Aster: A Framework for Sound Customization of Distributed Runtime Systems

Valérie Issarny Christophe Bidan

{issarny,bidan}@irisa.fr
IRISA, Campus de Beaulieu, 35042 Rennes Cédex, France

International Conference on Distributed Computing Systems
Hong-Kong, May 27-30, 1996.

Abstract

This paper introduces the Aster distributed configuration-based programming system that is aimed at easing the development of emerging distributed applications having quality of service requirements. Our approach is based on high-level customization: given the specification of application requirements using the Aster interconnection language, a distributed runtime system, customized for meeting these requirements is built. So as to make the customization process sound, we propose a formal method that allows to reason about specification matching of a customized distributed runtime system with the application's requirements.

1 Introduction

Progress in processor and network technology leads to an evolution of distributed computing systems: they are now eligible for supporting a wide class of applications, ranging from online multimedia services to massively parallel applications. However, for this use of computing systems to become effective, open issues relevant to the programming system as well as to the operating system areas need be addressed. Challenging issues include the best usage of the underlying hardware in order to support concurrent executions of high-demanding applications. Best usage of the available software must also be addressed in order to ease re-use and hence reduce the cost of software development. Our research is aimed at designing a base distributed programming system meeting such requirements, which leads to address the following issues:

- Support for software evolution, both at the application and runtime system levels since applications can no longer be considered as settled once installed.
- Support for software re-use to allow programmers to take benefit of available software, be it system-level or application-level.
- Support for handling various quality of service requirements such as timeliness, availability or security.
- Support for hardware and software heterogeneity.

Our design goals have led us to undertake an approach based on configuration-based programming because it supports naturally heterogeneity of programming languages, software re-use, and dynamic reconfiguration of applications (e.g., see [1, 2, 3, 4]). From the perspective of quality of service requirements, attempts are made to take them into account in existing configuration-based programming systems. Basically, proposed solutions consist of leaving some flexibility to the programmer for distribution management. For instance, the CORD system, extension of the Polylith system [5], allows the programmer to optimize Remote Procedure Calls (RPC) by choosing appropriate replication policies among a set of predefined ones [6]. Greater flexibility is left to the programmers in systems like Regis [4] and UniCon [7] through the provision of communication components. The programmer can then develop communication protocols customized to the application needs by implementing new communication components. However, to our knowledge, none of these proposals provides the application programmer with tools to help her/him to select the appropriate customized system, hence discouraging re-use of system-level software.

This paper introduces the Aster distributed configuration-based programming system that is aimed at easing the development of applications having various quality of service requirements through the re-use of system-level software. Our approach is based on *high-level customization*: requirements of an application, expressed by the programmer, are met, independently of the underlying architecture and the application's implementation, by means of distributed runtime system customization. Furthermore, so as to make the customization process sound, it relies on the definition of a formal method that allows to reason about the matching of the application's requirements with the runtime system's functions. This method further provides support for automatic selection and/or construction of customized systems.

This paper is structured as follows. Section 2 gives an overview of Aster basics, introducing its language and its default runtime system. Section 3 focuses on an example of customization, it shows how the communication model can be customized from the perspective of its failure semantics. Formal reasoning about the matching between the application's requirements and the underlying runtime system is then addressed in section 4. Finally, we draw some conclusions in section 5, discussing related work and implementation issues.

2 Overview of Aster

A configuration-based programming system includes an *interconnection language*, also called MIL (Module Interconnection Language), that allows to define interfaces of software components, and to specify the structure of an application in terms of interface interconnections. The runtime system, sometimes called *software bus* (e.g., see [5]), used for the application then manages interconnection of software components at runtime. The Aster interconnection language and default software bus are presented hereafter.

2.1 Interconnection language

Following existing configuration-based programming systems, the Aster interconnection language is a declarative language, independent of any implementation language. It enables to describe an application in terms of interconnection of software components where components are abstractly defined through their interfaces. More precisely, the Aster language allows to define software components of the following kinds: *interface* that declares the signatures of the component's client and service operations; *implementation* that binds an implementation file, which can be written in any programming language, to an interface component; and *hierarchical* that assembles a set of components (be they interface, implementation or hierarchical) through the appropriate bindings of client and service operations.

The two first types of components are simple. They have been extensively dealt with in the literature (e.g., see [5]) and hence are not further addressed in this paper. A hierarchical component is defined by stating: the set $\mathcal{S}$ of software components from which it is built, the bindings between operations of $\mathcal{S}$'s components, and its interface listing operations of $\mathcal{S}$'s components that are not bound.

Declaration of software components in the Aster language is illustrated by the following simple example, inspired by [5]. Let us consider an application implementing remote lookup within a dictionary. The application is made of two interface components: `User` and `Table` that respectively requests and services the `lookup` operation. The implementations for `User` and `Table` are respectively defined by the `IUser` and `ITable` implementation components. Finally, the application is defined by the hierarchical component `AccessDic` that binds the `lookup` operation. We get:

```
INTERFACE User {
  CLIENT string lookup(in string name) handles absent;};
INTERFACE Table {
  string lookup (in string name) raises absent;};
IMPLEMENTATION IUser {
  local.c IMPLEMENTS User;};
IMPLEMENTATION ITable {
  remoteTable.c IMPLEMENTS Table;};
HIERARCHICAL AccessDic {
  CONSTITUENTS IUser; ITable;
  BIND IUser lookup :  ITable lookup;};
```

2.2 Default software bus

Up to this point, we have outlined description of applications using the Aster language. Following the terminology used for the Polylith configuration-based programming system [5], the distributed runtime system for an Aster application is implemented by a software bus. A software bus defines a standard runtime interface for the application's software components: it handles component interactions.

A software bus is characterized by the functionalities it implements such as the communication protocol used. Purtilo introduces the notion of *abstract bus* as the abstract specification of such functionalities [5]. From an abstract bus, a *family of software buses* can be built. A software bus then defines an implementation of the abstract bus for a specific set of implementation languages, platforms, and communication media.

A software bus can be seen as another software component that provides an interface with which the application interconnects vertically. Briefly stated, vertical interconnection relates to the interconnection of software components that belong to distinct, adjacent, abstraction levels of the software architecture. This is to be contrasted with horizontal interconnection that refers to component interconnection at the same abstraction level. The bus interface defines at least two operations that respectively correspond to the emission and reception of a message on/from the bus, these operations being used to carry out component interactions. The default Aster abstract bus, called *AsterBus*, is defined by the following interface, declared using the Aster language:

```
ABSTRACTBus INTERFACE AsterBus {
    void receive (in pid src, dest; out msg m);
    void send (in pid src, dest; in msg m);
  PROVIDES
    PSync = Rpc{(sync:  synch)(failure:  at-most-once)};
    PAsync = Rpc{(sync:  async)(failure:  best-effort)};
  }
```

The declaration of *AsterBus* uses the *provides* clause that is used to state semantics implemented by the bus. The bus implements two types of bi-party RPCs for interaction management: synchronous, at most once, and asynchronous, best effort. The execution of software component interactions according to one of these two semantics depends upon application requirements for the interactions. The next section details how application requirements are expressed in Aster, and how they can be met by the appropriate selection of a customized bus.

3 Customizing Software Buses

So as to deal with various application requirements upon the distributed runtime system, declarations of software components are enriched with *requires* and *provides* clauses that allow the programmer to respectively specify system-level constraints and properties associated to components[1]. At the present stage of Aster definition, the only class of properties that is apprehended relates to the communication model. We are currently working on the integration of more elaborated requirements relating to quality of service such as security and availability.

[1] In the remainder, the term *component* is meant for any type of Aster components (i.e., hierarchical, interface, and implementation).

3.1 Expressing application requirements

In Aster, the expression of application requirements within components relies on the definition of a *property tree*, which specifies the set of properties recognized by Aster. This approach easily supports the introduction of new sets of requirements; it suffices to enrich the tree accordingly. The Aster property tree is structured as follows:

- Each node specifies a set of properties and is defined by a name.
- Each leaf specifies a set of properties corresponding to a given distribution management semantics, which are ordered according to the *stronger-than* relationship; each property is defined by a pair specifying the property's name and associated logical formula.

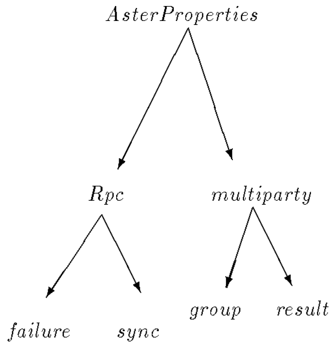

Figure 1: Aster property tree

Figure 1 states nodes of the current Aster property tree whose organization follows the classification of RPC semantics proposed in [8]. Concerning leaves definition, let us consider for illustration the RPC failure semantics, it is associated the following leaf:

$$(\textit{exactly\_once}, \mathcal{F}_1) \prec (\textit{at\_most\_once}, \mathcal{F}_2) \prec (\textit{best-effort}, \mathcal{F}_3)$$

where $\mathcal{F}_i$ and $\prec$ stand for the property's formula and the *greater-than* relationship respectively. Let us remark that the order on properties is only given as an indication, it can be inferred from the $\mathcal{F}_i$s. The formal definition of properties is further addressed in section 4.

Given the Aster property tree, the application programmer can specify the requirements associated to a given software component by means of the *requires* clause. This clause allows to state properties required for any operation, each property being specified as a path of the Aster property tree. For instance, let us consider the dictionary example. If the programmer wants to specify that the `lookup` operation of `AccessDic` must be executed according to the future, at most once semantics, `AccessDic` will be declared as:

```
HIERARCHICAL AccessDic {
  CONSTITUENTS IUser; ITable;
  BIND IUser lookup: ITable lookup;
  REQUIRES
    lookup:
      Rpc{(failure: at_most_once)(sync: future)};
  }
```

In the absence of property specification for a given client operation, the default semantics implemented for biparty communication is synchronous, at most once.

3.2 Customized software buses

In Aster, satisfying application requirements relies on the use of an appropriate software bus, possibly customized for meeting these requirements. Construction of software buses is similar to the one of applications: a bus is itself a configuration-based software. However, compared to an application-level software, a software bus is made of system-level components that are in charge of implementing properties of the Aster property tree.

There are two ways of building a customized software bus in Aster: explicitly by a system programmer, and automatically from software buses and system-level components available in the Aster environment. We examine these two approaches in turn.

Explicit customization. Given *AsterBus*, system programmers can build new software buses by interconnecting system-level software components with *AsterBus*. The behavior of a bus is then defined by an abstract bus that specifies properties satisfied by software buses implementing it. Such properties are stated in the *provides* clause, which is dual to the *requires* clause.

Following the Polylith proposal [5], abstract buses are implemented by software buses that are dedicated to a specific set of implementation languages, platforms, and communication media. Given the selection of an abstract bus $\mathcal{B}$ for executing an application, the software bus implementing $\mathcal{B}$ will be chosen according to the features of both the application's implementation and the execution platform.

Automatic customization. Through an adequate declaration of system-level components, Aster can build automatically a software bus, customized for a given application, from existing system-level components (including buses). For that purpose, the declaration of a system-level component states the properties implemented by the component together with the service operations to be used for ensuring each property.

For example, let us consider an application component requiring the exactly once failure semantics for its client operations. To implement this failure semantics from *AsterBus*, a system-level component undoing the effects of service operations is needed. Let us now assume that a system-level component implementing checkpointing, named `Checkpoint`, providing `save` and `restore` operations, is available within the Aster environment. Intuitively, coupling `Checkpoint` with *AsterBus* amounts to save the state of the called component prior to the execution of the service operation, and to restore this state if a failure occurs prior to the operation's termination. In other words, the `save` operation must be called when the call message is caught by the bus and the `restore` operation is to be called if a failure message is returned to the calling component instead of the result message. This is specified within the `Checkpoint` component by relating its service operations to the bus' operations:

```
SYSTEM INTERFACE Checkpoint {
    checkpt save (in idComp id);
    void restore (in checkpt idc);
    void release (in checkpt idc);
  PROVIDES
    undo: {
      checkpt x;
      (client,sendBus): {x:=save(id(server))};
      (client,receiveBus): {release(x)};
      (failure): {restore(x)}; };
  }
```

The above declaration implies that Aster is able to recognize that the combination of the undo and at most once properties ensures the exactly once failure semantics[2]. This is achieved through a formal method for reasoning about the properties of buses and system-level components, this method being described in the next section.

[2]Let us notice here that, for simplicity purposes, we assume that the service operation does not call remote service operations, and hence we do not deal with nesting.

We have exemplified the automatic construction of a specific customized software bus. The same principle applies for the implementation of any customization provided it relates to properties of the Aster property tree.

4 Reasoning about Software Buses

Up to this point, we have given the Aster base principles for meeting application requirements: given the specification of requirements within application-level software components, a customized software bus that meets these requirements is either selected or built –if possible– from available buses and system-level components. For this approach to be practical and sound, some means should be given to application programmers so that they can formally reason about the behavior of software buses and system-level components. The resulting formal method must further be decidable so as to support automatic selection and/or construction of customized software buses.

We are interested in the definition of a formal method that allows to soundly infer that a software bus matches the requirements of the application to be executed. This issue relates to the definition of a more general formal method currently under design for reasoning about the behavior of Aster applications. Such a method is aimed at addressing the problem of specification matching of software components at the various abstraction levels of the Aster software architecture, both horizontally and vertically. Horizontal matching includes specification matching of client and service operations, and of two software components when one is replaced by another in the context of dynamic reconfiguration. The issue of horizontal specification matching can be related to the work described in [9]; the proposed solution relies on formal specifications of components in terms of pre- and post-conditions, an implementation of specification matching using the Larch Prover (e.g., see [10]) being introduced.

In the following, we concentrate on vertical specification matching. Our approach is based on formal specification of Aster properties by means of predicates. Such a specification allows us to formally reason about vertical specification matching through the first-order predicate calculus and hence to deal with automatic selection and construction of customized software buses.

Vertical specification matching relies on an appropriate specification of system-level properties, which is addressed in the next subsection. The definition of vertical specification matching in then given in subsection 4.2. Finally, subsection 4.3 deals with automatic construction of customized software buses.

4.1 Specifying system-level properties

Properties offered by the current Aster property tree relate to three types of interaction semantics: synchronization, failure and topological. Due to space limitation, we concentrate on the formal treatment of the failure semantics in the following. The interested reader is referred to [11] for the treatment of the other semantics.

Failure semantics defined by the Aster property tree can be characterized by their behavior with respect to the emission and reception of a call request both in the presence and in the absence of failure. More precisely, these semantics can be described as follows, upon the emission of a call request on the bus:

- *best-effort*: the call is received at least once by the called component in the absence of failure;
- *at most once*: the call is received exactly once by the called component in the absence of failure, and is received once or none in the presence of failure;
- *exactly once*: the call is received exactly once by the called component in the absence of failure, and is received none in the presence of failure.

The above definitions allow us to characterize failure semantics through only send and receive actions, and failure occurrences, which leads to define corresponding *send*, *receive* and *failure* predicates. In the remainder, we use the following logical connectors: $\Rightarrow$ denotes implication, $\neg$ denotes negation, $\vee$ denotes logical or, and $\wedge$ denotes logical and.

Base predicates: *Let C_1 and C_2 be two components, then the emission of a message m from C_1 to C_2 is noted: $send(C_1, C_2, m)$.*

Dually, the predicate stating that m is received exactly once *by C_2 is noted: $receive(C_1, C_2, m)$.*

We further introduce the predicate $receive^+(C_1, C_2, m)$ that holds when m is received at least once *by C_2.*

Finally, the predicate noted $failure(C_1, C_2, m)$ is true when a communication or a node failure occurs between the emission of m by C_1 and the end of the treatment of m by C_2.

The definition of the *failure* predicate applies for both request and result messages. In the former case, the end of the message treatment is meant for the termination of the called operation on the side of the called component. In the latter, it is meant for the reception of the result value by the calling component. Given the above predicates, failure semantics can be defined formally as follows:

$$\begin{array}{l} \textit{best-effort}(C_1,C_2,req) \equiv \\ \quad \textit{send}(C_1,C_2,req) \wedge \\ \quad (\neg \textit{failure}(C_1,C_2,req) \Rightarrow \textit{receive}^+(C_1,C_2,req)) \end{array}$$

$$\begin{array}{l} \textit{at-most-once}(C_1,C_2,req) \equiv \\ \quad \textit{send}(C_1,C_2,req) \wedge \\ \quad ((\neg \textit{failure}(C_1,C_2,req) \Rightarrow \textit{receive}(C_1,C_2,req)) \wedge \\ \quad\; (\textit{failure}(C_1,C_2,req) \Rightarrow \\ \quad\; (\neg \textit{receive}^+(C_1,C_2,req) \vee \textit{receive}(C_1,C_2,req)))) \end{array}$$

$$\begin{array}{l} \textit{exactly-once}(C_1,C_2,req) \equiv \\ \quad \textit{send}(C_1,C_2,req) \wedge \\ \quad ((\neg \textit{failure}(C_1,C_2,req) \Rightarrow \textit{receive}(C_1,C_2,req)) \wedge \\ \quad\; (\textit{failure}(C_1,C_2,req) \Rightarrow \neg \textit{receive}^+(C_1,C_2,req))) \end{array}$$

where req is meant for a message that carries a client request.

The predicates defining failure semantics allow to specify properties that are either required or offered by a software component through adequate quantification. For instance, if the exactly once failure semantics is required for the execution of any client operation of a given component C_1, we get: $\forall C \in \mathcal{I}_{C_1}$, $\forall req$: $\textit{exactly-once}(C_1, C, req)$, where $\mathcal{I}_{C_1}$ stands for the set of components with which C_1 interacts as a client. A system-level component (including buses) implementing properties for any interaction, the predicate's parameters are all quantified universally. For instance, stating that a bus implements the at most once failure semantics is specified as: $\forall C$, $\forall C'$, $\forall req$: $\textit{at-most-once}(C, C', req)$.

4.2 Vertical specification matching

Given the specification of system-level properties, a bus is eligible for executing an application if the set of properties implemented by the bus satisfies the application's requirements. This is formally handled though the notion of *plug-in match*, noted $\lhd_{plug-in}$, between sets of properties. Let $\mathcal{P}_i$ denotes the set of properties $\{P_i^j\}$ implemented by component i, we define plug-in match as:

$$\mathcal{P}_1 \lhd_{plug-in} \mathcal{P}_2 \equiv \forall P_2^i \in \mathcal{P}_2 : \exists P_1^j \in \mathcal{P}_1 \mid P_1^j \Rightarrow P_2^i.$$

Notice that the $\lhd_{plug-in}$ relation corresponds to a formal interpretation of the customization notion. When a software bus B_1 is a customization of a software bus B_2, we have: $\mathcal{P}_{B_1} \lhd_{plug-in} \mathcal{P}_{B_2}$ where $\mathcal{P}_{B_i}$ stands for the properties implemented by B_i.

Given the definition of $\lhd_{plug-in}$, the selection of a customized software bus that meets the requirements of a given application to be executed is straightforward. Let $\mathcal{R}_A$ be the set of the application's requirements, and $\mathcal{P}_A$ be the set of properties implemented by the application's components, a software bus implementing the set of properties $\mathcal{P}_{bus}$ can then be selected for the application if and only if: $(\mathcal{P}_{bus} \cup \mathcal{P}_A) \lhd_{plug-in} \mathcal{R}_A$.

4.3 Automatic customization

For automatic construction of customized buses according to application requirements, we need be able to infer that the interconnection of a set of system-level components with an existing bus satisfies the properties expected by the application to be executed. This amounts to select a set of components such that the conjunction of their properties with the ones of the base bus satisfies at least the application's requirements. Properties declared by a system component have not to belong to the property tree, they can be any formula whose conjunction with properties declared in other system components and/or a software bus leads to a property of the tree.

We introduce the operator $\bowtie$ that defines the augmentation of the properties of a software bus B with properties implemented by a set of system components, $\{S_i,\ i = 1..n\}$. We define:

$$\mathcal{P}_B \bowtie \bigcup_{i=1..n} \mathcal{P}_{S_i} = \bigcup_{\forall P \in \mathcal{P}_B} (P \wedge \bigwedge_{i=1..n} \bigwedge_{j=1..m} P_{S_{i_j}})$$

where $\mathcal{P}_B$ denotes the set of properties implemented by B, $\mathcal{P}_{S_i}$ denotes the set of properties implemented by S_i, and $P_{S_{i_j}}$ denotes one of these properties.

Automatic construction of a customized bus can then be defined in terms of plug-in match. Let $\mathcal{S}_E$ and $\mathcal{B}$ be the set of available system components and software buses respectively. Let A be the application to be executed and $\mathcal{P}_A$, and $\mathcal{R}_A$ be the set of its properties and requirements respectively. A customized bus can then be built for A if the following predicate holds:

$$\begin{array}{ll} bus(A) \equiv & \exists B \in \mathcal{B}, \exists \{S_1, ..., S_n\} \in \mathcal{S}_E \mid \\ & \{((\mathcal{P}_B \bowtie \bigcup_{i=1..n} \mathcal{P}_{S_i}) \cup \mathcal{P}_A) \\ & \lhd_{plug-in} \\ & (\mathcal{R}_A \cup \mathcal{R}_B \cup \bigcup_{i=1..n} \mathcal{R}_{S_i})\} \end{array}$$

where $\mathcal{P}_B$ denotes the set of properties implemented by B, $\mathcal{P}_{S_i}$ denotes the set of properties implemented by system component S_i, and $\mathcal{R}_B$ and $\mathcal{R}_{S_i}$ denote the requirements of B and S_i respectively.

When $bus(A)$ holds, the customized bus that is used to execute A is the one that *best* guarantees A's requirements, i.e., it is the least customized bus and it minimizes the number of system components.

For illustration, let us assume that *AsterBus* is the only bus available within the Aster environment. Let us further consider a component C requiring the exactly once failure semantics. We have the following specifications for C and *AsterBus* respectively:

$$\mathcal{R}_C \equiv \\ \{\forall C' \in \mathcal{I}_C,\ \forall req\text{: } \textit{exactly-once}(C, C', req)\}$$

$$\mathcal{P}_{AsterBus} \equiv \\ \{\forall C_1,\ \forall C_2,\ \forall req\text{: } \textit{at-most-once}(C_1, C_2, req), \\ \forall C_1,\ \forall C_2,\ \forall req\text{: } \textit{best-effort}(C_1, C_2, req)\}$$

Let us now assume that the Aster environment provides the `Checkpoint` system-level component. The *undo* property of `Checkpoint` is formally defined as:

$$\mathcal{P}_{Checkpoint} \equiv \\ \{\forall C_1,\ \forall C_2,\ \forall req\text{:} \\ failure(C_1, C_2, req) \Rightarrow \neg receive^{+}(C_1, C_2, req)\},$$

which allows to conclude that:

$$(\mathcal{P}_{AsterBus} \bowtie \mathcal{P}_{Checkpoint}) \lhd_{plug-in} \mathcal{R}_C$$

holds.

From the standpoint of implementation, we are currently designing a tool in λProlog [12] implementing verification of vertical specification matching and selection of system-level components for the automatic construction of customized buses. The design of the tool is based on the work presented in [13] that proposes a tool for searching software components using specification matching.

5 Conclusion

This paper has introduced the Aster distributed configuration-based programming system that is aimed at easing the development of emerging distributed applications requiring quality of service. Our approach is based on high-level customization: given the specification of application requirements using the Aster language, a distributed runtime system, called *customized software bus*, customized for meeting these requirements is built. So as to make the customization process sound, we have proposed a formal method that allows to reason about specification matching of a customized software bus with the application's requirements.

Related work. System customization is becoming an increasingly widespread subject of interest. In particular, there has been much work in the operating system community that aims to support system customization. Existing proposals include definition of new operating system kernels so as to support efficient and safe customization (e.g., see [14, 15]), and system structuring in terms of high-level abstractions so as to ease the customization process (e.g., see [16, 17]). Work in the first category must be seen as complementary to ours; they provide solutions directed towards efficiency that could be used at the level of the base operating systems executing Aster. Proposals dealing with system structuring relate to our work since they provide solutions that can be seen as alternative to customized software buses. However, the undertaken approaches are aimed at lower level customization, they do not provide the application programmer with tools to help her/him to select the appropriate customized system.

Work has also been done in the programming language community so as to ease construction of customized systems (e.g., see [18, 19, 4, 7]). However, the existing proposals do not address correctness of the customization with respect to application requirements. More generally, they do not consider the formal verification of matching of application requirements with customized systems.

From the perspective of the formal method proposed for reasoning about matching of a customized software bus with an application's requirements, the closest work in this area is the one of [9]. It introduces a method for reasoning about specification matching of the software components constituting the application. The reference addresses a complementary issue that is matching of components at one abstraction level of the software architecture.

Implementation issues. In addition to our work on property specification for sound customization of distributed runtime systems, we are currently working on an implementation of Aster. As for other configuration-based programming systems, this requires a set of tools such as the ones described in [20]. Our approach for the implementation of Aster, discussed in [21], consists of building Aster basics upon the OMG's standard distributed software architecture CORBA [22]: the Aster language is an extension of the OMG's IDL and the Aster default software bus is built from an ORB. The CORBA environment allows us to take benefit of various implementations meeting the specifications that are part of the Object Management

Architecture [23], including system-level components.

References

[1] J. Magee, J. Kramer, and M. Sloman, "Constructing distributed systems in Conic," *IEEE Transactions on Software Engineering*, vol. 15, no. 6, 1989.

[2] J. Kramer and J. Magee, "The Evolving Philosophers Problem: Dynamic Change Management," *IEEE Transactions on Software Engineering*, vol. 16, no. 11, pp. 1293–1306, 1990.

[3] J. M. Purtilo and C. Hofmeister, "Dynamic reconfiguration of distributed programs," in *Proceedings of the Eleventh International Conference on Distributed Computing Systems*, 1991.

[4] J. Magee, J. Kramer, and M. Sloman, "Regis: A constructive development environment for distributed programs," *Distributed Systems Engineering*, vol. 1, no. 5, pp. 663–675, 1994.

[5] J. M. Purtilo, "The Polylith software bus," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 1, pp. 151–174, 1994.

[6] T. H. Kim and J. M. Purtilo, "Configuration-level optimization of RPC-based distributed programs," in *Proceedings of the Fifteenth International Conference on Distributed Computing Systems*, pp. 307–316, 1995.

[7] M. Shaw, R. DeLine, D. Klein, T. Ross, D. Young, and G. Zelesnik, "Abstractions for software architecture and tools to support them," *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 314–335, 1995.

[8] Y. M. Huang and C. V. Ravishankar, "Designing an agent synthesis system for cross-RPC communication," *IEEE Transactions on Software Engineering*, vol. 20, no. 3, pp. 188–198, 1994.

[9] A. M. Zaremski and J. M. Wing, "Specification matching of software components," in *Proceedings of the ACM SIGSOFT'95 Foundations of Software Engineering Symposium*, 1995.

[10] J. V. Guttag and J. J. Horning, eds., *Larch: Languages and Tools for Formal Specifications.* Text and Monographs in Computer Science, Springer-Verlag, 1993. With S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing.

[11] V. Issarny, C. Bidan, and T. Saridakis, "Aster: un système de configuration de logiciels fondé sur Corba," research report, Inria-Irisa, Rennes, France, 1996. To appear.

[12] D. A. Miller and G. Nadathur, "Higher-order logic programming," in *Proceedings of the Third International Conference on Logic Programming*, 1986.

[13] E. J. Rollins and J. M. Wing, "Specifications as search keys for software libraries," in *Proceedings of the Eighth International Conference on Logic Programming*, pp. 173–187, 1991.

[14] B. Bershad, S. Savage, P. Pardyak, E. Gunder, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers, "Extensibility, safety, and performance in the SPIN operating system," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pp. 267–284, 1995.

[15] D. R. Engler, M. F. Kaashoek, and J. O'Toole, "Exokernel: An operating system architecture for application-level resource management," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pp. 251–266, 1995.

[16] R. H. Campbell, N. Islam, R. Johnson, P. Kougiouris, and P. Madany, "Choices: Framework and refinement," in *Proceedings of the 1991 International Workshop on Object Orientation in Operating Systems*, pp. 9–15, 1991.

[17] M. Banâtre, Y. Belhamissi, V. Issarny, I. Puaut, and J. P. Routeau, "Adaptive placement of method executions within a customizable distributed object-based system: Design, implementation and performance," in *Proceedings of the Fifteenth International Conference on Distributed Computing Systems*, pp. 279–286, 1995.

[18] M. A. Hiltunen and R. D. Schlichting, "Constructing a configurable group RPC service," in *Proceedings of the Fifteenth International Conference on Distributed Computing Systems*, pp. 288–295, 1995.

[19] D. C. Sturman and G. A. Agha, "A protocol description language for customizing failure semantics," in *Proceedings of the Thirteenth Symposium on Reliable Distributed Systems*, pp. 148–157, 1994.

[20] J. R. Callahan and J. M. Purtilo, "A packaging system for heterogenous execution environments," *IEEE Transactions on Software Engineering*, vol. 17, no. 6, pp. 626–635, 1991.

[21] V. Issarny and C. Bidan, "Aster: A CORBA-based software interconnection system supporting distributed system customization," in *Proceedings of the Third International Conference on Configurable Distributed Systems*, 1996. To appear.

[22] OMG, "The Common Object Request Broker: Architecture and Specification – Revision 2.0," tech. rep., OMG Document, 1995. Available *via* anonymous ftp at `omg.org`.

[23] OMG, "Object Management Architecture Guide (OMA Guide)," Tech. Rep. 92.11.1, OMG, 1992. Available *via* anonymous ftp at `omg.org`.