A Programming System for the Development of TINA Services

Titos SARIDAKIS, Christophe BIDAN, Valérie ISSARNY
IRISA/INRIA
IRISA – Campus de Beaulieu, 35042 Rennes Cédex, FRANCE.

Abstract

Programming environments for the development of distributed applications are called to face issues such as the configuration of the interconnection system, the coordination of heterogeneous application constituents, etc. Aster is a configuration-based system which deals with those issues and allows the programmer to guide the customization of the runtime environment so as to meet the application needs. In this paper we show that our system conforms with TINA's computing architecture and we demonstrate its capabilities on formal reasoning about functional and nonfunctional application requirements. An example of trading both within the same runtime environment and between heterogeneous ones is given. In this example we demonstrate the automatic customization capabilities of Aster and we illustrate a unified way for dealing with the issues of software heterogeneity, software reuse, and interoperability.

Keywords

Configuration-based programming, Customization, Distributed processing environment, Interoperability, Trading

1 INTRODUCTION

The evolution of distributed systems and their ever increasing requirements have led to the need for standard software architectures for distributed computing in order to capture, comprehend, and model their aspects. International standardization work is summarized in the Reference Model of Open Distributed Processing (RM-ODP) (ISO/IEC 1994), which aims at defining a generic model to cover all aspects of distributed processing systems. At the same time, industrial consortia have been specifying distributed software architectures which can be seen as specialized instances of ODP with respect to the type of products on which they focus. Such efforts include OMG's Common Object Request Broker Architecture (CORBA) (OMG 1995) and the Telecommunication Intelligent Network Architecture (TINA) from the TINA Consortium (TINA-C) (TINA-C 1995-a). TINA provides an architecture for distributed telecommunication systems, which deals with the various application requirements for Quality of Service (QoS) and the interoperation of telecommunication services developed for heterogeneous distributed platforms. While there exist implementations complying with CORBA specifications, providing an environment completely conforming to TINA specifications remains an open issue. In this paper we show that Aster is candidate for the development of TINA services.

Aster is a *configuration-based* programming environment for distributed applications (Issarny and Bidan 1996-a). Its runtime system consists of the application modules requiring some execution properties and interconnected by a *software bus* (Purtilo 1994), which is *customized* to meet application requirements. The software bus consists of a *base-bus* providing communication primitives and a set of system level components offering the set of execution properties that meet application requirements. Like other configuration-based systems, Aster is independent of the module programming languages, supports heterogeneous base-systems (i.e. hardware platforms and operating systems), offers flexibility in module interconnections, and supports dynamic changes during execution to allow system evolution. Moreover, Aster supports automatic customization of the runtime system with respect to application requirements, eases QoS management and promotes software reuse.

In this paper we use Aster to configure telecommunication services having different QoS requirements and executing on various platforms. We demonstrate that our system closely approximates the requirements of TINA's computing architecture and we focus on the correspondence of Aster constituents with the constituents of the TINA computing architecture (§2). Moreover, we argue that Aster offers a simple and uniform hierarchical model for interconnecting application modules and we demonstrate how it can supports interoperability requirements by an example of a customized software bus that provides trading services both within the same platform and between heterogeneous environments (§3). Finally, we summarize the paper, refer to related work and present the current status of our work (§4).

2 DEVELOPING TINA SERVICES

The TINA architecture consists of the *network, management, service,* and *computing* architectures. Computing architecture defines the modeling concepts from the engineering and computational viewpoints. From the latter viewpoint, a telecommunication application is an ensemble of interacting software entities called *computational objects*, which are described in the TINA Object Definition Language (ODL) (TINA-C 1995-b). The engineering viewpoint introduces the abstract infrastructure of TINA's Distributed Processing Environments (DPEs) (TINA-C 1994-a), which are distributed platforms conforming to TINA, and describes how to deploy computational objects on the DPEs.

2.1 Relating Aster with TINA

Although TINA specifications and requirements are well known in the area of distributed programming, there does not exist a single distributed programming system completely conforming with them. This fact, combined with interoperability and reusability issues, presents a twofold challenge: to build TINA services by using a system that closely conforms to TINA specifications and to provide runtime system customization that supports communication among DPEs on different platforms with matching behaviors (i.e. *federation* requirements (TINA-C 1994-a)).

Computational Modeling Concepts. The Aster configuration language is used to declare application components. An application component can be of type: `i)` *interface*, which describes the services required and provided by a module, `ii)` *implementation*, which associates interfaces to module implementations, or `iii)` *hierarchical*, which describes the construction of a module in terms of bindings among the operations provided and required by a set of existing components (interface, implementation or hierarchical). Interface and implementation components correspond to ODL's *interface* and *object* templates respectively. Hierarchical components do not have a direct correspondence in ODL, although they are close to TINA's *building blocks*, an architectural structure which aggregates objects, and controls the availability of *contracts* (Natarajan 1995). Building blocks also provide the means to manage and configure their constituent objects as a single, centralized entity. From that perspective, hierarchical components are more general since they can satisfy application requirements for both centralized and distributed properties. Aster allows to declare application specific requirements concerning both functional (i.e., distribution transparencies and object life cycle) and nonfunctional (i.e., DPE properties and QoS) properties. Application requirements in ODL are defined by means of data types whereas in Aster we associate a formula of the first order predicate calculus to each requirement. This provides a way to reason formally about matching of component requirements with properties and QoS offered by the underlying system.

Engineering Modeling Concepts. A base-bus in Aster provides the primitives required for distributed execution and communication, which roughly correspond to fundamental DPE constraints (Kelly *et al.* 1995). It does not have an exact counterpart in TINA, although it is close to the notion of DPE *kernel* (see figure 1). The latter provides a supporting infrastructure for the QoS related to Communication, Concurrency, Availability, Real time in terms of RPC timeouts, bandwidth reservation, communication protocols and end to end latency guarantees respectively (TINA-C 1994-b). The system components used to build customized software buses over the base-bus, can be seen as TINA objects implementing DPE functions. The Aster programmer,

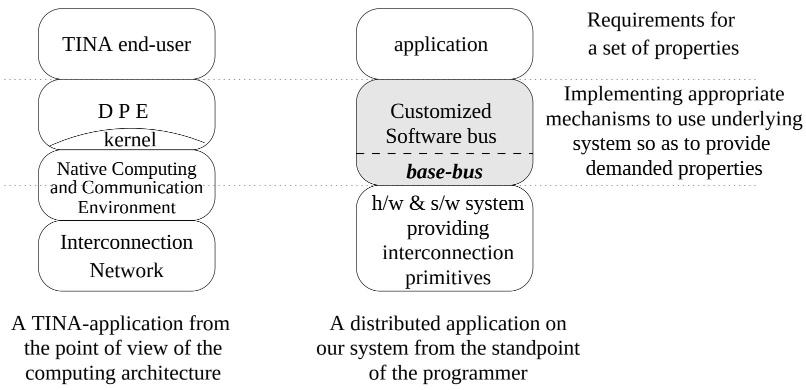

Figure 1 Correspondence between TINA computing architecture and the components of Aster.

instead of explicitly specifying module interconnections that yield a system satisfying application requirements, only declares application requirements in application module interfaces. Based on the requirements declared in module interfaces, appropriate system components are *automatically* selected from the system repository and used in the construction of the software bus which meets application needs. By selecting existing software from the system repository, Aster promotes software reuse and minimizes programmer's effort to build the interconnection environment.

2.2 Functional and Nonfunctional Requirements

In TINA, the notions of *application* and *platform profiles* are used to capture the actual requirements (functional and nonfunctional) on the application infrastructure and the set of models and services supported by a given DPE respectively. The programmer has to draw the application profile and to look for a platform with a matching profile. In Aster, the selection of a platform according to a given application profile, is automatic and relies on a formal method. Formal methods have been used for verifying semantic correctness of an application with respect to application modules (Zaremski and Wing 1995), and the verification process has been automated using the Larch prover (Guttag and Horning 1993). However, such approaches cannot be used directly for *profile matching*, i.e. to verify that application constraints are met by a given runtime system. Aster introduces a formal method which relies on the definition of profiles in terms of execution properties and is used for profile matching, and to automatically build customized software buses from available base-buses and software components.

Formal Definition of Profiles. We define profiles in terms of execution properties corresponding to different distribution management semantics. Execution properties are stored in a structure called *property tree*, where each node is identified by a name and specifies a set of properties, and each leaf specifies a set of distribution management semantics in terms of execution properties, which are ordered according to the *stronger-than* relation. Each execution property is identified by a name and has an associated formula of first order predicate calculus. For example, the RPC failure semantics are found in the following leaf:

$$(\text{exactly\_once}, \mathcal{F}_1) \prec (\text{at\_most\_once}, \mathcal{F}_2) \prec (\text{best\_effort}, \mathcal{F}_3)$$

where $\mathcal{F}_i$ is the formula associated to the corresponding execution property and $\prec$ denotes the *stronger-than* relation. Figure 2 presents the property subtree for distribution management semantics related to RPCs (Huang and Ravishankar 1994). In §3 we exemplify the use of formulas related to specifications of automatic trading requirements.

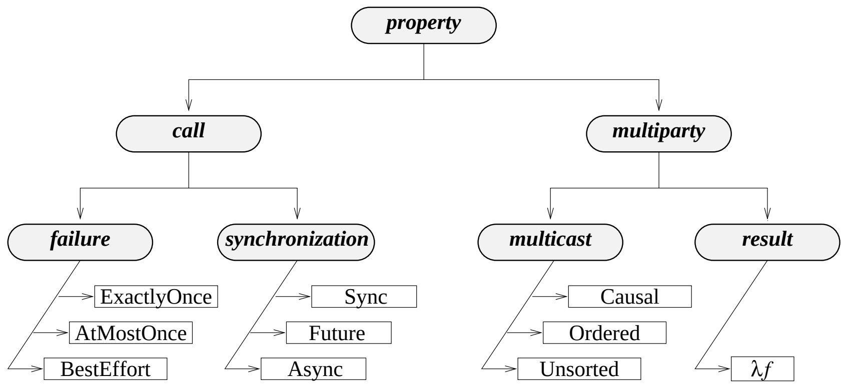

Figure 2 The property subtree related to RPC distribution management semantics.

Profile Matching. A profile corresponds to a set of behaviors, where each behavior is the conjunction of a set of properties that do not belong to the same leaf. For $\mathcal{P}_1$ and $\mathcal{P}_2$ being two profiles, we define the *plug-in* profile matching between them when $\mathcal{P}_1$ guarantees *at least* all $\mathcal{P}_2$ behaviors (i.e. if for every $\mathcal{P}_2$ behavior there exists a $\mathcal{P}_1$ behavior to satisfy it). More formally:

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

where P_i^j is a behavior declared by profile $\mathcal{P}_i$. Notice that $\lhd_{plug-in}$ corresponds to a formal interpretation of the customization notion: for $\mathcal{R}_A$ being an application profile, and $\mathcal{P}_A$ the behaviors provided by application components, a software bus with platform profile $\mathcal{P}_{bus}$ is customized to meet application needs *if and only if:* $(\mathcal{P}_{bus} \cup \mathcal{P}_A) \lhd_{plug-in} \mathcal{R}_A$

Customized Software Buses. The execution properties declared by system components do not necessarily belong to the property tree. They can be formulas whose conjunction with execution properties declared in other system components leads to a property in the tree. We can enrich the behaviors of a given software bus B with the execution properties implemented by a set of system components $\{S_i, i = 1..n\}$. The behaviors provided by the resulting software bus consist of the logical *and*s of each B behavior with all the properties provided by every S_i. To denote this enrichment we define operator $\bowtie$ as:

$$\mathcal{P}_B \bowtie \bigcup_{i=1..n} P_{S_i} \quad \equiv \quad \bigcup_{\forall P \in \mathcal{P}_B} \left(P \wedge \bigwedge_{i=1..n, j=1..m} P_{S_{i.j}}\right)$$

where $\mathcal{P}_B$ denotes B's profile, P_{S_i} denotes S_i's properties, and $P_{S_{i.j}}$ denotes S_i's *j-th* property. By using $\bowtie$, we can express in terms of $\lhd_{plug-in}$ the construction of a customized software bus, which guarantees to *at least* satisfy application requirements. For $\mathcal{S}_E$ being a set of system components each having properties P_{S_i} and requirements $\mathcal{R}_{S_i}$, $\mathcal{B}$ being a set of software buses each having requirements $\mathcal{R}_B$ and profile $\mathcal{P}_B$, $\mathcal{R}_A$ being the application profile, and $\mathcal{P}_A$ the application behaviors, a software bus $bus(A)$ customized to application need is defined as:

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

3 DPE CUSTOMIZATION

A binding between a module exporting an operation and another one importing it is explicitly declared by the programmer if the instance of the module exporting the operation is known at the implementation stage. In the case where the instance of a module exporting an operation is not known at the implementation stage, it is possible to perform a *dynamic* binding, i.e. to postpone the binding until execution time. To deal with dynamic binding, the notion of *trader* or *trading service* has been introduced by various distributed software architectures including TINA and OMG's OMA (OMG 1992). A trader provides the means for locating a module in an executing application according to some key (e.g. interface type).

3.1 Intra-domain Trading

To reason about automatic trading, we introduce the *trading* property which states that a request not specifying the destination component shall be served by some component matching its constraints. For C being a software component requiring a service specified by an interface type I_C and conforming to constraints S_C, and $\mathcal{RS}$ being the set of servers registered in the execution environment, we define:

$$\begin{aligned} trading(\mathcal{RS}, C, I_C, S_C) \equiv \quad & send(C, null, \langle I_C, S_C, msg\rangle) \land \\ & (\forall S \in \mathcal{RS} \mid I(S) = I_C \land S_C(S) \colon receive(C, S, msg)) \end{aligned}$$

where *send* and *receive* correspond to predicates expressing that the **send()** and **receive()** operations respectively, were performed. The execution property expressing dynamic binding is now direct. For $\mathcal{C}$, $\mathcal{I}$, and $\mathcal{S}$ being the set of software components, interfaces, and possible constraints respectively, we define:

$$dynamic(\mathcal{RS}) \equiv \forall C \in \mathcal{C}, \forall I \in \mathcal{I}, \forall S \in \mathcal{S} : trading(\mathcal{RS}, C, I, S)$$

The Trader. The system component providing the *dynamic* property is called *trader*. The interconnections with the base-bus are described in its interface and are used to guide the automatic construction of a software bus customized to provide dynamic binding. The interface declaration conforming to TINA trading service specifications (TINA-C 1994-a), is given in figure 3. In general, the *trader* can be based on a number of other services, e.g. in OMG's OMA (OMG 1995) it is based on Object Interface Repository Service, Object Name Service, Object Query Service, and Object Security Service. For simplicity, we present the *trader* as a simple component omitting its complex nature. Furthermore, notice that dynamic trading concerns the set of registered *local* servers, in contrast to *global* trading discussed in §3.2.

Example. To illustrate dynamic trading, we use the example of a nationwide institution with branches in different cities, where each branch stores technical documents in a local electronic library. Researchers have access to *all* on-line libraries, and they can search for documents according to some keywords without explicitly determining the address of library where the search should be effectuated. This document distribution transparency implies a software bus customized to provide dynamic binding. The customization is achieved by introducing the *trader* which undertakes the interactions with all registered branch libraries, to determine the receivers of the client request. Moreover, if we assume that the institution offers part of its documents to the public, then we need a software bus further customized to provide au-

```
system interface I-trader {
    void export (in pid service, interface intServ, attribute attServ);
    void withdraw (in pid service);
    void modify (in pid service, in attribute mode);
    void search (in interface intServ, attribute attServ; out setpid setServices);
    void select (in interface intServ, attribute attServ; out pid service);
  provides dynamic (local);
  interconnection
    dynamic(local): { (client, send(C,∅,⟨I,S,msg⟩)) →
                      {select(I,S,srv); send(C,srv,msg)} };
}
```

Figure 3 The trader interface.

thentication and authorization in order to conform with the institution access policies.

3.2 Interoperability Issues

Dynamic trading allows a client to interact with appropriate servers by only specifying desired properties (e.g. interface type). This suffices for module interactions in DPEs with identical profiles that are also implemented by the same software system. However, a TINA application is expected to be executed on a set of DPEs implemented by different software systems. In order to achieve interoperability, we should allow trading in a *federation* of heterogeneous execution platforms. This enables negotiations for the sharing of services without losing control of local management policies. Assuming the existence of a trading service within each of the federated domains, *global* trading depends on the interoperation between the domain platforms, and the compatibility of domains in terms of management policies related to functional and non-functional requirements. The former aspect is implementation dependent and relies on the provision of gateways that make appropriate translations (e.g. between data type formats), and the latter is semantics-dependent and relies on the compatibility of the different platform profiles. In other words, two domains are allowed to interoperate if their platform profiles are equivalent. This strong requirement can be relaxed by defining alternative predicates which express some weaker, application specific compatibility. For D_1 and D_2 being two domains with platform profiles $\mathcal{P}_{D_1}$ and $\mathcal{P}_{D_2}$ respectively, we define *compatibility* as:

$$compatibility(D_1, D_2) \equiv (\mathcal{P}_{D_1} \lhd_{plug-in} \mathcal{P}_{D_2}) \wedge (\mathcal{P}_{D_2} \lhd_{plug-in} \mathcal{P}_{D_1})$$

Given the *compatibility* predicate, we can derive a formula expressing whether it is possible for a given domain to enter in a set of federated domains. For

$\mathcal{D}$ being a set of federated domains and D_1 being a domain to be inserted in the federation, we define the *federation* property as:

$$federation(D_1) \equiv \forall D_i \in \mathcal{D} : compatibility(D_1, D_i)$$

The Global Trader. Similar to the intra-domain case, we introduce a component that provides an inter-domain trading service and is based on the *federation* property and the *I-trader* interface. The difference of this *global* trader with the *local* trader introduced previously is this trader can contact components in the global set, which is the set of all servers registered in each DPE of the federation. The global trader inherits from *I-trader* and requires the *federation* property for interacting with servers in other DPEs (figure 4a). The *federation* property is provided by a hierarchical component consisting of a system component verifying profile matching, and another providing data type translation (figure 4b).

```
interface I-Gtrader : I-trader {
  provides dynamic(global);
  requires federation;
  interconnection
    dynamic(global): {
      (client, send(C,∅,⟨I,S,msg⟩)) →
          {select(I,S; srv);
           send(C,srv,msg)}
    }
}
```

(a)

```
system interface I-Gaccess {
    void inter_send(in pid sender,
        in interface srv, in data msg);
  provides federation;
  interconnection
   federation: {
    (client, send(C,S,msg)) →
       { if S ∈ local(𝒮)
          then send(C,S,msg)
          else inter_send(C,S,msg)}
    }
}
```

(b)

Figure 4 Interfaces related to global trading.

According to *federation* specifications, every call requiring dynamic binding is handled both in the local and the remote DPEs. The inter_send operation first confirms profile matching of the remote DPE and then forwards the request to the component responsible for data types translation which finally dispatches it in a format recognized by the remote DPE.

Example. To illustrate global trading, we extend the previous example to include the case where the distributed library is called to interoperate with distributed libraries provided by other institutions. Each institution has a different DPE for managing library accesses and enforcing internal policies. The goal is to create a federation of those DPEs in order to construct an inter-institutional distributed library. To achieve this, we further customize the software bus implementing each DPE, as shown in figure 5. In this figure,

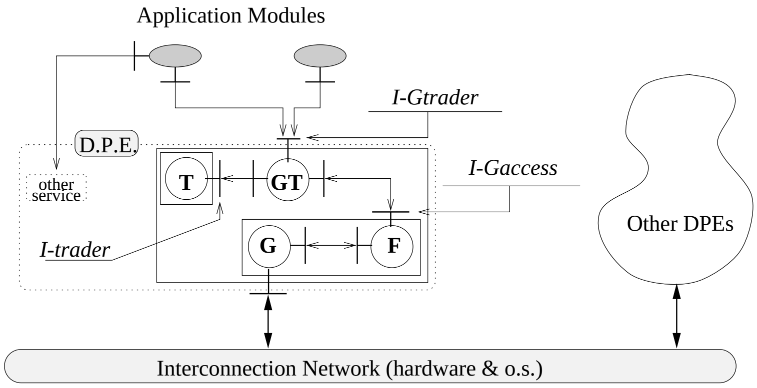

Figure 5 Interconnections among the elements implementing global trading.

the component GT integrates local and remote trading by forwarding client requests both to the trader component T for performing local trading, and to other DPEs for remote trading. Remote DPEs are accessed through component F that provides *federation* and component G that provides data type translation. The role of F and G is not restricted in assisting global trading. Like the trader component, they can also be used by other services requiring inter-DPE communication. Notice that we do not elaborate on the data format used during inter-DPE communication since it belongs to communication modeling field and there is a number of existing communication protocols which can serve for this purpose.

4 CONCLUSIONS

We demonstrated that Aster matches the specifications of the TINA computing architecture and constitutes a candidate for implementing TINA's computational and engineering modeling concepts. To illustrate customization, we used an example of trading, i.e. the need to dynamically interconnect application modules during execution time, according to application requirements. We showed that customized software buses provide all the functionalities expected from a DPE and also cover a number of aspects from the *Native Computing and Communication Environment* (NCCE), including DPE interoperability issues. Hence, Aster allows the programmer to specify explicit intra-communication policies as much as strict strategies for interaction with the external world. Application module declarations describe in terms of required execution properties, the rules governing module interactions. Based on those declarations, Aster selects from the system repository a set of system components and a base bus, and interconnects them to build an execution system customized to application needs.

Related Work. Based on the relevance of TINA with distributed systems, we expect that a number of existing distributed programming systems will be qualified to implement the TINA computational and engineering modeling concepts. Systems like Polylith (Purtilo 1994) and Conic (Magee *et al.* 1989) will evidently influence the way that software evolution will be accommodated by the DPEs. Capabilities like those offered by Durra (Barbacci *et al.* 1993), are expected to be used in a variety of telecommunication software with dynamic interaction nature. Also, systems like UniCon (Shaw *et al.* 1995) which propose a very rich model for specifying the interactions of the application and the structure interconnection environment, will probably be adopted for describing sophisticated telecommunication applications. Experience gained by efforts like ACL (Singh and Gisi 1996) is expected to help in the area of coordinating heterogeneous modules in a real-time system. Finally, CORBA is expected to provide a standard platform for deployment and interaction of distributed components (Kitson 1995). Advantages of our system over other candidates, are *(i)* the automatic construction of the software bus from existing system components, customized to meet application requirements, *(ii)* the formal description of properties that application requires from the interconnection system, and *(iii)* the straightforward way of integrating interoperability with trading and hence allowing the transparent interaction among application modules that may reside in the same, or heterogeneous DPEs.

Current Status. Work on Aster is ongoing both on a theoretical and on a practical level. From the former aspect, we are studying the expansion of the property tree to accommodate a number of QoS which are characterized indispensable for today's distributed computing. We especially emphasize on security and availability in terms of fault tolerance and timeliness. From the practical aspect, we work on the first implementation of our system which is an extension of CORBA (Issarny and Bidan 1996-b). The Aster configuration language is implemented as an extension of the OMG's Interface Definition Language (IDL) and the base-bus complies with base communication mechanisms of CORBA (ORB-core) plus the *Static Invocation Interface* mechanism.

REFERENCES

Barbacci M. *et al.* (1993) Durra: a Structure Description Language for Developing Distributed Applications. *Software Engineering Journal*

Guttag J. V. and Horning J. J. (1993) Larch: Languages and Tools for Formal Specifications. *Text and Monographs in Computer Science*, Springer-Verlag

Huang Y. M. and Ravishankar C. V. (1994) Designing an Agent Synthesis System for Cross-RPC Communication. *IEEE Transactions on Software Engineering* 20(3)

ISO/IEC (1994) Reference Model of Open Distributed Processing. No. 10746

Issarny V. and Bidan C. (1996-a) Aster: A Framework for Sound Customization of Distributed Runtime Systems. *Proceedings of the 16th International Conference on Distributed Computing Systems*

Issarny V., Bidan C. (1996-b) Aster: A CORBA-based Software Interconnection System Supporting Distributed System Customization. *Proceedings of the International Conference on Configurable Distributed Systems*

Kelly E. *et al.* (1995) TINA DPE Architectures and Tools. *Proceedings of TINA'95*

Kitson B. (1995) CORBA and TINA: The Architectural Relationship. *Proceedings of TINA'95*

Magee J. *et al.* (1989) Constructing Distributed Systems in Conic. *IEEE Transactions on Software Engineering*, 15(6)

Natarajan N. (1995) INAsoft DPE: A Platform for Distributed Telecommunications Applications. *Proceedings of TINA'95*

OMG (1992) Object Management Architecture Guide (OMA Guide).

OMG (1995) The Common Object Request Broker: Architecture and Specification - Version 2.0.

Purtilo J. (1994) The Polylith Software Bus. *ACM Transactions on Programming Languages and Systems*, 16(1)

TINA-C (1994-a) Engineering Modeling Concepts (DPE Architecture). TB_NS.005_2.0_94

TINA-C (1994-b) Engineering Modeling Concepts (DPE Kernel Specifications) – Version 1.0. TR_KMK.001_1.1_94

TINA-C (1995-a) Overall Concepts and Principles of TINA. TB_MDC.018_1.0_94

TINA-C (1995-b) TINA Object Definition Language (TINA-ODL) Manual – Version 1.3. TR_NM.002_1.3_95

Shaw M. *et al.* (1995) Abstractions for Software Architectures and Tools to Support Them. *IEEE Transactions on Software Engineering*, 21(4)

Singh N. and Gisi M. A. (1996) Coordinating Distributed Objects with Declarative Interfaces. *Proceedings of the COORDINATION'96*, Lectures in Computer Science No.1061, Springer - Verlag

Zaremski A. M. and Wing J. M. (1995) Specification Matching of Software Components. *Proceedings of the ACM SIGSOFT'95*