Component-Based Programming of Distributed Applications

Valérie Issarny[1], Luc Bellissard[2], Michel Riveill[2], and Apostolos Zarras[1]

[1] Inria, Irisa, Campus de Beaulieu, 35042 Rennes Cédex, France
[2] Imag-Inria, Inria, 655 Avenue de l'Europe, 38330 Montbonnot, France

Abstract. The software architecture research domain arose in the early 90s and seeks solutions for easing the development of large, complex, software systems based on the abstract description of their software architectures. This research field is quite recent and there still does not exist a consensus on what should be the description of a software architecture. However, guidelines are already provided. In particular, it is now accepted that an architecture definition decomposes into three types of elements: component, connector, and configuration, which respectively correspond to a computation unit, an interaction unit and an architecture. It is also admitted that the description of an architecture should rely on a well-defined set of notations, generically referred to as architecture description languages. This document gives an overview of the capabilities offered by development environments based on the architecture paradigm. In a first step, we examine basic features of architecture description languages, which may be seen as their common denominator although existing languages already differ from that standpoint. We then concentrate on two specific environments, developed by members of the Broadcast working group, which aim at easing the implementation of distributed applications out of existing components. The Aster environment from the Solidor group at Inria-Irisa provides means for the systematic synthesis of middleware from non-functional requirements of applications. The Olan environment from the Sirac group at Inria-Grenoble offers support for the deployment of distributed applications composed of heterogeneous software elements.

1 Introduction

The ever increasing complexity of distributed applications calls for methods and tools for easing their development. In that framework, industrial consortia have emerged so as to provide application developers with standard distributed software architectures. In general, such a standard specifies a base distributed system for communication management, a set of services for distribution management (e.g., naming service), and a set of tools for application development (e.g., Interface definition language). The definition of a standard architecture then serves as a guideline for the implementation of a programming system. A well known example of standard distributed architecture is the Oma (*Object Management*

S. Krakowiak, S.K. Shrivastava (Eds.): Distributed Systems, LNCS 1752, pp. 327–353, 2000.
© Springer-Verlag Berlin Heidelberg 2000

Architecture) [33] from the OMG (*Object Management Group*) for the development of distributed applications relying on client-server type interactions.

A more ambitious approach to the development of complex distributed applications is the one undertaken in the software architecture research field of software engineering. Instead of concentrating on the definition of a specific architecture, the ongoing research work aims at providing a sound basis for the specification of various styles of software architectures (e.g., see [35,40]). An architectural style identifies the set of patterns that should be followed by the system organization, that is, the kinds of components to be used and the way they interact. Although the software architecture field is continuously evolving, it is now accepted that the description of an application architecture decomposes into at least three abstractions (e.g., see [39]):

(*i*) *Components* that abstractly define computational units written in any programming language,
(*ii*) *Connectors* that abstractly define types of interactions (e.g., pipe, client-server) between components,
(*iii*) *Configuration* that defines an application structure (i.e., a software architecture or configuration-based software) in terms of the interconnection of components through connectors.

Development environments that are based on the software architecture paradigm then integrate an *Architecture Description Language* (ADL) that allows application specification in terms of the three above abstractions, together with runtime libraries that implement base system services (including primitive connectors). Such an application description fosters software reuse, evolution, analysis and management.

In the light of research results in the software architecture field, configuration-based description of distributed applications constitutes a promising approach for facilitating the development of correct, complex distributed applications. The next section provides a general definition of ADLs together with an overview of research work in the area. Sections 3 and 4 then concentrate on two specific development environments based on the architecture description of distributed applications. Section 3 presents the Aster environment, which is being developed within the Solidor research group at INRIA-IRISA. The Aster environment offers a set of tools for the systematic synthesis of middleware from the formal specification of non-functional properties (e.g., security, availability, timeliness) as respectively required by distributed applications and provided by available middleware services. Section 4 discusses the Olan environment from the Sirac research group at INRIA-Grenoble. The Olan environment provides means to deploy a distributed application over a given infrastructure with respect to the application's configuration. Finally, Section 5 offers some conclusions.

2 Description of Software Architectures

Research effort in the software architecture domain aims at reducing costs of developing large, complex software systems. Towards that goal, formal notations

are being provided to describe software architectures, replacing the usual informal description of software architectures in terms of box-and-line diagrams. These notations are generically referred to as *Architecture Description Languages* (ADLs). Basically, an ADL allows the developer to describe the gross organization of his system in terms of coarse-grained architectural elements, abstracting away the elements' implementation details. Except this general definition and despite the increasing interest in the software architecture domain since its appearance in the early 90s, it still does not exist a consensus on what is an ADL. As previously mentioned, prominent elements of a software architecture subdivide into the three following categories: component, connector and configuration. However, some ADLs do not model connectors as first-class objects in which case connectors are implicitly defined within configurations through the connections (or bindings) among components. In general, existing ADLs differ depending on the aspects that are targeted for the construction of software architectures. We identify at least two research directions, which are sometimes both covered by the same ADL:

(*i*) Architecture analysis that relates to the formal specification of architecture behavior. Work in this category further subdivides into the analysis of the properties provided by either components, connectors, or configurations.
(*ii*) Architecture implementation that relates to the implementation of an application from the description of its architecture.

The interested reader is referred to Chapter 2 of [1] and [30] for a survey of existing ADLs and associated CASE tools. An overview of work on the formal specification of the behavior of architecture elements may further be found in [19]

In general, most existing ADLs should rather be seen as complementary rather than as competitive. As a consequence, this has led to the definition of an architecture description interchange language so as to allow the developer to combine the various facilities provided by different ADLs, for the construction of his applications [17]. The following provides a general definition of ADLs, based on existing work in the area, together with a brief overview of work relating to the implementation of applications from their architectural description. However, we do not intend to be exhaustive from the standpoint of ADLs taken as references since the number of proposed ADLs is quite large and the number of related projects keeps growing (see [15] for an overview of latest results). Examples specifically considered in the following are Aesop [16], Aster [20], Darwin [27], Olan [6], Rapide [26], Sadl [32], UniCon [39], and Wright [2].

The following paragraphs give a general definition of component, connector and configuration as offered by existing ADLs knowing that the corresponding language constructs vary according to ADLs. For illustration purpose, we take the example of a Distributed Information System (DIS). However, simplified declarations will be provided, which are sufficient to exemplify ADL features. Basically, a DIS is composed of a set of clients interacting with a possibly distributed server for information access; we further assume that the interaction

protocol among components is RPC-like. Thus, the DIS may be seen as a special instance of a client-server architecture and corresponds to various specific architectures depending on the instantiation of the DIS architectural elements. For instance, examples of DIS incarnations include the Web, distributed file systems, and online services providing access to discrete data.

2.1 Component

A component may be either primitive or complex: a complex component is equivalent to a configuration; and a primitive component corresponds to either a computation unit or a data store, whose implementation details (e.g. programming language, supporting platform, ...) are abstracted away. The description of a primitive component gives the component interface. In general, the interface specifies the interaction points (e.g. port in UniCon) of the component with respect to the communication protocol that is used (i.e. it prescribes the expected type of connector).

The other feature of a component that is of equal interest is its functional behavior. Here, the component's interface states the list of operations (or services) provided for other components and the list of operations required from other components. Let us remark that the latter definition of component interface may be seen as an extension of an IDL (Interface Definition Language) interface. Even closer to this ADL definition of component falls *Module Interconnection Languages* (MILS) that were introduced in the 70s for the implementation of large-scale software [14]. Simply stated, a MIL allows the developer to abstractly describe the configuration of his application through bindings among the functions offered and provided by the components. Hence, when the ADL definition is oriented towards application implementation such as UniCon (as opposed to ADLs aimed at architecture analysis), it appears to closely resemble a MIL except it integrates the notion of connector. Let us further remark that this distinction falls short when a development environment that is based on a MIL enables to build applications that may run above various (possibly distributed) platforms. For instance, the Polylith environment [36] belongs to this category of environments. The distinction that can then be made between a MIL-based and an ADL-based environment for application deployment above a given platform is that the know-how about the platform usage is within the corresponding connector for the former, while the latter requires wrapping the platform so as to make it accessible through the API specified by the environment for use by components. Hence, the connector notion enables the development of adequate CASE tools for the generation of interfacing code among components and the underlying platforms as abstracted by connectors. Such a facility is for instance supported by the Aster and Olan environments, as further addressed in the two next sections. On the other hand, to our knowledge, the wrapping of platforms within MIL-based environments is realized in an ad'hoc manner, on a case-by-case basis. In the following, despite the aforementioned distinction between MIL and ADL-based development, when concerned with implementation-oriented ADLs, we include

MILs in this category. To be more precise, implementation-oriented ADLs and MILs may be referred to under the generic term of *configuration languages* [10].

The definition of component interfaces –be they interaction-, functional-related, or both– may simply take the form of operation signatures or be more precise by formally specifying the behavior of operations [34,13]. Other attributes may further be stated in the declaration of components. For instance, when concerned with architecture implementation, the ADL provides way to specify the implementation file that corresponds to the component.

For illustration, Figure 1 gives a description of the client component of the DIS example. For the sake of generality, we do not take the syntax of a specific ADL but instead use a self-explanatory syntax, and merge the component's interaction- and functional-related specification. The *client* component declaration specifies a component interface, which may be instantiated through different implementations.

```
component interface client =
   port
      Declares the port used for interaction, e.g., client-type port
   functional
      Declares the operations offered/required by the client component,
      e.g.: operations for opening, closing a connection,
      for reading, writing information
   interaction
      Declares the port used for accessing the declared operations,
      which prescribes whether they are offered or provided by the
      component. In the example operations given above, they are all
      required and use a client-type port.
```

Fig. 1. An example of component description

In general, software reuse being one of the primary objectives of ADL-based development environments [18], ADLs can support reuse by modeling abstract components as types and instantiating them multiple times in an architecture description [30]. Enhanced support for software reuse may further be offered through subtyping and parameterized types. All the ADLs taken into consideration for this chapter distinguish component types from instances. On the other hand, not all offer subtyping and parameterization: the former is supported by Acme, Aesop, Aster, and Rapide, and the latter is supported by Acme, Darwin, and Rapide.

2.2 Connector

Similarly to a component, a connector may be either primitive or complex. A primitive connector corresponds to a communication protocol of the target execution platform. For instance, over a Unix platform, there will be a connector describing a pipe. The description of a primitive connector includes the connector interface, which may further formally specify the connector's behavior [2]. In addition, if the architecture description is to be used for the architecture's implementation, the implementation corresponding to the connector is given.

A complex connector is a connector that is built from a set of connectors and components. A typical example of complex connector is a middleware that comprises a set of services for enhanced management of component interoperation [7]. As a more precise example, we may consider a CORBA Distributed Processing Environment (DPE) composed of some Common Object Services (COSs) interconnected by the CORBA Object Request Broker (ORB). Thus, the properties provided by a connector relate to: (*i*) properties of the base underlying communication protocol (e.g. asynchronous message passing, RPC, pipe, ...), and (*ii*) to additional non-functional properties (e.g. dependability, security, timeliness, ...) characterizing the embedded services [21].

For illustration, Figure 2 gives the definition of the connector describing an RPC-based transactional middleware providing atomicity, and isolation properties, as for instance offered by the CORBA ORB combined with the Object Transaction Service (OTS) and Concurrency Control Service (CCS).

```
connector interface TransactionalServiceMdw =
   role
      Declares the roles offered for achieving interaction among
      components, e.g., client-type and server-type roles as
      offered by RPC-based middleware
   non-functional
      Specifies the additional operations provided by the middleware
      for the enforcement of some non-functional (or quality)
      properties over the interactions, e.g., operations for creating,
      committing, and aborting a transaction in the case of the
      considered middleware
```

Fig. 2. An example of connector description

As for the definition of components, enhanced reuse is provided by the ADL if connectors are modeled as types. However, let us recall here that not all the ADLs define connectors as first-class entities, in which case connectors are defined in-line within the configuration, hence prohibiting connector naming, subtyping

or reuse in general. Among the ADLs taken as examples, Acme, Olan, Sadl, UniCon, Wright and the latest Aster version model connectors explicitly, which is realized according to one of the following two forms: (*i*) definition of an extensible type system in terms of communication protocols and independent of implementation (e.g. Acme, Aesop, Wright); (*ii*) definition of a set of types based on their implementation mechanisms (e.g. Aster, Olan, Sadl, UniCon).

2.3 Configuration

The description of a configuration consists in interconnecting a set of components so as to bind the operations that are required by some configuration components to the corresponding operations that are provided by other components of the configuration. These interconnections (or bindings) are further realized through connectors, hence specifying the communication protocols that are used for the resulting interactions among components. As for components and connectors, formal specification of configuration behavior has given rise to a number of proposals, providing ways to reason about correctness of architecture refinement [32] and about legal dynamic changes to architectures [25], and to carry out behavioral analysis of architecture properties [26,2,24].

For illustration, Figure 3 gives the definition of a configuration for the DIS example where the definition of the client component that is used is the one given previously.

A configuration ultimately corresponds to the abstract description of an application. However, as exemplified by the DIS configuration, it generally defines a software architecture from which various applications (or more specialized configurations) may be derived by refining the description of embedded components and connectors. Such a feature is crucial from the standpoint of both design and software reuse. A configuration constitutes the blueprint for the implementation of a specific application. The choice for a given blueprint then results from various factors such as the functionalities targeted for the application or the execution platform to be used for execution.

2.4 Implementation of Applications from Architecture Description

Implementation of applications from their architectural description is one area of active research in the software architecture domain. Basically, the implementation of an application from its architectural description consists of generating appropriate stub code for the realization of component interactions *via* connectors (*e.g.* see [12] for an overview although this reference addresses application implementation from its description using a MIL). One of the most difficult parts in the implementation of an application from the description of its software architecture lies in the integration of possibly heterogeneous architectural elements, which may be partly simplified through the use of component-based middleware architectures such as CORBA or EJB. This issue is being examined by various research groups of the field. It is currently simplified in most prototypes

```
connector interface service =
   role
      Declare client- and server-type roles
component interface server =
   port
      Declares a server-type port
   functional
      Declares operations for opening, closing sessions, and
      for information access
   interaction
      Declares that all the operations use the server-type port

configuration DIS(typeInt N) =
   component
      The DIS configuration is made of a number of
      client components and a server component
   connector
      Declares a service connector per client component
   binding
      Each client component port binds to the client
      role of the connector it is associated to.
      The server role of all the connectors are bound
      to the role of the server component
   functional binding
      Binds the operations of each client component with
      the corresponding operations of the server component
```

Fig. 3. An example of configuration description

of ADL-based development environments by integrating architectural elements aimed at the same platform although different platforms may be targeted by a prototype. The Darwin environment currently supports the construction of distributed systems above the Regis [28] (A specific platform developed at Imperial College) and CORBA platforms. The Aster [20] and Olan [6] prototypes also support the implementation of applications above CORBA platforms. The Olan environment further provides base solutions for the integration of heterogeneous architectural elements through dedicated declarations within connectors, which is further detailed in Section 4. The UniCon environment [39] deals with lower level platforms by addressing specifics of the underlying operating system. In particular, it enables the implementation of applications requiring real-time scheduling capabilities (Real-Time Mach operating system in the case of the UniCon prototype) [40]. The implementation of an application from its architec-

tural description does not prescribe any specific execution platform. Platforms that are supported by an environment result in general from design and implementation decisions for the given prototype based on different factors such as the types of applications that are targeted or the implementation effort required for the integration of architectural elements.

3 Synthesizing Middleware from Non-functional Requirements

Middleware services provided by infrastructures such as CORBA, DCOM or EJB can all be characterized by the functionality they provide when combined into a middleware. The principal idea of the systematic middleware synthesis that is being investigated within the Aster research activity[1] is to match this functionality against the demands of the application. Based on the match, an appropriate set of middleware services is selected and combined to form the middleware with desired properties. Thus, the entire synthesis process is driven by demands of the application that is to use the middleware.

In the current practice, specification of properties provided by a middleware is semi-formal, consisting of a description of the middleware interface and a description of the middleware behavior. The interface description is formal, given in an interface definition language specific to the infrastructure. The behavior description is informal, given in a natural language. Such an informal description is not useful for the systematic customization of middleware, as it lacks precision and makes automated reasoning about required and provided properties virtually impossible. So as to remedy this problem, we employ the basic concepts introduced in the software architecture paradigm. The input of the systematic customization is an architecture description of the application that, apart from the definitions given previously, includes a specification of the requirements on the properties of the middleware connectors that mediate the interaction among the components (i.e. non-functional properties required by the application). We also rely on the recursive nature of the architecture description to specify how the middleware connectors are built from the middleware services.

Figure 4 depicts an architecture description using the Aster ADL[2] of the example application we use throughout this section to demonstrate the systematic synthesis of middleware. The example application is a specialization of the DIS example used in the previous section and describes a (simplified version) of a distributed file system. The specification describes two component types, *FileClient* and *FileServer*, and a connector *ReliableConnector*. The *FileServer* component type provides an interface that exports a set of basic operations for accessing files, the *FileClient* component type requires the same interface so as to issue requests for accessing files. The *ReliableConnector* definition requires

[1] Information about the Aster activity may be found at the URL: http://www.irisa.fr/solidor/work/aster.

[2] The syntax taken for the Aster ADL is based on the one of the TINA-ODL object definition language, which is itself an extension of the OMG's IDL.

```
interface FileSystem {
  void fopen (in string name, out handle file);
  void fread (in handle file, in long size, out sequence<octet> data);
  void fwrite (in handle file, in sequence<octet> data);
  void fclose (in handle file);
};

object FileClient { requires FileSystem FS;};
object FileServer { provides FileSystem FS;};
connector ReliableConnector { property CModel, TModel; };

configuration Application
{
  instances
     Server FileServer;
     Client FileClient;
     Connector ReliableConnector;
  bindings
     Client.FS to Server.FS through Connector;
};

property CModel { ReliableComm; };
property TModel { Atomicity, Isolation; };
```

Fig. 4. An example architecture description

the connector to provide properties *ReliableComm* for reliable communication, and *Atomicity* and *Isolation* for the atomicity and isolation properties known from the flat ACID transaction model. The configuration part of the description defines two components, *Client* and *Server*, of the *FileClient* and *FileServer* component types. The interfaces of those components are bound together through a connector of the *ReliableConnector* type.

3.1 Property Specification

Until now, our definition of properties provided by or required from a middleware connector has been rather intuitive. To precise the term, we first define *observable behavior* of a middleware as a sequence of events that influence the application components. Ideally, the middleware would be completely encapsulated and the observable behavior would be defined as the interaction that takes place through the middleware interfaces. Typically, however, the middleware can also influence the application components through the operating system, libraries or other shared resources. This makes our definition of the observable behavior include not only the interaction through the interfaces, but also the indirect influence

on the state of the components, the activities executing within the components, etc. Based on the definition of observable behavior, a property of a middleware is then simply a constraint on this behavior. Expressed in a natural language, examples of such constraints can be:

- Whenever a request is issued by a source component for delivery, it is eventually delivered to the destination component once. No request is delivered unless issued. This constraint intuitively defines a property of reliable communication.
- When a set of requests is processed by a set of components and processing of any request of the set ends with failure, the state of the components at the end of processing the requests will be the same as at the beginning of processing the requests. This constraint reflects the atomicity property from the ACID transaction model.
- While a set of requests is processed, no request not belonging to this set will be processed by the components processing this set. This constraint reflects the isolation property from the ACID transaction model.

Formalized Specification The formalized specification of a middleware property in temporal logic follows the approach of defining constraints on the observable behavior. The events that make up the observable behavior are described by temporal logic predicates that hold whenever the particular event happens, the constraints on the observable behavior are expressed by temporal logic formulas. In this paper, we employ the temporal logic notation found in [29]. Here is a brief list of operators used:

- Boolean operators $\wedge$ (and), $\vee$ (or), $\neg$ (not), $\Rightarrow$ (implies), with their usual meaning.
- Quantifiers $\forall$ (for all), $\exists$ (exists), with their usual meaning.
- Operators $\oplus P$ (next), $\ominus P$ (previous), stating that P holds at the very next time in the future, resp. that P held at the just passed time in the past.
- Operators $\Diamond P$ (eventually), $\Diamond\!\!\!\!-\, P$ (once), stating that P holds at some time in the future, resp. that P held at some time in the past.

Here is further a brief list of symbols and predicates used:

- A set of components of the application, $\mathcal{C}$.
- A set of possible states of a component $C \in \mathcal{C}$, Σ_C.
- A set of requests exchanged by components, $\mathcal{R}$.
- Predicates characterizing components, namely:
  - $[\sigma_C]$, holding when $C \in \mathcal{C}$ is in state $sigma_C \in \Sigma_C$.
  - $export(src, dst, req)$, holding when $src \in \mathcal{C}$ exports request $req \in \mathcal{R}$ to $dst \in \mathcal{C}$.
  - $import(src, dst, req)$, holding when $dst \in \mathcal{C}$ imports request $req \in \mathcal{M}$ from $src \in \mathcal{C}$.
- Predicates characterizing requests, namely:
  - $failure(req)$, holding if $req \in \mathcal{R}$ reports a failure.

The above notations make it possible to formally specify a middleware property as a temporal logic formula that describes constraints on the observable behavior. Following our example, the specification of the reliable communication property in temporal logic is[3]:

$$
\begin{aligned}
&ReliableComm \equiv \forall src, dst \in \mathcal{C}, req \in \mathcal{R} : \\
&\quad (export(src, dst, req) \Rightarrow \\
&\qquad \Diamond(import(src, dst, req) \wedge \neg \ominus \Diamond\!\!\!-\, import(src, dst, req) \wedge \\
&\qquad \neg \oplus \Diamond import(src, dst, req))) \wedge \\
&\quad (import(src, dst, req) \Rightarrow \Diamond\!\!\!-\, export(src, dst, req))
\end{aligned}
$$

The two remaining properties from our example, atomicity and isolation, are specified in a similar way. We use three additional predicates in the specification to denote involvement of a component in processing a set of requests, and beginning and end of processing the requests[4]:

- $involved(C, \mathcal{S})$ holds if $C \in \mathcal{C}$ is involved in processing $\mathcal{S} \subset \mathcal{R}$, formally $\exists src \in \mathcal{C}, req \in \mathcal{S} : \Diamond\Diamond\!\!\!-\, import(src, C, req)$.
- $begin(C, \mathcal{S})$, $end(C, \mathcal{S})$ holds when $C \in \mathcal{C}$ begins, resp. ends, processing requests from $\mathcal{S} \subset \mathcal{R}$.

$$
\begin{aligned}
&Atomicity \equiv \forall \mathcal{S} \subset \mathcal{R}, C \in \mathcal{C} \mid involved(C, \mathcal{S}) : \\
&\quad \exists req \in \mathcal{S} \mid failure(req) \Rightarrow \\
&\qquad \exists \sigma_C \in \Sigma_C \mid (\Diamond\Diamond\!\!\!-\, ([\sigma_C] \wedge begin(C, \mathcal{S}))) \wedge (\Diamond\Diamond\!\!\!-\, ([\sigma_C] \wedge end(C, \mathcal{S})))
\end{aligned}
$$

$$
\begin{aligned}
&Isolation \equiv \forall \mathcal{S} \subset \mathcal{R}, req \in \mathcal{R} - \mathcal{S}, src \in \mathcal{C}, dst \in \mathcal{C} \mid involved(dst, \mathcal{S}) : \\
&\quad (import(src, dst, req) \wedge \oplus \Diamond begin(dst, \mathcal{S})) \vee \\
&\quad (end(dst, \mathcal{S}) \wedge \oplus \Diamond import(src, dst, req))
\end{aligned}
$$

Property Matching The primary use for the formalized property specification is an automated matching of properties provided by the middleware against properties required by the application. A middleware providing property P can be used in an application requiring property R if any observable behavior that satisfies the constraints characterizing P also satisfies the constraints characterizing R. In this case, we say that P *refines* R. Formally, the refinement is expressed as an implication between the constraints. Let P denote the constraints characterizing property P and R the constraints characterizing property R. Then, P matches R if $P \Rightarrow R$. Although formally precise, this definition of property matching does not reflect the differences between the specification of provided and required properties.

A specification of a required property is something perceived by the application designer. Typically, such a specification is short and abstract, not going

[3] For sake of brevity, we presume that each request is unique, i.e. the same request cannot be issued more than once.

[4] In a full property definition, these predicates would be related to e.g. *begin*, *commit* and *abort* operations provided by the middleware connector.

into detail unless it is necessary for application functionality. On the other hand, a specification of a provided property is associated with a specific middleware, and is therefore very detailed so as to describe the middleware precisely. The difference in the level of detail can often lead to a situation where none of the provided middleware properties quite satisfies the specification of the required property simply because this specification does not allow for minor differences that are not, in fact, vital for the application. This is the case in our example, where the specification of the *ReliableComm* property requires absolutely reliable delivery that cannot be provided in a realistic environment. Instead, the available middleware will guarantee slightly weaker property of at-most-once or at-least-once delivery, which is probably what the application designer had in mind when specifying the requirement anyway. Similarly, the available middleware might provide atomicity as long as the number of components involved is less than 32767 or other handy limit. Again, such a service would not meet the requirements for atomicity as specified by *Atomicity*, even though the limit is probably something the application designer would not mind. To overcome this drawback, we introduce a concept of an *ideal behavior* as a behavior that the middleware would exhibit if there were no failures in neither the hardware nor the software the middleware relies on, and if there were no resource shortages. We do not further detail specification of middleware behavior, the interested reader is referred to [43].

When built solely on the predicates introduced at the beginning of this section, the temporal logic specifications tend to grow quickly in size even for relatively simple middleware properties. The specifications are then not only difficult to write and read, but the proofs of relationships between them become computationally expensive. In a sense, this is similar to building a program out of too primitive statements, and can be solved by introducing predicates that describe complex events and conditions, as was done in the case of the *involved*, *begin* and *end* predicates in the example above. What predicates are used then depends on the domain of the property the temporal logic formula describes. For instance, formulas describing communication properties can take advantage of a predicate stating what transport protocol the middleware uses, rather than specifying the format of the messages on the wire.

3.2 Middleware Architecture

With the formalism for specifying middleware properties in place, we now focus on the description of the middleware architectures that provide specific properties. The goal is to provide a middleware architecture description associated with, and later on selected by, the properties that it provides. Once selected, the architecture description is used to assemble the middleware as per the application requirements. In principle, there is no significant difference between architecture description of middleware and architecture description of any other software system. Employing this fact, we use the same architecture description language to describe the middleware structure as the one we use to describe

the structure of the entire application in Figure 4. In our example, a middleware architecture that provides reliable communication, atomicity and isolation is required. Focusing on the CORBA infrastructure, as mentioned in the previous section, such a middleware can be built from an ORB and the OTS and CCS services.

Reusing Middleware Architectures A middleware architecture often reuses properties that are provided by other middleware architectures. In our example, we can consider the ORB component alone as a middleware architecture that provides reliable communication. The rest of the example then builds on top of the reliable communication to provide the atomicity and isolation properties. Any middleware architecture that provides reliable communication compliant with the CORBA standard can be used in place of the ORB component. To reflect this, we allow a component definition within the middleware architecture to refer not only directly to a specific middleware service, but also indirectly to any other middleware architecture that provides specific properties required from the component. This, in fact, is a recursive application of the principle where a connector definition lists required properties rather than specifying a particular connector to be used. A fragment of a middleware architecture description illustrating our example is presented on Figure 5. The description does not detail the interconnection of the OTS and CCS services as, although the two services are specified independently by OMG, they cannot in fact be implemented separately. The *Binding* interface denotes the interface the middleware architecture is to bind within the application, its definition is therefore taken from the application architecture description. The *TSIdentification* interface is a standardized interface used to connect an ORB with an OTS service, its functionality is described by the *TSPortability* property.

From Architecture to Implementation To assemble a middleware on the implementation level, we require the implementations of the middleware services to export operations defined in the middleware architecture description[5]. The implementations are then linked together with a binding code generated from the architecture description. This process is complicated by the fact that some of the interface definitions within the middleware architecture are derived from the interface definitions of the application components connected by the middleware. In our example on Figure 5, these are the interfaces of type *Binding*. Some of the middleware services come with a specialized tool for generating implementation based on the interface description, such as an IDL compiler available with an ORB. Such tools, however, may not be available for every middleware service used within the architecture, or may not accept input in the form used within the architecture description. To overcome this problem in the systematic synthesis framework, we introduce a macro language that makes it possible to

[5] We also require the implementations to provide certain housekeeping operations such as operations for service initialization and shutdown.

```
interface Transaction {
  void begin ();
  void commit ();
  void abort ();
};

object OTSCCSBlock {
  provides Binding ServerIn, Transaction Control;
  requires Binding ServerOut, TSIdentification Ident;
};
object ORBBlock {
  provides Binding ClientIn, TSIdentification Ident;
  requires Binding ServerOut;
  property ReliableComm, TSPortability;
};

configuration Middleware
{
  provides Binding ClientIn, Transaction Control;
  requires Binding ServerOut;
  instances
     OTSCCS OTSCCSBlock;
     ORB ORBBlock;
  bindings
     ClientIn to ORB.ClientIn;
     Control to OTSCCS.Control;
     ORB.ServerOut to OTSCCS.ServerIn;
     OTSCCS.ServerOut to ServerOut;
     OTSCCS.Ident to ORB.Ident;
};
```

Fig. 5. An example middleware architecture

parameterize those parts of the middleware implementation that depend on the specific application interfaces [42]. The macro language can be used to provide expected input for the available implementation generation tools, or to generate the middleware implementation directly, as the situation requires. Taking our example into the ORBIX environment, the implementation of the ORB component is partially generated from the application interface description by the ORBIX IDL compiler. The output of the IDL compiler needs to be further supplemented with the binding code that sets up connection between the application components. Figure 6 provides fragments of both the IDL compiler input and the C++ binding code defined using the macro language. The *$(Macro)* macros expand according to the specific application architecture, in our example scenario this means the *$(Interface.Name)* macro expands to *FileSystem* etc.

```
interface $(Interface.Name) {
  $(Iterate $(Interface.Operations Op
    $(Op.Type) $(Op.Name) ($(Op.Args));
  )
};

$(Proxy.Name)::$(Proxy.Name) () {
  pReference =
    $(Proxy.Type)::_bind (":$(Target.Component)","$(Target.Host)"); };
```

Fig. 6. Usage of macros for ORBIX ORB

3.3 Middleware Repository

Selecting from the available middleware architectures based on the properties they provide leads to a need of a repository that contains the available middleware architectures together with descriptions of the properties they provide, and that can be searched with the required properties as a search key.

The middleware repository needs to be organized in a way that allows for efficient searching. This issue is even more emphasized by the fact that the middleware properties are matched using a theorem prover, which is computationally expensive. In traditional databases, efficient search is achieved by exploiting an ordering on the search keys. In the middleware repository, the search keys are middleware properties that can be partially ordered using the refinement relation from Section 3.1. The refinement allows to structure the repository as a lattice, allowing to employ search methods that are more efficient than a linear search [31]. Apart from properties provided by existing middleware architectures, the lattice structure of the repository also contains abstract properties that are not associated with any particular architecture. The abstract properties are used to group the detailed middleware properties into domains. The reason for introducing abstract properties is twofold. Since the properties typically provided by middleware architectures are qualified as not related by the refinement relation, the lattice structure built only from these properties would be rather shallow and thus of little help when trying to make the search efficient. The abstract properties make the lattice structure deep enough to warrant more efficient searching [20]. The abstract properties also make it easier to browse the middleware repository. The domains defined by the abstract properties provide an ample navigation aid to the application designer, who will typically want to browse the available properties before specifying the application requirements rather than providing the definitions of the required properties straight away. Within the repository, a middleware architecture is linked to those nodes of the refinement lattice that define the properties provided by the architecture. As detailed in Section 3.2, the middleware architecture is represented by a formal-

ized architecture description whose every component is associated either with a specific middleware service or with a list of properties the component is to provide.

3.4 Middleware Integration

Based on the repository, the systematic middleware synthesis takes part in three steps:

- First, the repository is searched based on the properties required by the application, and a middleware architecture that provides these properties is retrieved.
- Second, the retrieval is repeated recursively to replace every middleware component in the architecture that is specified by its properties with an architecture that those properties specify. The recursion stops when all components of the middleware architecture are associated with specific middleware services.
- Third, the middleware services are combined together according to the retrieved middleware architecture to form the synthesized middleware.

Note that at each step of the process, multiple valid results can be obtained if the repository provides multiple middleware architectures that satisfy the same requirements. In this case, the developer is asked for selecting the most suitable architecture among the eligible ones.

3.5 Discussion

This section has provided an overview of the features offered by the ADL-based Aster development environment so as to support the systematic customization of middleware given the non-functional requirements imposed by the application under construction. The Aster prototype described in [20] is currently being enhanced regarding the middleware selection process, including the use of the STeP theorem prover [11] and the efficient implementation of the middleware repository. The reader further interested by detailed specification of non-functional properties may refer to [8,9] and [38] for security and fault-tolerance related properties, respectively.

Except the benefits of characterizing non-functional properties within architecture description from the standpoint of middleware synthesis, this is also beneficial from the standpoint of the software design process. As illustrated in [38], the specification of non-functional properties may serve as a basis for characterizing generic architectures aimed at the enforcement of non-functional properties, which may be conveniently combined with an application architecture so as to produce the overall architecture of a given software system. Open issues regarding the proposed approach include the combination of possibly interfering non-functional properties within an architecture [22]. Another issue relates to the practical use of the Aster environment where software developers are in general reluctant to the use of formal specifications. The aforementioned issues are currently being examined within the Aster research activity.

4 Deploying Distributed Applications

In the previous section, we have presented an approach to the systematic synthesis of middleware with respect to the applications requirements. Given the middleware to be used by an application, it is further required to conveniently deploy the application's components. In addition, enhanced software reuse is supported if the environment enables to encapsulate legacy software within components. This section gives an overview of the Olan environment[6], which offers the ADL-based Olan Configuration Language (OCL) [4,41] and a number of tools for the deployment of configurations, possibly made up of legacy software [3].

4.1 Olan Configurations

From an OCL description, the OCL compiler generates the stub classes for the components and the connectors (if these do not already exist) needed by the application. These are stored in a distributed repository. The compiler also generates a script (called the deployment script) that contains orders and guidelines for the deployment of the application in a distributed environment. The remainder of this section details the execution structures which are generated for components and connectors, as well as the interpretable deployment scripts.

Execution Structures Basically, two kinds of structures can be distinguished at runtime: the execution structures for components encapsulating legacy software and their binding to the underlying communication systems, and the execution structures for connectors in charge of integrating a particular communication schema between components. In the current prototype, execution structures are implemented as objects programmed in the Python interpreted language [37]. Object orientation is a convenient way to manage customizable classes of components and connectors. Let us now detail the components and connectors execution structures.

A component is represented by an object that is responsible for managing the interface with the encapsulated code as well as the communications with other components via connectors. The main characteristic of such an object is to be configurable and its purpose is to allow dynamic positioning of the interconnections with other components as well as dynamic loading of integrated software. In between the component object and the various integrated software (called modules), is the stub, that homogenizes parameters format, and the wrapper that knows how to access the encapsulated modules according to the kind of integrated software. Stubs and wrappers are automatically generated by the compiler. According to the kind of modules (or classes), the work to be performed by the programmer to have his code integrated, ranges from almost nothing to the explicit redirection of the outgoing calls to the wrapper.

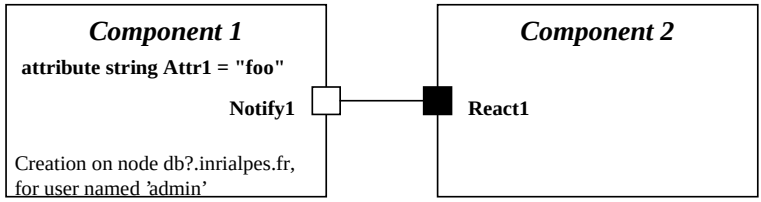
[6] Information about the Olan activity can be found at the URL: http://sirac.inrialpes.fr.

The main role of connectors is to bind a set of potential senders to a set of potential receivers for communication schemes based on service request or event notification broadcast. A connector is represented by two main kinds of Python objects: the service adapters (resp. notification) and the sender/receivers objects The *adapters* represent both the entry and exit points for the connector structure. On a sender side, they provide a function that allows a given component to initiate a communication as described in OCL. On a receiver side, they have the ability to call a given function (representing either a provided service or a notification handler), provided by the component. The adapters are also in charge of executing the user-level code which may have been added to the connector description (e.g. code for data flow adaptation). The sender/receiver objects are in charge of:

- Encapsulating the use of the communication system,
- Handling the possible control flow translations (e.g. when a sender asks for an asynchronous service request while the receivers provide the service in a synchronous way), and
- Handling remote communication according to the placement of the interconnected components.

Deployment Scripts The deployment script (also called configuration machine script) contains a list of commands that can be executed by the Olan Configuration Machine (OCM). Those commands correspond to the requests for the creation of components and connectors execution structures and the interconnection between the components, according to the architectural requirement expressed in OCL. The command for the creation of components requires several information from the script: the name of the component in order to be able to find the associated execution structure and the administration parameters that characterize the node and context of creation. The interconnection parameters contain the name of the connector to be instantiated and the list of components (and the services name) that must be bound together. Several other kind of commands can be found in the deployment script corresponding to every abstractions of OCL (e.g. attribute assignment, creation of composite structures, ...).

For illustration, Figure 7 gives a script, which comes from an application described with OCL named *Appli*, where two primitive components *Component1* and *Component2* are defined (the first one with a required service named *Notify1* and an attribute named *Attr1* of type string, and the second with a provided service named *React1*). There is one instance of each component and they are interconnected together with the *asyncCall* connector. The value '*foo*' is assigned to the first component attribute. The administration parameters indicate that the first component instance must be created on a node whose Internet name is '*db*?*.inrialpes.fr*', and whose processor usage is less than 10% with less than 10 users connected to the machine. In addition, the user for which the component should be created must have '*admin*' as login name.




```
## Primitive Component creation and administration parameters
nInfo = ( 'name':  "(node.name [1:] == "db") and  ## constraints on Node
           (node.name[3:-1] == ".inrialpes.fr")",
   'IPAdr': "true",
   'platform': "true",
   'osType': "true",
   'osVersion': "true",
   'CPULoad': "node.CPULoad <= 10",
   'UserLoad': "node.UserLoad <= 10"
   )
uInfo = ('name': 'user.name == "admin"','uid': "true",
   ## constraints on User
   'grpId": "true")
CreatePrimitiveComponent('Appli_Component1Impl:Appli_Component1Impl:_1',
           uInfo, nInfo)
## Interconnections
## component1.Notify1() -> component1.React1() using asyncCall
Bind('Appli:AppliImpl_asyncCall_1:AppliImpl_1_52',
   [('Appli_Component1Impl:Appli_Component1Impl:_1', 'Notify1', 'itf')]
   [('Appli_Component2Impl:Appli_Component2Impl:24', 'React1', 'itf')])
## Attribute assignment \\
SetAttribute('Appli\_Component1Impl:Appli\_Component1Impl:\_1', \\
   '\_a\_Attr1',
   "('string*', 'foo')")
```

Fig. 7. Example of a deployment script

4.2 The Olan Configuration Machine

The OCM machine is in charge of creating and configuring components and connectors instances, according to the configuration constraints expressed in the deployment script produced by the OCL compiler. More precisely, the OCM performs the following tasks:

- Deploying components: for each component, the OCM tries to find a relevant node able to host it, according to its placement constraints.
- Installing components: the installation step consists of the creation of the components execution structures and the assignment of their initial parameters. Components can be created at various time: initially, at the beginning of the execution, or during execution if the OCL description contains dynamic component instantiation.
- Setting interconnections: once components are created on the various nodes, the setting of interconnections consist in creating a connector execution structure according to the user-specified communication mechanism and the optional insertion of the code in charge of data flow transformation. Let us mention that connector structure can be spanned on multiple nodes, depending on the component location.
- Support for the application execution: this final step allows users to launch an application execution and handles the authorized connections or withdrawals of users in a running session.

The general architecture of the OCM relies on several abstract machines: the *component machine* in charge of managing the components execution structures, the *connector machine* in charge of handling the configuration of interconnections and a *repository* allowing a distributed access to the components core implementation and the OCL compiler generated structures. Each node able to host an application execution contains an instance of the OCM. It relies on an Object Request Broker when remote communication between computer nodes are required. The ORB is used whenever remote configuration machines need to communicate with each other, for example at the deployment step when querying nodes or at the installation step when creating components remotely. However, the ORB is not used for the communication between components during the application execution (unless the application designer has specified its use in an interconnection).

Functions of the OCM Machine Three management functions are carried out by the OCM:

- The management of the distributed environment, called a *cell*,
- The control of the deployment and execution of a single distributed application, called a *session*,
- The management of the part of a distributed application that executes on a given node for a given user, called a *context*.

A distributed environment is managed by a set of *Cell Servers*, one per node for each cell. One particular Cell Server (called the Master Cell Server), is responsible for managing the join and the withdrawal of nodes inside the cell. A Cell Server provides features for the following tasks:

- Instrumenting the local node with predefined sensors that return information corresponding to the management attributes. Information returned by

sensors can be either static (e.g. the name, the IP address) or dynamic (e.g. the average load, the number of logged users).
- Specifying policies for the evaluation of administration parameters contained in the OCL description. The criteria for eligible nodes may differ according to the cell administrators.
- Managing the deployment of an application within the cell according to the scripts for the application. This task is performed through calls to the Session management level.

A given distributed application that is currently being executed is called a *session*. A session is represented and managed by one *session server*. The session server provides services for the support of the installation and configuration of components and connectors of the application. Some of these services, like the creation of a component require the use of the context management level. Each time a user launches an application, he specifies the name of the session in which he wants the application to be executed. This may thus correspond to either the launching of a new session, or to the incoming of a new user into an existing session. More precisely, the launch of an application execution is managed through an additional script produced by the OCL compiler, which is given in Figure 8.

```
Start(applicationName, serviceName, args):
  If the given session already exists
       Get the reference of the local Session Server
       Ask for the execution of the given service
  Else
       Create a new Session Server
       perform the application deployment
       Ask for the execution of the given service
```

Fig. 8. Script for launching an application

A *context* represents the part of an application that executes for a given user on a given node. In other words, a context is an execution space for components, which ensures that components belonging to distinct users will not execute in the same address space, for protection reason. A context is managed by a *context server*, which is in charge of the actual creation and initialization of components within the context. A context server is also responsible for the creation and initialization of the parts of the connectors that involve the components it manages.

4.3 Discussion

A prototype implementation of the OCL compiler and Configuration Machine has been achieved using the Python language [37]. Using Python is of a particular interest for rapid prototyping due to facilities such as dynamic typing, reflexivity features, easy manipulation of complex structures such as lists, dictionaries, etc. Another major interest is the portability of Python code across various platforms (flavors of Unix, NT, W95,...). However, the price to pay is the poor performance at runtime due to the interpretive approach. The configuration machine uses ILU [23], a CORBA compliant ORB, for its own communication purposes. However, ILU is not used for the actual execution of the application, except if the architecture contains explicit use of a connector based on ILU mechanisms. The component and connector structures are also implemented in Python. Stubs and wrappers may be implemented both in Python and in the native language of the integrated software modules. Finally, the implementation of the communication protocol within connectors depends on the kind of connector which is used. For instance, there are multiple implementations of a remote synchronous call: one using sockets, another one using sun RPC, and a third one using ILU.

It should be noted that good performances at runtime were not expected from this experiment. The choice of the various implementation languages and tools were mainly motivated by the objective of rapid prototyping. The lessons drawn from our experiments mainly concern: the feasibility of the approach; the ease of application configuration; the flexibility of the deployment procedure; the transparent use of a distributed environment.

The prototype has been used for the construction of two applications: a cooperative document editor [5] and an electronic mail facility. The choice of these applications was motivated by the wish to address real-life scenarios which actually require heterogeneous components to be integrated within a distributed environment. The first scenario consists in transforming an existing single-user interactive application (e.g. a document editor or a CAD tool) to be used within a distributed groupware environment. No change can be applied to the code of the application itself (as usually only binary code is available). A way to achieve this goal consists in replicating the application on each user node, and building a coordination function defined as a set of cooperating components also replicated on each node. These components communicate together to achieve the control of the coordination between them, but they are the only one allowed to interact with the instance of the application of their node. The second scenario consists in extending an existing application (in this case an electronic mail browser) with additional facilities, acting autonomously on behalf of the end-user (e.g. for filtering and/or forwarding messages according to parameters customized by the user). Here again, the application itself cannot be changed and new functions are implemented as a set of cooperating components (also called agents here because of their specific role) which interact with existing software modules (i.e. the sendmail program, and the Netscape mail browser in this scenario). In both cases, the Olan environment has proved to be helpful for the following reasons:

- The OCL compiler greatly aids in providing wrappers to encapsulate existing binary code thus integrating legacy applications in a distributed environment (e.g. the Netscape browser and the editor in the previous scenarios)
- The Olan Configuration Machine also revealed to be very helpful in the implementation of the deployment process of the distributed application configurations. In the previous scenarios, nothing related to the distribution has to be implemented by the component programmer. Everything concerning distribution is externalized and the remote communications are handled by connectors.

In addition, it should be noted that an OCL description facilitates the reusability at the architecture level. This is a major advantage as it allows easy customization of an application for a specific use. Customization can be achieved in two ways: at the component level, to provide new facility, functionally equivalent to the former one, but implementing new policies; at the connector level, to change the communication schema between a set of interrelated components. For example, in the cooperative editor application, the component in charge of implementing the floor-passing policy can be changed on demand. Moreover, the same architecture may be reused to extend an existing CAD application towards a groupware environment. This does not require to redevelop the whole application from scratch.

5 Conclusion

This chapter has given an overview of work in the software architecture field. Using Architecture Description Languages, an application is abstractly described as a configuration that consists of a set of components characterizing computation units, which are interconnected through connectors that define communication protocols. Associated to ADLs are methods and tools for architecture analysis. Analyses that can be performed include:

- Correctness of bindings among components, ensuring that the behavior of an operation provided by a component matches the one expected by the component that uses it.
- Correctness of component interconnection through connectors, ensuring that the behavior provided by a connector matches the one expected by the components that use it with regard to both interaction and non-functional properties.
- Compatibility among configurations, ensuring that a configuration is a specific instance of another.
- Behavioral analyses of architectures so as to prove properties relating to liveness, and safety.

Another area of active research in the field of ADLs, which has been the main focus of this chapter, is the provision of tools for the implementation of distributed applications from their architectural description. In particular, we have

presented features of the Aster and Olan ADL-based development environments, developed by BROADCAST members. The Aster environment provides support for the systematic synthesis of middleware from the (non-functional) requirements stated within the architectural description of an application. Such requirements serve for the selection of necessary middleware services, which are then composed with the application components. The Olan environment provides tools for the deployment of applications, including those made out of legacy software, over a distributed architecture.

The software architecture research field is quite recent and there is still much work to be done for it to address the overall requirements of distributed application construction. However, existing results already demonstrate that it is a promising approach. In particular, this research field is shown to offer a convenient testbed for the development of a number of CASE tools, which not solely ease the design and implementation of distributed software systems but are also applicable to real such systems.

References

1. R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, USA, 1997.
2. R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.
3. R. Balter, L. Bellissard, F. Boyer, M. Riveill, and J.Y. Vion-Dury. Architecturing and configuring distributed applications with olan. In *Proc. IFIP Int. Conf. on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, 1998.
4. L. Bellissard, S. Ben Atallah, F. Boyer, and M. Riveill. Distributed application configuration. In *Proc. 16th International Conference on Distributed Computing Systems*, 1996.
5. L. Bellissard, S. Ben Atallah, A. Kerbrat, and M. Riveill. Component-based programming and application management with Olan. In *Proceedings of the Object Based Distributed and Parallel Computation Franco-Japan Workshop (OBDPC'95)*, 1995.
6. L. Bellissard, F. Boyer, M. Riveill, and J. Vion-Dury. System services for distributed application configuration. In *Proceedings of the Fourth International Conference on Configurable Distributed Systems*, 1998.
7. P. A. Bernstein. Middleware: A model for distributed system services. *Communications of the ACM*, 39(2):86–98, 1996.
8. C. Bidan and V. Issarny. Security benefits from software architecture. In *Proceedings of* COORDINATION *'97: Coordination Languages and Models*, pages 64–80, 1997.
9. C. Bidan and V. Issarny. Dealing with multi-policy security in large open distributed systems. In *Proceedings of the Fifth European Symposium on Research in Computer Security*, pages 51–66, 1998.
10. J. Bishop and R. Faria. Connectors in configuration programming languages: Are they necessary. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, pages 11–18, 1996.

11. N. Bjorner, A. Browne, M. Colon, B. Finkbeiner, Z. Manna, M. Pichora, H. M. Sipma, and T. E. Uribe. STeP: The Stanford Temporal Prover Educational Release. Technical report, Stanford University, 1998. 1.4-a edition.
12. J. R. Callahan and J. M. Purtilo. A packaging system for heterogenous execution environments. *IEEE Transactions on Software Engineering*, 17(6):626–635, 1991.
13. Y. Chen and B. H. C. Cheng. Facilitating an automated approach to architecture-based software reuse. In *Proceedings of the IEEE International Conference on Automated Software Engineering*, pages 238–245, 1997.
14. F. DeRemer and H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, 2(2):80–86, 1976.
15. P. Donohe, editor. *Software Architecture*. Kluwer Academic Publishers, 1999.
16. D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of the ACM SIGSOFT'94 Symposium on Foundations of Software Engineering*, pages 175–188, 1994.
17. D. Garlan, R. Monroe, and D. Wile. ACME: An architecture interchange language. Technical report, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, USA, 1997. http://www.cs.cmu.edu/afs/cs/project/able/www/-papers.html.
18. D. Garlan and D. E. Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4):269–274, 1995.
19. V. Issarny. Configuration-based programming systems. In *Proceedings of SOFSEM'97: Theory and Practice of Informatics*, pages 183–200, 1997.
20. V. Issarny, C. Bidan, and T. Saridakis. Achieving middleware customization in a configuration-based development environment: Experience with the Aster prototype. In *Proceedings of the Fourth International Conference on Configurable Distributed Systems*, pages 275–283, 1998.
21. V. Issarny, C. Bidan, and T. Saridakis. Characterizing coordination architectures according to their non-functional execution properties. In *Proceedings of the Thirty First Hawaii International Conference on System Sciences*, pages 275–285, 1998.
22. V. Issarny, T. Saridakis, and A. Zarras. Multi-view description of software architectures. In *Proceedings of the Third ACM SIGSOFT Software Architecture Workshop*, 1998.
23. B. Janssen and M. Spreitzer. ILU 2.0alpha10 Reference Manual. Technical report, Xerox Corporation, Palo Alto, CA, 1996.
24. J. Kramer and J. Magee. Exposing the skeleton in the coordination closet. In *Proceedings of* COORDINATION *'97: Coordination Languages and Models*, pages 18–31, 1997. LNCS 1282.
25. D. Le Métayer. Software architecture styles as graph grammars. In *Proceedings of the ACM SIGSOFT'96 Symposium on Foundations of Software Enineering*, pages 15–23, 1996.
26. D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, 1995.
27. J. Magee and J. Kramer. Dynamic structure in software architecture. In *Proceedings of the ACM SIGSOFT'96 Symposium on Foundations of Software Engineering*, pages 3–14, 1996.
28. J. Magee, J. Kramer, and M. Sloman. REGIS: A constructive development environment for distributed programs. *Distributed Systems Engineering*, 1(5):663–675, 1994.
29. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

30. N. Medvidovic and R. Taylor. A framework for classifying and comparing architecture description languages. In *Proceedings of the Joint European Software Engineering Conference - ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 60–76, 1997.
31. A. Mili, R. Mili, and R. Mittermeir. Storing and retrieving software components: A refinement based system. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 91–100, 1994.
32. M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct architecture refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, 1995.
33. OMG. Object management architecture guide (OMA guide). Technical Report 92.11.1, OMG, 1992. http://http.omg.org.
34. D. E. Perry. The Inscape environment. In *Proceedings of the Eleventh International Conference on Software Engineering*, pages 2–12, 1989.
35. D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.
36. J. M. Purtilo. The Polylith software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, 1994.
37. Rossum Van R. Python Reference Manual. Technical report, Dept. AA., CWI, 1995.
38. T. Saridakis and V. Issarny. Developing dependable systems using software architecture. In *Software Architecture*, 1999.
39. 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*, 21(4):314–335, 1995.
40. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Disciplines*. Prentice Hall, 1996.
41. J.-Y. Vion-Dury, L. Bellisssard, and V. Marangozov. A component calculus for modelling the olan configuration language. In *Proc. Second Int. Conf. on Coordination Models and Languages, (COORDINATION'97)*, 1997.
42. A. Zarras and V. Issarny. A framework for systematic synthesis of transactional middleware. In *Proceedings of Middleware98 – IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 257–274, 1998.
43. A. Zarras, P. Tuma, and V. Issarny. Using software architecture for the systematic synthesis of middleware. Submitted for publication, 1999.