

Software Architectural Specification for Optimal Object Distribution

M. Cecilia Bastarrica^{*}, Steven A. Demurjian Sr.^{*}, and Alex A. Shvartsman[†]
{cecilia,steve,aas}@cse.uconn.edu
Computer Science & Engineering Dept.
191 Auditorium Rd., Box U-155
The University of Connecticut, Storrs, CT 06269-3155, USA

Abstract

Software architectural design is essential for complex distributed applications. Architectural specifications need to define the key aspects of the application, including its intended functions, interfaces, interoperability requirements, distributability, scalability, extensibility, target platforms and networks, and required middleware services. In this paper, we present an architectural extension for specifying distributable components of the application and for quantifying their interaction patterns. The goal of the specification is to serve as the basis for obtaining optimal distributions of the application components over a target network that minimizes remote communication among the components. The optimal distributions are obtained by using the architectural specification to derive a BIP (binary integer programming) model and solving the resulting system. We formalize the object-oriented architectural style using the Z specification language. This style defines the detailed information necessary to derive optimal distributions using the BIP model.

1. Introduction

Software architecture design is an accepted part of software development methodologies. Architectural specification frameworks are used to define key aspects of applications using a variety of specification techniques. For distributed software applications, the architectural activity involves, at a minimum, the identification of distributable software components and their interconnections and interfaces. For example, this can be done in an object-oriented architectural style using CORBA IDL [9]. Other architec-

tural styles [17] may use implicit-invocation and broadcast, where the communication is one-to-many and one-to-all, respectively. For the repository and client-server architectures there are specialized components: clients, which generate request messages, and repositories and servers that respond to requests. If a distributed application has rigid deployment patterns for specialized components, then once the application is modeled using some of these architectural styles, the resultant architecture completely characterizes the interactions within the distributed system.

Designing and developing distributed software for systems with flexible or modifiable component deployment patterns involves aspects of traditional software design, but also includes activities related to component distribution. OSF started addressing some of these activities as part of its I4DL proposal [14], specifically dealing with inheritance, interface, implementation, and instantiation in the distributed setting. Additional activities may be necessary to specify where the components are to be located in the network and how the communication protocols or middleware is to be used. Depending on the technology used, the decision of where to locate the components in the network can be postponed to the point of application installation or deployment (cf. CORBA [9]).

Performance of distributed systems depends on many factors that range from the efficiency of the abstract algorithms to the capabilities of the underlying communication networks and the processing nodes. The performance of a particular system also depends on the way that the components are deployed in a specific network. Components that are mapped to the same node interact by communicating *locally*, while components mapped to different nodes interact by communicating *remotely* using the underlying network. Local communication is faster and cheaper, and it is not affected by the network characteristics. Distributing components enables the application to take advantage of available computation and storage resources in the network, but requires remote communication, which is typically slower and more expensive, and which critically depends on the under-

^{*}The work of these authors was partially supported by The Mitre Corp., Eatontown, NJ.

[†]The work of this author was partially supported by a grant from the GTE Laboratories, Waltham, MA, and performed in part at the Massachusetts Institute of Technology, Cambridge, MA.

lying network. Thus, minimizing the remote communication in a distributed system while maintaining the necessary distribution of its components is an attractive goal.

In this paper, we present an architectural framework for specifying distributable components of the application and for quantifying their interaction patterns. This framework extends traditional architectural specifications with the goal of obtaining optimal distributions of the application components over a target network that minimizes remote communication among the components. Our object-oriented architectural style is formalized using the Z specification language [18]. The optimal distributions are obtained by using the architectural specification to derive a BIP (binary integer programming) model and solving the resulting system.

Installing a distributed application can be very expensive, so it is desirable to do it right the first time. Colored Petri Nets [6, 11] and static coupling evaluation [7] have been used to simulate distributed software execution. However, such techniques, while checking for constraint violations and for the absence of bottlenecks, do not provide the means to determine an initial deployment of components to be tested. Our BIP model allows us to calculate the optimal deployment of an application over a given network, minimizing the remote communication. The parameters of the BIP model are the storage needed for each component, the frequency for each message in the interface, the storage available in each network node, and the maximum throughput that network connectors can handle. We have already applied our BIP model to a standalone legacy C++ application and we have obtained an optimal result [4].

In [12], there is an attempt to provide a methodology for optimal distribution of components using similar parameters, but the authors do not give specific decision criteria and instead suggest the use of AI techniques or attempt to reuse an existing similar deployment. In [15], a BIP model is utilized for optimization of object distribution. That model has different parameters—distance between sites is modeled as contrasted with our modeling of connector capacities. Our approach provides a specific algorithm for optimizing the deployment, and we also provide an upper-bound for the time-complexity of the algorithm [4].

Section 2 formalizes the object-oriented architectural style based on the object type and class concepts. In Section 3, we explain how the architectural specification should be enriched, the way the parameters are derived, and the BIP model equations. Finally, in Section 4, we describe the ongoing work and offer concluding remarks.

2. Software Architectural Design

Architectural software design consists of determining the software components and the communication between those

components, and may vary from just a box and line diagram to a formal definition of the components functionality and a complete characterization of the communication [8, 17]. The level of detail and the kind of details included depend on the information available and the utilization of the design. If the architectural design is going to be utilized to analyze the possibility of connecting all of the described components, it should include a description of the style of connectors [3]; if the functionality is going to be tested to check that all of the requirements are satisfied, components should also be characterized [2, 13].

2.1. The Object Architectural Style

In the object architectural style, the components are objects—or instances of a class or object type—and the communications are method calls or message passing to other objects. Following the formalism used in [1] and [16], we employ the Z specification language [18] to define the object architectural style.

An *Object_Type* is defined by its *state* or list of attributes, and its interface or the methods in the *provides* and *needs* lists. All of the methods in the *provides* list are targeted to `self Object_Type`.

[OBJTYPE]

Object_Type _____

ot_id : OBJTYPE

state : \mathbb{N}

provides : \mathbb{P} Method

needs : \mathbb{P} Method

$\forall m \in \textit{provides} \bullet m.\textit{target} = \textit{self}$

Method _____

size : \mathbb{N}_1

target : *Object_Type*

`self` \in *target.provides*

The *provides* list is the set of services that other *Object_Types* can invoke, while the *needs* list specifies those services required from other *Object_Types*.

Methods are the specification of the relationships between *Object_Types*. *Methods* in the *needs* list indicate services provided by other *Object_Types* that are necessary to be present whenever this *Object_Type* is used. *Methods* in the *provides* list specify the set of services that other *Object_Types* can invoke, and so the programmer should insure this *Object_Type* implements them.

A *Class* is the implementation of an *Object_Type*; it has a set of *Object* instances that form its *extent*. Communication is implemented as *Message* passing and these messages

correspond to the implementation of the *Methods* defined in the *Object_Type*. The *uses* and *exports* lists of a *Class* are defined as bijections between the *Methods* defined for the *Object_Type* and the *Messages* implemented. *Messages* are the connectors for the object-oriented architectural style. They can be defined independently, but their role is given by their link with the *Classes* as part of their *exports* or *uses* functions. An *Object* can then be defined as an element of the *Class extent*.

<p><i>Class</i></p> <p><i>Object_Type</i></p> <p><i>extent</i> : $\mathbb{P} \text{Object}$</p> <p><i>uses</i> : <i>Method</i> \rightarrow <i>Message</i></p> <p><i>exports</i> : <i>Method</i> \rightarrow <i>Message</i></p> <hr/> <p>dom <i>uses</i> = <i>needs</i></p> <p>dom <i>exports</i> = <i>provides</i></p> <p>$\forall mt \mapsto ms \in (\text{uses} \cup \text{exports}) \bullet$</p> <p style="padding-left: 2em;"><i>mt.size</i> = <i>ms.size</i> \wedge</p> <p style="padding-left: 2em;"><i>mt.target.ot_id</i> = <i>ms.target.ot_id</i></p>

<p><i>Message</i></p> <p><i>target</i> : <i>Class</i></p> <p><i>size</i> : \mathbb{N}_1</p> <hr/> <p>self \in ran <i>target.exports</i></p>
--

<p><i>Object</i></p> <p><i>class</i> : <i>Class</i></p> <hr/> <p>self \in <i>class.extent</i></p>

All *Objects* in the same *Class* have the same interface (*exports* and *uses*) and the same *state*, because they all belong to the same *Object_Type*. However, *Classes* are disjoint and each *Object* belongs only to one *Class*, even if there is more than one *Class* of the same *Object_Type*.

In order to have a well formed application, every *Message* that may be sent should have its *target Class* as part of the application, so that it is able to handle the message, that is, the *Message* is part of the range of its *exports* list. Based on the prior definitions, we can formally specify an *Application* for an object-oriented architecture.

<p><i>Application</i></p> <p><i>appl</i> : $\mathbb{P} \text{Class}$</p> <hr/> <p>$\forall c \in \text{appl}; m \in \text{ran } c.\text{uses} \bullet$</p> <p style="padding-left: 2em;">$\exists c' \in \text{appl} \mid m.\text{target} = c' \wedge$</p> <p style="padding-left: 4em;">$m \in \text{ran } c'.\text{exports}$</p>
--

As part of the software architectural design process, we can group some classes into *Components*. When encapsulating classes from a legacy application (bottom-up) or

designing new components from scratch (top-down), this characterization is the same, allowing for a hierarchical design. A *Component* is, as an *Application*, a set of *Classes*, with the only difference being that not all messages need to be satisfied within the component. The component's *exports* list is the union of the range of the *exports* of all of its *Classes*. The *uses* is formed by all of those *Messages* from the range of the *Classes*' *uses* whose *target* is not part of the same *Component*.

<p><i>Component</i></p> <p><i>comp</i> : $\mathbb{P} \text{Class}$</p> <p><i>exports</i> : $\mathbb{P} \text{Comp_Message}$</p> <p><i>uses</i> : $\mathbb{P} \text{Comp_Message}$</p> <hr/> <p><i>exports</i> = $\{ c \in \text{comp}; m \in \text{ran } c.\text{exports};$</p> <p style="padding-left: 2em;">$cm : \text{Comp_Message} \mid cm.\text{mess} = m \bullet cm \}$</p> <p><i>uses</i> = $\{ c \in \text{comp}; m \in \text{ran } c.\text{uses};$</p> <p style="padding-left: 2em;">$cm : \text{Comp_Message} \mid$</p> <p style="padding-left: 4em;">(let <i>ct</i> = <i>cm.comp_target</i> \mid</p> <p style="padding-left: 6em;"><i>ct</i> \neq self \wedge</p> <p style="padding-left: 6em;"><i>cm.mess</i> = <i>m</i> \wedge</p> <p style="padding-left: 6em;"><i>m.target</i> \in <i>ct.comp</i>) $\bullet cm \}$</p>
--

The communication between components is determined by those messages between classes that are not satisfied within the same component as shown in Figure 1.

<p><i>Comp_Message</i></p> <p><i>comp_target</i> : <i>Component</i></p> <p><i>mess</i> : <i>Message</i></p> <hr/> <p><i>mess.target</i> \in <i>comp_target.comp</i></p>
--

The *Soft_Arch_Application* is now defined in terms of the components and messages between components.

<p><i>Soft_Arch_Application</i></p> <p><i>soft_arch_appl</i> : $\mathbb{P} \text{Component}$</p> <hr/> <p>$\forall c \in \text{soft_arch_appl}; cm \in c.\text{uses} \bullet$</p> <p style="padding-left: 2em;">$\exists c' \in \text{soft_arch_appl} \mid$</p> <p style="padding-left: 4em;">$cm.\text{comp_target} = c' \wedge m \in c'.\text{exports}$</p> <p>$\forall cl : \text{Class} \bullet$</p> <p style="padding-left: 2em;">$\#\{ c \in \text{soft_arch_appl} \mid cl \in c.\text{comp} \bullet c \} \leq 1$</p>
--

A *Soft_Arch_Application* is a set of *Components* such that all foreign invocations must be solved within the application, as we did for *Application*. We also require that any *Class* can only be part of at most one *Component* of the *Soft_Arch_Application*. Even though many *Classes* of the same *Object_Type* can be implemented in different *Components*, they are different *Classes* and their *extents* are disjoint. This condition is important for *Messages* to have a uniquely identified *target* in the *Soft_Arch_Application*.

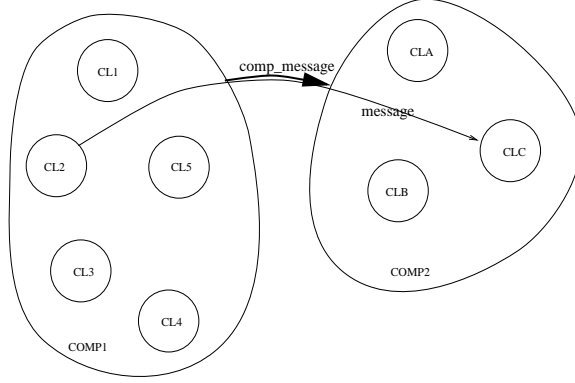


Figure 1. Communication Between Components.

3. Optimal Deployment of Distributable Units

The performance of a distributed application depends on the location of each component within the network. Since remote communication is both slower and more expensive, as compared to local communication, we consider minimizing the remote communication as a way of improving performance. The optimal deployment depends on both the application and the network characteristics.

In this section, we refine the definition given for a *Component* (see Section 2) and also provide a formal definition of the most relevant parameters of the network for our model. We introduce and discuss a binary integer programming (BIP) model that calculates the optimal deployment based on the application and the network parameters.

3.1. Storage and Communication

The way the application is used determines the storage needed for the components and the communication between them. The **storage** needed for the components is determined by the number of instances of each *Class* in the *Component* (*extent*), and the storage needed for each instance (*state*). We do not consider the storage needed for the method implementations of each clas, since this value does not scale with the extent. The **communication** between components is given by the amount of information units sent and is the product of the *size* of the *Messages* in the *uses* list and the *frequency* with which these messages are sent. We rewrite the definition of *Component* adding this runtime information.

$$\begin{array}{l}
 \text{Component}' \\
 \text{Component} \\
 \text{storage} : \mathbb{N} \\
 \text{frequency} : \text{Comp_Message} \leftrightarrow \mathbb{N} \\
 \hline
 \text{storage} = \sum_{c \in \text{comp}} c.\text{state} \times \#c.\text{extent} \\
 \text{dom frequency} = \text{uses}
 \end{array}$$

If we define an array *CS* of required storage for components, we will have, for any component *i*:

$$CS(i) = i.\text{storage} \quad (1)$$

and its value can be entirely derived from the class definitions of the component.

The communication between a pair of components per time unit is the summation of all of the messages sent from one component to the other per time unit. If we define an array *COMM*, the total communication between components *i* and *j* is:

$$COMM(i, j) = \sum_{m \in i.\text{uses} \wedge m.\text{comp_target} = j} m.\text{mess.size} \times i.\text{frequency}(m) \quad (2)$$

Here the *frequency* for each *Comp_Message* in the *Component's uses* list is provided by the designer. This information might be a rough estimation at an early stage of development, but most of the time it is derivable by either the system users or the designer. Notice that communication between components is not symmetrical. We are not considering the response to any message.

3.2. The Computers and the Network

The network characteristics are determined by the characteristics of the computers (nodes) and the connectors.

$$\begin{array}{l}
 \text{Node} \\
 \text{storage} : \mathbb{N}
 \end{array}$$

For the *Nodes*, we are only concerned with the available *storage*. If we have an array *NS* for the storage available in each node of the network, we can define the storage in a node *i* as:

$$NS(i) = i.\text{storage} \quad (3)$$

A *Connector* goes from one node to another of the network, and it has a certain capacity or bandwidth.

$\begin{array}{l} \text{Connector} \\ \hline \text{end1, end2} : \text{Node} \\ \text{capacity} : \mathbb{N} \\ \hline \text{end1} = \text{end2} \Leftrightarrow \text{capacity} = \infty \end{array}$
--

We are building a model for minimizing remote communication, so we assume there is no restriction on local communication. This is expressed by defining connectors between a node and itself with infinite capacity. If we define an array *CONN*, the capacity of the connector between nodes *i* and *j* in the network will be:

$$\text{CONN}(i, j) = c.\text{capacity} \mid c.\text{end1} = i \wedge c.\text{end2} = j \quad (4)$$

Connectors are not necessarily symmetrical.

In a similar fashion to our *Application* definition, we can define a *Network* in terms of its *Nodes* and *Connectors*.

$\begin{array}{l} \text{Network} \\ \hline \text{Nodes} : \mathbb{P} \text{Node} \\ \text{Net} : \text{Node} \times \text{Node} \mapsto \text{Connector} \\ \hline \forall n, n' : \text{Node}; c : \text{Connector} \bullet \\ \quad (n, n') \mapsto c \in \text{Net} \Leftrightarrow n \in \text{Nodes} \wedge \\ \quad \quad \quad \quad \quad \quad \quad n' \in \text{Nodes} \wedge \\ \quad \quad \quad \quad \quad \quad \quad c.\text{end1} = n \wedge \\ \quad \quad \quad \quad \quad \quad \quad c.\text{end2} = n' \end{array}$

3.3. BIP Model for Optimal Deployment

A feasible deployment of an *Soft_Arch_Application* over a *Network* can be formally specified as:

$\begin{array}{l} \text{Deployment} \\ \hline N : \text{Network} \\ A : \text{Soft_Arch_Application} \\ \text{Deploy} : \text{Component} \rightarrow \text{Node} \\ \text{Usage} : \text{Connector} \rightarrow \mathbb{N} \\ \hline \text{dom}(\text{Deploy}) = A.\text{soft_arch_appl} \quad [\text{A}] \\ \text{ran}(\text{Deploy}) \subseteq N.\text{Nodes} \quad [\text{B}] \\ \forall n \in N.\text{Nodes}; \forall c \in \text{Deploy} \triangleright n \bullet \\ \quad \sum_c CS(c) \leq NS(n) \quad [\text{C}] \\ \text{dom}(\text{Usage}) \subseteq \text{ran}(N.\text{Net}) \quad [\text{D}] \\ \forall cn \mapsto f \in \text{Usage}; \\ \quad c \in \text{Deploy} \triangleright cn.\text{end1}; \\ \quad c' \in \text{Deploy} \triangleright cn.\text{end2} \bullet \\ \quad (f = \sum_{c, c'} \text{COMM}(c, c')) \leq \\ \quad \quad \text{CONN}(cn.\text{end1}, cn.\text{end2}) \quad [\text{E}] \end{array}$

The *Deployment* schema is interpreted as follows:

Completeness - Each application unit is deployed to one and only one node of the network ([A] and [B]).

Storage - The total storage required by all of the components deployed to each node does not exceed the node's storage capacity ([C]).

Connectors - Components communicate with remote components only through the network connectors ([D]).

Throughput - For each connector, the total communication bandwidth does not exceed the connector's capacity ([E]).

Fixing the location of selected components is modeled by defining *a-priori* the value of some tuples of the *Deploy* function.

Among the possible deployments that satisfy all of these requirements, we are interested in the one(s) that minimizes the remote communication bandwidth. We state all of these conditions as constraints of a BIP problem whose objective function is to minimize the total remote communication.

Our basic **decision variables** are:

$$x_{i,j} = \begin{cases} 1 & \text{if component } i \text{ is assigned to node } j \\ 0 & \text{otherwise} \end{cases}$$

the **completeness** condition can be then expressed as:

$$\sum_{j=1}^N x_{i,j} = 1 \quad (5)$$

where *N* is the total number of nodes in the network. The **storage** constraint is:

$$\sum_{i=1}^C x_{i,j} \times CS(i) \leq NS(j) \quad (6)$$

where *C* is the total number of components in the application, and *CS* and *NS* are the parameters defined in (1) and (3), respectively.

For the remaining equations, we need to define additional auxiliary decision variables to express the conjunction of conditions of having two components assigned to two nodes:

$$y_{a,i,b,j} = \begin{cases} 1 & \text{if unit } a \text{ is assigned to node } i \text{ and} \\ & \text{unit } b \text{ is assigned to node } j \\ 0 & \text{otherwise} \end{cases}$$

These decision variables relate to the $x_{i,j}$ since $y_{a,i,b,j} = x_{a,i} \times x_{b,j}$. Note that this is a non-linear equation. But, by taking advantage of binary numbers properties, we can restate this non-linear equation as a series of linear inequalities:

$$y_{a,i,b,j} \leq x_{a,i} \quad (7)$$

$$y_{a,i,b,j} \leq x_{b,j} \quad (8)$$

$$1 + y_{a,i,b,j} \geq x_{a,i} + x_{b,j} \quad (9)$$

We can now state the **connectors** and **throughput** constraints:

$$\sum_{a=1}^C \sum_{b=1}^C COMM(a, b) \times y_{a,i,b,j} \leq CONN(i, j) \quad (10)$$

where $COMM$ and $CONN$ are the parameters defined in (2) and (4), respectively.

Finally, the **objective function** is expressed as:

$$\text{Min } Z = \sum_{i=1}^N \sum_{j=1(j \neq i)}^N \sum_{a=1}^C \sum_{b=1}^C y_{a,i,b,j} \times COMM(a, b) \quad (11)$$

Minimizing the remote communication ($j \neq i$) is equivalent to maximizing the local communication.

4. Ongoing Work and Conclusions

Using the architectural specification presented here, we showed in [4] how to use our BIP model to produce an optimal distribution. We used the GAMS optimization package for the implementation [5], and applied the technique to a standalone legacy C++ application, with the parameter values measured by monitoring the actual application execution.

Binary integer programming is a NP-complete problem [10]. Given the particular structure of our problem, we also showed in [4] that for the branch-and-bound algorithm used to solve the BIP problem the upper-bound on time is N^C , where N the number of nodes in the network and C is the number of distributable components in the application. The actual observed execution times have been substantially faster than what is suggested by the upper bound.

We are currently building an interactive tool that allows the designer to specify the architectural components by providing parameters directly or deriving them by parsing a piece of Java code. The characteristics of the network are specified as well. This tool checks the style constraints [1, 3], derives the type of communication for each style, checks the consistency of the architectural specification, and calculates the optimal distribution using the BIP model. We are experimenting with a more powerful solver than GAMS that allows us to solve larger problem instances.

We are also refining our application and network models by including new constraints that make the models more realistic. We plan to include the response messages as part of the communication load, the limit on the total communication a node can handle simultaneously given the physical connections to the network, and the maximum allowable response time for the different types of messages.

References

- [1] G. Abowd, R. Allen, and D. Garlan. *Formalizing Style to Understand Descriptions of Software Architecture*. ACM Transactions on Software Engineering, 1994.
- [2] Gregory Abowd, Jonathan Engelsma, Luigi Guadagno, and Okonon Okon. *Architectural Analysis of Object Request Brokers*. Object Magazine, special issue on distributed systems, 1996.
- [3] Robert Allen and David Garlan. *A formal basis for architectural connection*. ACM Transactions on Software Engineering and Methodology, 6(3):213–249, July 1997.
- [4] M. C. Bastarrica, Alex. A. Shvartsman, and Steven A. Demutjian. Sr. *A Binary Integer Programming Model for Optimal Object Distribution*. Technical Report CSE-TR-98-1, Department of Computer Science and Engineering, University of Connecticut, April 1998.
- [5] Anthony Brooke, David Kendrick, and Alexander Meeraus. *GAMS. A user's guide*. The Scientific Press Series. Boyd and Fraser publishing company, 1992.
- [6] A. Diagne and F Kordon. *A multi formalisms prototyping approach from formal description to implementation of distributed systems*. In N. Kanopoulos, editor, *Proceedings of the 7th International Workshop on Rapid System Prototyping*, pages 102–107, Greece, june 1996. IEEE comp Soc Press.
- [7] K. El Guemhioui. *Information Engineering of Parallel and distributed systems Using an Object-Oriented Design model*. PhD thesis, University of Connecticut, Department of Computer Science and Engineering, May 1997.
- [8] David Garlan and Dewayne E. Perry. *Introduction to the Special Issue on Software Architecture*. IEEE Transactions on Software Engineering, 21(4), apr 1995.
- [9] Thomas J. Howbray and Ron Zahavi. *The Essential CORBA. Systems Integration Using Distributed Objects*. John Wiley and Sons, Inc., 1995.
- [10] Frederick S. Hillier and Gerald J. Lieberman. *Introduction to Operations Research.*, Holden-Day, Inc., Fifth edition, 1995, Oakland, California.
- [11] J. B. Jorgensen and K. H. Mortensen. *Modeling and Analysis of Distributed Program Execution in Beta us ing Coloured Petri Nets*. In Lecture Notes in Computer Science Vol. 1091, pages 249–268, Osaka, 1996. 17th International Petri Net Conference, Springer-Verlag.
- [12] W. E. Kaim and F. Kordon. *An Integrated Framework for Rapid System Prototyping and Automatic Code Distribution*. In N. Kanopoulos, editor, *Proceedings of the 5th International Workshop on Rapid System Prototyping*, pages 52–61, Grenoble, France, june 1994. IEEE comp Soc Press.

- [13] Rick Kazman, Len Bass, Gregory Abowd, and Mike Webb. *SAAM: A Method for Analyzing the Properties of Software Architecture*. In Proceeding of the International Conference on Software Engineering - ICSE'16, pages 81 – 90, 1994.
- [14] Open Software Foundation. *OSF Distributed Management Environment (DME) Architecture*. May 1992.
- [15] S. Puroo, H.K Jain and D. Nazareth. *Effective Distribution of Object-Oriented Applications*. Communications of the ACM, 41(8), August, 1998.
- [16] Michael D. Rice and Stephen B. Seidman. *Using Z as a Substrate for an Architectural Style Description lan guage*. Technical Report CS-96-120, Department of Computer Science, Colorado State University, september 1996.
- [17] Mary Shaw and David Garlan. *Software Architecture. Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
- [18] J. M. Spivey. *Understanding Z*. Cambridge Tracts in Theoretical Computer Science 3. Cambridge University Press, 1995.