and required services. *Bindings* between requirements and provisions represent the fact that an agent requiring a service can use a service provided by another agent. Thus we obtain a more detailed specification of the allowable communications between agents.

Decomposition. Agents are part of a decomposition hierarchy. The definition of a composite agent specifies which agents it contains and bindings to, from and between interfaces of these agents and interfaces of the composite agent. This feature enables the construction of more complex agents from primitive agents.

Types. Agents are typed. Agents of the same type share certain configuration characteristics such as the same kinds of sub-agents and the same bindings. Agent *instantiation* is parameterised, which enables configuration when an agent type is instantiated. This yields a high degree of abstraction, which can be increased further by adding the concept of inheritance. In essence the definition of an agent type defines what happens when an agent type is instantiated, eg. which interfaces and sub-agents need to be created and which bindings need to be established.

The above concepts facilitate reuse of agent definitions in different contexts. Several configuration languages supporting these concepts exist [Pur94, BWD+93, Gra91, RS94, AG94, MEK95, Arb96].

All the four layers of the specification of a distributed system result in particular requirements for an implementation of such a system – certain run-time system support is needed. In the case of the configuration layer, the concepts supported by the configuration language have to be mapped to those supported by the run-time system. This is typically done by a compiler for the configuration language. The run-time system functionality can be used for configuration management - which by its very nature needs to be done at run-time. However, there are several problems with such a *static* approach:

- The run-time system functionality is less abstract than the configuration language, thus complicating configuration management.
- The configuration management, unlike the configuration language, is implementation dependent. This limits reuse and portability.
- The configuration management is totally dissociated from the specification of the configuration layer of the system in the configuration language. The inherent relatedness of the two thus cannot be exploited.

The aim of our research was to overcome these limitations. Our approach is based on a more dynamic view of configuration languages as dynamically interpreted languages. The full expressiveness and layers of abstraction provided by the configuration language can be brought to use for configuration management. Our research therefore addresses the important question of *how* configuration management is performed:

> Configuration management is performed by evaluating expressions of the configuration language.

Configuration management is lifted to the same level of abstraction and implementation independence as configuration languages. However, such generality has a price. We require an interpreter for our configuration language that is smoothly integrated into the entire system, so that we have the ability to examine and modify configurations. These requirements can be satisfied by basing the implementation of the distributed system on agent technology.

3 *Evolution*

In order to perform configuration management using a configuration language that language has to be dynamic, supporting all the configuration concepts in a dynamic fashion. We chose the configuration language *Darwin*[MDEK95, MD93] as a basis for our new language *Evolution*. *Darwin* allows for a dynamic configuration in a restricted sense – some possible dynamic changes can be indicated at compile time. The main advantages of *Darwin* over other configuration languages are that it allows high degree of abstraction, is a very small language and makes very few assumptions about the other system layers.

As the implementation basis of our distributed system we chose the *Rosette*[TLM+92, TKS+89] language. This language represents agents as actors [Agh86], is fully interpreted and has support for cross-platform distribution. The language extends the basic actor model by adding object-oriented concepts similar to SmallTalk [GR89]. Crucially for our work *Rosette* is reflective and enables the definition of extensions to the language itself. *Evolution* is implemented using this mechanism and is therefore completely integrated into the *Rosette* system. This considerably eases the integration of the configuration layer with the remaining system layers.

In this paper we will introduce *Evolution* with an example – a HiFi system. The following is the definition of a band-pass filter agent type. It inherits from a more general filter agent type. The instantiation of the agent type can take two parameters, defining the frequency range that the filter will let through. Both parameters have a default value and can thus be omitted.

```
(defAgent BandPass (extends& Filter)
  (formals& loFreq      300.0
            hiFreq      5000.0))
```

3.1 Configuration Scripts

Evolution generalises the concept of *configuration scripts*. Static configuration languages embed exactly one configuration script in the definition of agent types. It specifies the configuration management actions that are to be performed when the agent type is instantiated. In *Evolution* several scripts can be associated with an agent, defining the so-called *management interface* of the agent. Each script is bound to a management operation. Instantiation is one such operation, making the static configuration support just a special case of the dynamic configuration management.

The instantiation script for the band pass agent type above is shown in Fig. 2 together with a graphical representation of the resulting structure. A band pass agent is a composition of a low pass

```
(defScript BandPass (instantiate)
  (require 'in)
  (provide 'out)
  (inst* ['lo   LoPass [hiFreq]]
         ['hi   HiPass [loFreq]])
  (bind* [['lo 'in]     'in]
         [['hi 'in]     ['lo 'out]]
         ['out          ['hi 'out]])
  )
```

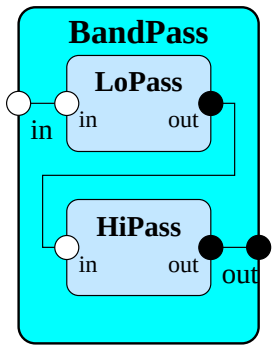

Fig. 2. Instantiation of the Band Pass Agent

agent and a high pass agent. It requires input and provides output. The interfaces of the agents are bound to create a pipe-like structure. To create an instance of a band pass for frequencies between 100 and 10000 Hz we would write `(new BandPass 100 10000)`. The operation `new` creates a new instance of an agent and then invokes the `instantiate` operation with the remaining arguments.

Configuration scripts are methods on actors. They are invoked by sending a message to the agent, typically resulting in reconfiguration of the agent. A configuration script can invoke configuration operations on the same or other agents. This enables functional decomposition and propagation of configuration activities between agents. The importance of the former is increased by the the fact that *Evolution* supports inheritance and polymorphism. Thus all the primitive configuration operations (such as `provide`, `require`, `inst`, `bind`) are defined on a single agent type from which all other agent types inherit. Agents can redefine these operations if they wish. The propagation of configuration activities between agents is typically carried out in a hierarchical fashion along the decomposition hierarchy. Examples of such hierarchical management operations are creation (ie. instantiation) and destruction of agents. Management operations confined to a single agent include the adding and removing of interfaces and bindings.

3.2 Specification of Dynamic Systems

The functionality of many applications requires the structure of a distributed system to change dynamically. For instance, conferencing applications need to support the joining and leaving of members and the establishing of communication links between various members. *Evolution* offers full support for the specification of such dynamic systems.

The computation aspects of an agent is represented in terms of methods and instance variables, which may refer to other agents. The parameters of the agent instantiation are special instance variables and the configuration scripts are special methods. The configuration functionality and associated state information are thus part of the overall agent functionality / state. The advantage of this uniform representation of configuration and computation aspects is twofold. Firstly,

reconfiguration is not confined to system structure but can also affect the agent behaviour because configuration scripts can invoke computation methods and change the computation part of the agent state. Secondly, reconfiguration can be triggered by the computation part. This makes reconfiguration an integral part of the system functionality, rather than a configuration management task triggered externally. It enables us to build highly dynamic systems while still preserving the division into coordination, configuration, computation and communication.

The following agent type definition describes a simple HiFi system consisting of a CD-player, tuner, amplifier and a pair of speakers.

```
(defAgent HiFiSystem
  (slots& current-source        'cd-player))

(defScript HiFiSystem (instantiate)
  (inst* ['cd-player    CDPlayer]
         ['tuner        Tuner]
         ['amplifier    Amplifier]
         ['left-sp      Speaker]
         ['right-sp     Speaker])
  (bind* [['amplifier 'in]      [current-source 'out]]
         [['left-sp 'in]        ['amplifier 'left]]
         [['right-sp 'in]       ['amplifier 'right]])
  )

(defScript HiFiSystem (switch-to new-source)
  (unbind ['amplifier 'in] [current-source 'out])
  (bind ['amplifier 'in] [source 'out])
  (assign current-source new-source)
  )
```

The instantiation of `HiFiSystem` doesn't require any parameters. A configuration-level instance variable `current-source` specifies which source is currently connected to the amplifier. The management interface of the agent contains a `switch-to` operation. When it is invoked it connects the input of the amplifier to the source supplied as parameter. Fig. 3 shows the system structure resulting from switching the source to the tuner after an instantiation of the `HiFiSystem` agent.

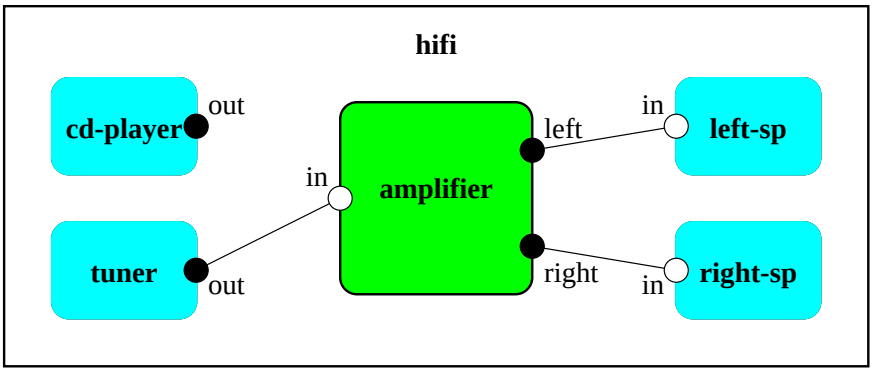

Fig. 3. HiFi System after Reconfiguration

The reconfiguration capabilities of the *Evolution* system go even beyond that. Because *Rosette* is fully interpreted and reflective we can actually define new agent types, modify existing ones,

add and modify configuration scripts and computation methods at run-time. This makes system management more powerful than it ever was. It means that changes to the system can be made that were not anticipated in the original specification. These changes can happen while the system is fully operational. In the above example we could, for instance, add a new management operation to unplug the source from the amplifier:

```
(defScript HiFiSystem (unplug)
  (if (not (niv? current-source))
      (unbind ['amplifier 'in] [current-source 'out]))
  (assign current-source #niv)
  )

(defScript HiFiSystem (switch-to new-source)
  (if (not (niv? current-source))
      (unbind ['amplifier 'in] [current-source 'out]))
  (bind ['amplifier 'in] [source 'out])
  (assign current-source new-source)
  )
```

Note that we also redefined the `switch-to` operation to account for the fact that no source may currently be plugged into the amplifier.

3.3 Management Agents

The scheme for defining management operations described so far is tightly bound to the decomposition hierarchy - the only agents known to the configuration layer of an agent, and hence the only agents on which management operations can be invoked, are the agent itself, its sub-agents and the containing agent. Thus, the role of *manager of an agent* can only be assigned to an agent that is close to the managed agent in the above sense. All the standard management operations (ie. operations that are bound at all agents) fit into this type management structure. Indeed, all management operations can be implemented within the structure as any agent can manage any other agent by propagating the management operations through the decomposition hierarchy. However, such a solution would neither be efficient nor flexible.

In *Evolution* we can use *reflection*[MWY91, JA92] to represent the management interface as a true configuration level interface provided by the agent. Hence bindings to this interface can be established by other agents. These agents can then perform the role of managers, with their computation aspects defining their behaviours. Managers can in turn be managed by other management agents. The scheme thus enables the description of complex configuration management structures using the configuration language itself.

In Fig. 4 we define a managing agent for our HiFi system. Note that `HiFiSystem` (as any other component) provides an implicit `manage` interface, representing the management interface.

4 Management Tools

In *Evolution* all configuration management activities can be performed using a command line interface. While this simple text based interface ultimately offers the most flexibility it requires the user to learn the syntax of *Rosette*. As an alternative we have therefore implemented a WWW interface. It enables the traversal of the decomposition hierarchy and the inspection of individual agents. All the primitive configuration operations are invoked with button clicks. Other operations defined in the management interface are equally easy to activate. Most day-to-day configuration management tasks can be carried out using the WWW interface. It also offers a cost-effective solution to the problem of remote support. In principle the entire distributed system can be managed remotely from any machine that is setup to run a standard WWW browser. Figure 5

```
(defAgent HiFiManager)

(defScript HiFiManager (instantiate)
  (require 'm)
  )

(defAgent System)

(defScript System (instantiate)
  (inst* ['hifi         HiFiSystem]
         ['manager      HiFiManager])
  (bind  ['manager 'm] ['hifi manage])
  )
```

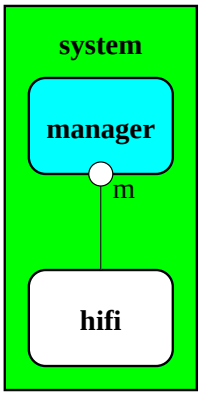

Fig. 4. Agents as Managers

shows the WWW configuration management interface to an instance of the band filter agent type defined earlier.

The interactive nature of *Evolution*, paired with the expressiveness and flexibility of *Rosette* can form a basis for the construction of configuration management tools. Such tools would either be written in *Evolution* and integrated into the system or they could access the configuration management capabilities via one of the remote access interfaces provided by *Evolution*. The latter enables the use of existing configuration management tools with graphical user interfaces, domain-based management capabilities etc.

5 Related Research

The issues of configuration management have received considerable attention from, the research community during the last couple of years. However, the approaches have mostly focussed on the management of systems with a static structure that only changes infrequently, usually due to intervention from the system administrator. Also configuration management has been viewed as being quite distinct from configuration programming. By contrast, our approach is fully integrated with configuration programming. In fact we view configuration management as nothing but *interactive* configuration programming. *Evolution* smoothly integrates the configuration layer with the other layers in a system. That makes configuration an intrinsic part of system functionality which can be accessed dynamically. It enables the construction of systems with a highly dynamic structure. Instead of relying mostly on human intervention, configuration management can be performed by management agents.

In [CDF+95] a configuration management tool-set for *Darwin* has been outlined. It uses the run-time system functionality of *Darwin* to perform configuration tasks. The drawbacks of such an approach were – lack of abstraction, implementation dependence, dissociation from the specification of the configuration layer. Since *Darwin* is not an interactive language, the functionality of the configuration management tools is inherently limited – they cannot change the behaviour of agents and it is impossible to define new kinds of agents. *Darwin* is also limited with respect to the definition of a dynamic configuration layer. Management agents cannot be defined. Nevertheless *Darwin* has proved to be suitable for the construction of many distributed systems and the functionality provided by the configuration management tools is sufficient in most cases. *Evolution* can be seen as a logical extension of *Darwin* that inherits many of the key concepts, such as components, interfaces and bindings. Using the *Darwin* configuration tools for configuration management in *Evolution* should be a straightforward and worthwhile task.

VISIFOLD is a visual programming environment for systems based on the MANIFOLD[Arb96] configuration language. Extensions are planned to enable the monitoring and debugging of running

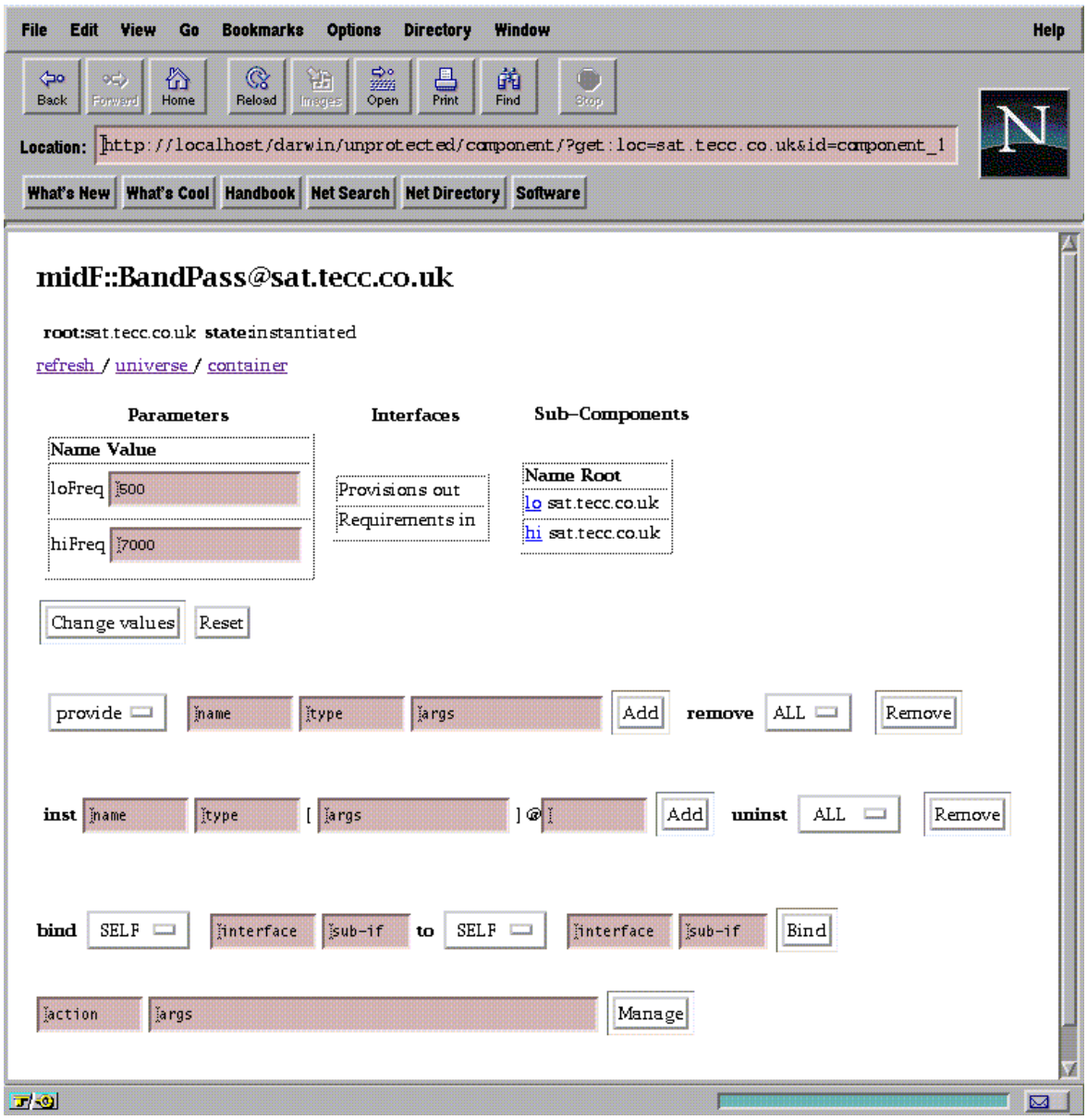

Fig. 5. Configuration Management via the World-Wide-Web

systems. These extensions would use the run-time system for performing their tasks, which may include configuration management activities. Thus the system suffers from the drawbacks of a run-time system based approach. In common with *Darwin*, MANIFOLD is not an interactive language and thus is subject to the same limitations. MANIFOLD has an event-based mechanism for triggering configuration changes. Dynamic configuration layers are thus easy to define. Actual communication between agents is stream-based. Since this is incompatible with the event-based communication used for dynamic reconfiguration, management agents cannot be defined.

6 Summary and Future Work

We have introduced a new approach in configuration management that is based on the application of agent technology. Configuration management is performed by evaluating expressions of a dynamic configuration language, called *Evolution*. This yields a high degree of implementation independence and abstraction. *Evolution* allows the specification and management of highly dynamic systems by smoothly integrating the configuration layer with the coordination, computation and communication layer of a system. Management structures can be defined freely using management agents. The configuration management can be carried out remotely via a WWW interface. *Rosette*'s remote access capabilities ease the integration with new and existing configuration management tools.

We will continue our work on the *Evolution* language by adding further useful constructs. The aim is to enables the specification of more and more complex management operations in *Evolution* itself (ie. the configuration part of a system) rather than the computation part. It is important though to retain the level of generality currently present in *Evolution*. The current prototypical *Evolution* system is going to be extended to provide better support for code distribution and agent migration. Security issues need to be addressed. A translator from *Rosette* to *Java*[Rit95] and visa versa will make dynamic cross-platform distribution easier and allow the integration of the increasing number of *Java* applets becoming available. A prototype for CORBA[MZ95] interoperability exists. We are aiming to make *Rosette* fully CORBA compliant. *Evolution* can then serve as a configuration language for CORBA systems. *Rosette* itself is currently being extended to provide built-in support for coordination. Ultimately this will lead us to a system where coordination, configuration and computation are smoothly integrated while still maintaining an open architecture.

7 Acknowledgements

We gratefully acknowledge the advice and help provided by the Distributed Software Engineering Research Section at the Imperial College Department of Computing (`http://www-dse.doc.ic.ac.uk`), TECC (`http://www.tecc.co.uk`), and the financial support provided by the EPSRC under grant ref: GR/K73282.

References

[AG94] R. Allan and D. Garlan. Formalizing architectural connection. In *Proc. of the 16th Int. Conf. on Software Engineering*, May 1994.

[Agh86] G.A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.

[Arb96] F. Arbab. The iwim model for coordination of concurrent activities. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models*, volume 1061 of *LNCS*. Springer Verlag, April 1996.

[BWD+93] M. Barbacci, C. Weinstock, D. Doubleday, M. Gardner, and R. Lichota. Durra: A structure description language for developing distributed applications. *IEE Software Engineering Journal*, 8(2):83–94, March 1993.

[CDF+95] S. Crane, N. Dulay, H. Fossa, J. Kramer, and M. Sloman. Configuration management for distributed software services. In *Proc. of the Int. Symposium on Integrated Network Management, Santa Barbara, USA*. Chapman & Hall, May 1995.

[GP94] D. Garlan and D. Perry. Software architecture: Practice, potential and pitfalls. In *Proc. of the 16th Int. Conf. on Software Engineering*, May 1994.

[GR89] A. Goldberg and D. Robson. *SmallTalk-80: The Language.* Addison-Wesley, 1989.

[Gra91] H. Graves. Lockheed environment for automatic programming. In *Proc. of KBSE'91, 6th IEEE Knowledge Based Software Engineering Conf.*, pages 68–76, 1991.

[GS93] D. Garlan and M Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, volume 1. World Scientific Publishing Co., 1993.

[JA92] S. Jagannathan and G.A. Agha. A reflective model of inheritance. In *ECOOP'92 Proceedings*. Springer-Verlag, 1992.

[MD93] J. Magee and N. Dulay. Programming with Darwin - an introduction to configuration oriented programming. Imperial College Department of Computing Internal Report, February 1993.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Fifth European Software Engineering Conf.*, Barcelona, 1995.

[MDK93] J. Magee, N. Dulay, and J. Kramer. Structuring parallel and distributed programs. *Software Engineering Journal*, 8(2):73–82, March 1993.

[MEK95] J. Magee, S. Eisenbach, and J. Kramer. System structuring: A convergence of theory and practice? In K.P. Birman, F. Mattern, and A. Schiper, editors, *Theory and Practice in Distributed Systems, Proc. of the Dagstuhl Workshop*, volume 938 of *LNCS*. Springer Verlag, 1995.

[MWY91] S. Matsuoka, T. Wanatabe, and A. Yonezawa. Hybrid group reflective architecture for object-oriented concurrent reflective programming. In O. Nierstrasz, editor, *ECOOP'91 Proceedings*, LNCS. Springer-Verlag, 1991.

[MZ95] T. Mowbray and R. Zahavi. *The Essential CORBA: Using Systems Integration, Using Distributed Objects*. John Wiley & Sons, 1995.

[Pur94] J.M. Purtilo. The polylith software bus. *ACM Transactions on Programming Languages*, 16(1):151–174, January 1994.

[PW92] D.E. Perry and A.L. Wolf. Foundations for the study of software architectures. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[Rit95] T. Ritchey. *Java!* New Riders Publishing, Indianapolis, Indiana, 1995.

[RS94] M.D. Rice and S.B. Seidman. A formal model for module interconnection languages. *IEEE Transactions on Software Engineering*, 20(1):88–101, January 1994.

[TKS+89] C Tomlinson, W Kim, M Schevel, V Singh, B Will, and G Agha. Rosette: An object oriented concurrent system architecture. *Sigplan Notices*, 24(4):91–93, 1989.

[TLM+92] C Tomlinson, G Lavender, G Meredith, D Woelk, and P Cannata. The carnot extensible services switch (ess) - support for service execution. In Charles J Petrie Jr, editor, *Proc. of the First Int. Conf. on Enterprise Integration Modeling*, Cambridge, MA, 1992. MIT Press.

This article was processed using the LaTeX macro package with LLNCS style