ODP-based Distribution Configuration

Shahrzade Mazaher and Georg Raeder
NR – Norwegian Computing Center
P.O.Box 114, Blindern
0314 Oslo, Norway

Shahrzade.Mazaher@nr.no, Georg.Raeder@nr.no

Abstract

The new generation of open, distributed, object-based systems poses many new challenges. For example, it raises the issue of how the application should be mapped onto a distributed target platform so that performance and reliability constraints are met. This paper shows how the abstract Engineering Model of ODP, together with the high-level language SDL, can be used to achieve system descriptions that are independent of the concrete platform, despite the implementation-oriented nature of distribution configuration. A distribution configuration language and its tool support are presented. The language today covers static distribution configuration. Controlling the evolution of ODP-based systems also requires support for dynamic reconfiguration, and the paper outlines an approach to how the language can serve as a basis for further work in this area.

Keywords: *Distribution configuration, configuration languages, distributed targeting, open distributed processing.*

1 Introduction

The new breed of *object-based open architectures* [8] ("middleware") aims at establishing a uniform view of a distributed software environment, hiding the heterogeneity and the physical location of system entities (distribution *transparency*), and defining a uniform object model on which the application can be designed. It is at this level that standards must be defined to ensure application interoperability, and this is the goal of the ISO and ITU-T emerging standard on Open Distributed Processing (ODP) [3], and of the Object Management Group's (OMG) architecture based on an Object Request Broker (CORBA) [9, 10].

While promising to simplify the development and evolution of large distributed applications, these architectures also call for new methods and tools for managing non-functional requirements, such as performance and reliability. The complexity of distributed architectures complicates this aspect. But as we shall see in this paper, proper modelling of implementation structures may give us a useful framework.

Our particular focus is on *distribution configuration*, which includes the specification of how objects are assigned to computing nodes and other engineering structures. This implementation aspect may have significant impact on the performance and reliability of the system. We present a notation for expressing how ODP objects are to be distributed in the network, and we show how it can be used to achieve *platform independence*, despite the implementation-oriented nature of this type of configuration. The current version of the language is used in conjunction with SDL, the ITU-T Specification and Description Language. The notation describes a static configuration, but we also show how it can serve as a basis for a constraint-oriented approach to dynamic system reconfiguration.

In Section 2 we first briefly present the ODP Engineering Model, on which our work is based. Section 3 discusses related work and the kind of distribution support needed for ODP-based systems, and Section 4 further elaborates on our context. Section 5 presents our main contribution, a language for specifying distribution configuration. Section 6 discusses some central issues, and Section 7 concludes the paper.

2 The ODP Engineering Model

The emerging ISO/ITU-T standard for Open Distributed Processing (ODP) [3] proposes a *Computational Model*, an abstract programming model that defines a conceptual object model that is a central

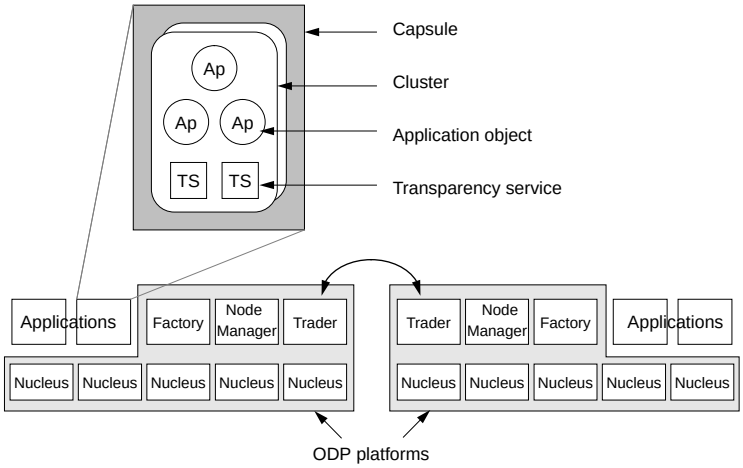

Figure 1: ODP Engineering Model concepts.

feature of that standard. In addition, an *Engineering Model* is defined that specifies an implementation architecture for the Computational Model. It introduces the following concepts:

- A *node* is an engineering abstraction of a host machine.
- A *node manager* maintains a configuration database about its node.
- A *trader* provides a trading space to which server objects can export interface references and from which client objects can import those references.
- A *nucleus* converts a host's resources to a basic virtual machine for distributed computing.
- A *transparency service* enables aspects of distribution to be hidden from application objects.
- A *factory service* creates capsules containing application objects.
- A *capsule* is the unit of protection and failure.
- A *cluster* is the unit of activation and migration.
- An *object* is the unit of encapsulation and distribution.

Figure 1 illustrates the concepts. A capsule is typically mapped to a process in the underlying operating system. The cluster provides an intermediate-level structure that can be used, for example, to group objects with strong interdependencies, causing them to be migrated together or to be stored together in a database. The concept of object is the same in both the Computational and the Engineering Models, i.e., it forms the link between the two models. The task of engineering in the ODP context thus includes the task of arranging the application objects in terms of engineering concepts such as clusters, capsules, and nodes.

3 Related work

For both parallel and distributed applications, the way in which they are mapped to the underlying processing units has great impact on their performance. In the case of parallel systems, better performance is achieved through methods and tools for analyzing the application (in terms of dataflow and -access, communication pattern, etc.). Based on the results of the analysis, optimizing compilers, in cooperation with run-time systems, can achieve improved scheduling and thus better performance. The approaches in this area are usually dependent on platform architecture, in fact, defining proper architectures is part of the solution to the problem.

In the case of object-based distributed systems, such as ODP- or OMG-based applications, the client-server nature of the model makes the communication pattern explicit, and the obvious candidates for distribution are the encapsulated objects of the application. The main task is thus to distribute these objects with the goal to satisfy the performance constraints put on the application via, e.g., *quality of service* (QoS) parameters. There is at present little integration of QoS constraints in object-based distributed architectures. The topic does not seem to be on the agenda of OMG. QoS is mentioned in the ODP drafts [4, 5], and some ODP-based work can be found in [11] Part 5. Distribution support for object-based architectures will depend on the abstract model used, but one can hope

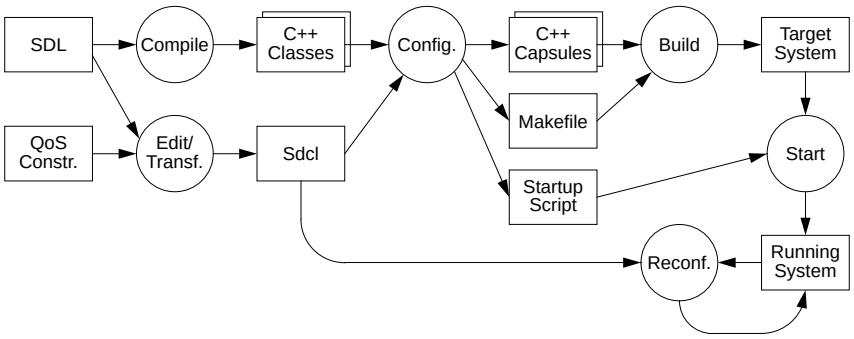

Figure 2: The distribution configuration tool chain.

to achieve methods and tools independent of the concrete platforms implementing those architectures.

While work on QoS is lacking, there is a body of work on *configuration* of distributed systems (see [6] for a collection of recent reports on this topic). However, most of this work only deals with the *logical* configuration of system components, i.e., it concerns configuration at the Computational Model level. Distribution configuration must also handle *engineering-level* concerns, such as the specification of physical distribution of components in the system. This aspect becomes more prominent in ODP-based systems, since the ODP Engineering Model defines comprehensive structuring concepts (see Section 2). The components of a distributed application have to be mapped to the physical nodes of the infrastructure and to be structured according to these engineering concepts.

Most existing work on configuration of distributed systems is based on a modules/ports model, where individual components (*modules*) are encapsulated and send and receive messages to and from local *ports*, i.e., they have no knowledge of other components in their environment. This approach leads to a complete separation between programming-in-the-small and programming-in-the-large. The former may use any programming language while the latter requires a special configuration language that interconnects the modules by binding the ports.

The ODP Computational Model [4, 5] is different in that the objects have knowledge (references to interfaces) of other objects external to them and communicate with these objects directly by invoking operations on known interfaces. The acquisition of an interface reference establishes a potential binding. Thus, the information comprising the programming-in-the-large aspects is interspersed among the individual components of an ODP application. While the modules/ports model assumes centralized control, ODP is based on autonomous objects and distributed control. This supports openness, and also makes dynamic reconfiguration easier.

In summary, we can make the following observations regarding distribution configuration in ODP:

- Logical composition (programming-in-the-large) is not the central focus, since this information is dispersed in the objects.
- The central focus of ODP configuration is the mapping of computational objects to the engineering infrastructure.
- ODP is strong on modeling open, dynamically reconfigurable systems, but some means is needed to control this freedom.

These observations form the basis of our work described in this paper.

4 Distribution support in SCORE

The RACE II project SCORE,[1] in which the authors are involved, aims to establish environments for the creation of new generation telecom services. These services are distributed applications, and the ODP framework is a prime target architecture. Hence, the distribution support developed in the project is ODP-based, but similar support for, e.g., OMG-based applications would not be very different.

[1] RACE Project 2017, Service Creation in an Object-oriented Reuse Environment.

One of the main formalisms used in SCORE for system design is SDL, the ITU-T Specification and Description Language. In particular, the SDL-92 definition, with its object-oriented extensions, is the focus of much of the novel work in the project. Since SDL allows detailed specification of system behaviour, it is possible to generate code automatically from SDL diagrams. SCORE is developing a translator from SDL-92 to C++. The SDL input to this translator has to adhere to certain guidelines to narrow the semantic gap between SDL and ODP ([11] Part 5). The output code conforms to ODS, a pilot ODP implementation.[2]

Thus, distribution configuration support developed in SCORE will have to deal with how to map SDL processes (the SDL "objects") to ODP engineering concepts. We have developed a *distribution configuration language* (Sdcl) for expressing this type of information, along with tools for mapping an application onto an ODP-based platform (such as ODS) according to a Sdcl specification. Figure 2 illustrates the tool chain.

SCORE aims at developing support for multimedia services. This type of service implies strict performance and reliability requirements, which are expressed mainly in terms of different kinds of quality of service (QoS) constraints [2]. An ideal tool environment would therefore contain means for deriving Sdcl configuration descriptions from QoS requirements, as shown in Figure 2. Similarly, to deliver the required performance, whenever a system is subjected to dynamic reconfiguration, or when the QoS constraints change, the configuration should be validated against the QoS constraints.

Translating QoS constraints into an engineering specification is a complex problem, however, calling for heuristics and knowledge-based techniques. This is outside the scope of our work. We have focused on developing a specification language that can be used for expressing a distribution configuration, be it generated by humans or an intelligent front-end tool.

5 An ODP-based distribution configuration language

Distribution configuration deals with how a logical system structure must be mapped to a corresponding engineering structure. For example, in SCORE we would like to map a SDL system specification to the ODP engineering concepts of nodes, capsules, clusters, and objects.

In this section, we list the main issues of concern, and we show via examples how our language, Sdcl, expresses these aspects.

A specification in Sdcl consists of a set of type definitions and the description of an initial configuration. The type definitions are based on the engineering concepts, such as capsules and clusters, and the description of the initial configuration assigns instances of the defined types to explicit physical nodes.

5.1 Type definition

Figure 3 depicts a SDL specification of a system[3] consisting of four process types—*UT*, *MT*, *ST*, and *DT*—along with their instances *user*, *main*, *sub*, and *dir*. There is one instance of each process type, except for *sub*, which may have from initially two up to five instances. The communication patterns are as indicated by the arrows.

Figure 4 (a) presents the capsule and cluster types to be used in the distribution configuration. It also lists the interfaces that each object class (SDL process type) supports and also those from other object classes that it uses (**provides** and **requires**). These **class** statements provide the link to the SDL specification. A cluster type is defined in terms of the types of its constituent objects (SDL process types). A capsule is in turn defined in terms of the types of the clusters and objects that it comprises, defining the object classes that the capsule can support at run-time.

As specified, there are two different capsule types that can contain instances of type *ST*. This can support, for example, reliability constraints (the capsule is the ODP unit of failure and protection).

5.2 Initial configuration

Figure 4 (b) shows an initial configuration of the system in terms of the structure of the nodes comprising it. The system is distributed over two computing nodes containing three capsules. Concrete instances of the appropriate types from the SDL specification are bound to the clusters and capsules. Since *sub* is a SDL process *set* consisting initially of two instances, we have to specify where each instance is to be located. Instances to be created dynamically are not part of the initial configuration of the system.

The client-server relationships (**uses** clauses) are included, since configuration tools may need them.

[2] The Open Distributed Systems platform from BNR Europe Ltd. Another platform experimented with in SCORE is ANSAware from APM Ltd.

[3] Since SDL *system* and *block* concepts do not map to engineering structures, we have omitted them from the figure.

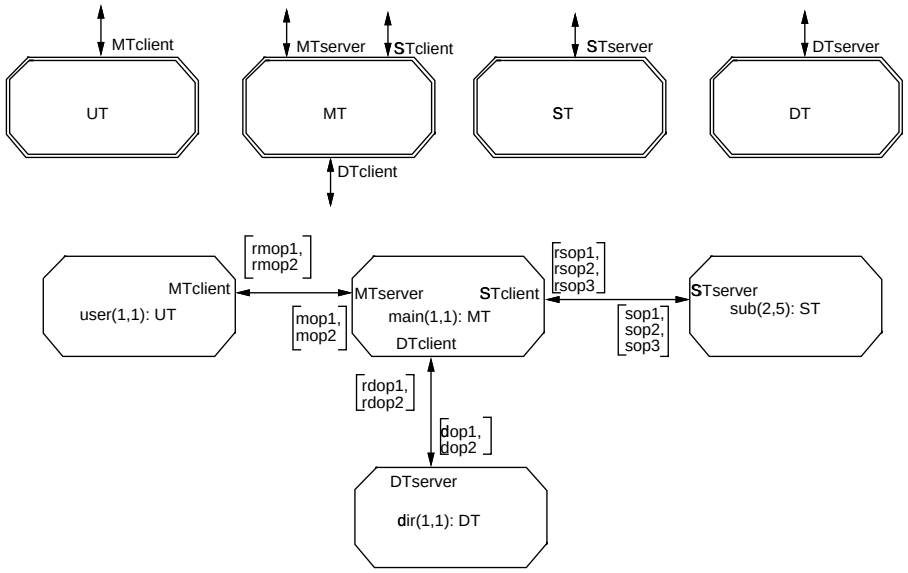

Figure 3: The example SDL-92 system.

These may actually be generated from SDL descriptions. In general, Sdcl code may either be written by hand, or it may be generated by various tools, such as a visual programming tool or a constraint resolution tool.

In addition to the basic structure requirements specified in Figure 4 (a), there may be many other engineering aspects we wish to express. We have focused on the following, depicted in Figure 4 (b):

Location

For each node in the distributed system, it is necessary to explicitly express its location, the concrete computer in the network on which it shall reside. This is taken care of by the **on** <*node*> construct. All capsules specified within node *nod1* will run on the given machine.

Clustering

The cluster is the ODP unit of migration, and it can therefore be used to specify collocation of objects. In the example, the *main* and *user* objects are assumed to have close interaction, and they are therefore clustered.

Instantiation vs. binding

When introducing a new system, it might be the case that instances of the types of some of the components in the system are already running in the network. An issue is thus whether to create a new instance of a component or to use an existing one. If an existing object is used, there is also a question of whether a specific instance is desired or any available instance will do. In Figure 4 (b), the *dir* object is specified to exist already, and no instance of this object will therefore be created. Instead, its client *main* will be bound to an instance already running in the distributed environment. If there is more than one appropriate instance, one is chosen (by the trader) based on the additional information given in the optional **with** clause, as shown.

5.3 Further extensions

Our work on distribution configuration is still in progress, and some of the areas we are currently considering are described below.

Dynamic reconfiguration

As mentioned above, one way to ensure consistency in the face of random changes is by constraints. The initial configuration should satisfy the constraints, and each change must be checked for consistency.

A Sdcl specification describes the initial configuration of a system. Later reconfigurations may render the system inconsistent with the description, but the system may still be valid with respect to the QoS constraints imposed on it. Essentially, a reconfiguration should result in a new Sdcl description that could be

```
class UT requires MTserver;
class MT requires STserver,
                  DTserver;
        provides MTserver;
class ST provides STserver;
class DT provides DTserver;

cluster clT is
      class MT;
      class UT;
end;

capsule capT1 is
      cluster clT;
      class ST;
end;

capsule capT2 is
      class ST;
end;
```

(a)

```
initial
    node nod1 on "holt.nr.no" is
        Cap1: capT1[
            Cl: clT [main: MT; user: UT];
            sub: ST
        ]
    end nod1;

    node nod2 on "rose.nr.no" is
        Cap2: capT2[sub: ST]
    end nod2;

    exists [
        dir: DT with "C=no; O=nr"]
    endexists;
endinitial

nod1.Cap1.Cl.user uses
    nod1.Cap1.Cl.main.MTserver;
nod1.Cap1.Cl.main uses
    nod2.Cap2.sub.STserver;
nod1.Cap1.Cl.main uses
    dir.DTserver;
```

(b)

Figure 4: Sdcl code for type definition (a) and initial distribution (b).

checked against the QoS constraints, just as the initial Sdcl description was (Section 4).

Performing this check may be quite complex. The QoS requirements on an application are stated at the Computational Model level, putting constraints on the computational interfaces of objects (see [11] Part 5 for an elaboration of this). It would help considerably if the engineering process could translate these constraints to *engineering-level constraints*, such as where to locate objects to ensure adequate performance, how many objects can be allowed on a computing node, whether to replicate objects for added reliability or performance, etc. Checking reconfigurations against these constraints, expressed in terms of the Engineering Model, may be expected to be relatively simple.

Engineering constraints may conveniently be included as an *invariant* part of the Sdcl specification. They would take the form of first order logic formulae and could contain functions such as:

$\mathsf{node}(x)$, $\mathsf{capsule}(x)$, $\mathsf{cluster}(x)$ – The node, capsule, or cluster of an object x.

$\mathsf{numcaps}(y)$, $\mathsf{numclus}(y)$, $\mathsf{numobjs}(y)$ – The number of capsules, clusters, or objects in a node or capsule y.

$\mathsf{card}(z)$ – The cardinality of a set z of instances of a given object type.

$\mathsf{type}(t)$ – The type of an object t.

A constraint for our example system could be:

$$\mathsf{cluster(main)} = \mathsf{cluster(user)} \\ \wedge\ \mathsf{node(user)} = \text{"holt.nr.no"} \\ \wedge\ \forall\, x, y \in \mathsf{sub},\ x \neq y : \mathsf{capsule}(x) \neq \mathsf{capsule}(y)$$

The need for constraints checking implies the existence of management objects that can query the objects in the system to verify the system state. (The ODP node manager could serve this purpose.) It also implies that the objects must provide a management interface. One interesting approach would be to generate automatically the management interface and operations for application objects, and to generate a management object that could maintain and update a Sdcl description of the system and periodically check it against the engineering constraints.

Note that the engineering-level constraints cannot be expected to be equivalent to the original QoS constraints, but they may provide a pragmatic approximation. There may thus be cases where one has to go back and check against the QoS constraints.

Groups

A *group* is a collection of objects that are related to each other, and that appear to other objects as a single entity. The relationship among the member objects of a group reflects its purpose. For example, a *replication* group could be used to support reliability and robustness. To satisfy requirements, the deployment of a new application may give rise to the creation of new groups, the inclusion of new member objects into existing groups, or simply using an existing group. In the case where groups are not supported by the distributed environment, the replication of some of the objects may be required.

Sdcl does not yet include support for groups as these are not part of the ODP standard (a proposed framework can be found in [1]), but it is an interesting topic for future investigation.

6 Discussion

6.1 The role of the ODP Engineering Model

As we have seen, the distribution configuration specification determines how the target system is to be structured in terms of ODP concepts, such as nodes, capsules, and clusters. It does not, however, have to determine the mapping of these concepts to a concrete platform, such as ODS or ANSAware. That step is taken care of by the translation tools. The ODP Engineering Model provides the abstract model that makes this *platform-independent targeting* possible. Without a target model, such as ODP, platform independence extends only to the computational model (SDL) level.

6.2 The role of SDL

In the SCORE tool chain (Figure 2), systems are specified and designed in SDL-92, and C++ code is automatically derived from the specification. Defining the application at such a high level is essential for platform independence. Code may be generated from the same SDL description for many different target platforms. Writing object code files in C/C++, on the other hand, makes platform details visible.

There are other benefits from using a well-defined high-level language as well. For SDL there exist powerful tools for analysis and simulation of the system.

Note that SDL has many similarities with the modules/ports model mentioned in Section 3. In fact, SDL block diagrams can be thought of as a programming-in-the-large configuration language, with process diagrams supplying "object" functionality. Thus, in our case, SDL takes care of the logical configuration, leaving engineering configuration for Sdcl.

The fact that SDL offers another computational model than ODP, does, however, result in some "impedance mismatch" [7]. (One way to model ODP in SDL is documented in [11] Part 5.) The ideal situation would be to have a high-level language embodying the ODP Computational Model, strengthened with logical composition facilities. Since SDL here is used as an ersatz ODP CM language, we see that SDL process types and instances actually show up in the Sdcl language, where only ODP interface types and objects should have been present.

6.3 The telecom context

The work described in this paper is part of a telecom project, and a few remarks are in order to put it in that context.

The above discussion of the relation to QoS requirements (Section 5.3) assumes that there is no other system interfering in the target environment. This is typically the case in dedicated real-time and control systems, and even in some business environments. In telecom, however, a new system (a "service") is downloaded in a network already in operation, and the engineering constraints must therefore take into account the current state of the network. The management objects must merge a network state specification with the engineering constraints when checking for configuration validity. Reconfigurations may actually be performed as a result of changes in the network and with the goal of maintaining QoS, rather than a desired modification triggering a QoS check. This adds further complexities to the problem of dynamic system reconfiguration.

In telecom, *creation* and *management* of distributed applications tackle, respectively, the static phase and the dynamic phase of an application's lifetime. These domains are not independent of each other; each relies on information from the other phase. For example, management activities need to know about the type of the objects they manage, the logical structure of the applications, and the constraints governing the objects and applications. Likewise, the creation phase, especially during targeting (code generation and deployment), needs to know about the structure of the network, the characteristics of its nodes, etc. Targeting is the activity on the creation side which handles the transition from the static to the dy-

namic domain, and the work described in this paper thus supplies a link between these domains.

6.4 Status

As of this writing, a tool for a previous version of Sdcl has been implemented that transforms a set of individual object files written for ANSAware (which is C-based) to capsule files corresponding to the Sdcl code. It also generates a makefile and startup script. We will move our work to the ODS platform (C++-based) and integrate it with the other tools of SCORE. The work will continue for 1995, and we plan to look at how dynamic reconfiguration driven by QoS constraints can be better supported.

7 Conclusions

This paper has argued that, when developing distributed applications based on the new generation object-based open architectures and standards, rigorous support for distribution configuration is needed. This is necessary for managing the performance requirements on these complex systems, and for reigning in some of the freedom of, for example, ODP, whose underlying model is basically anarchist.

To this end, a distribution configuration language (Sdcl) based on the ODP Engineering Model has been designed, and we have presented the notation through an example of its use. When coupled with a high-level design language, such as SDL, Sdcl allows platform-independent targeting onto distributed architectures, thus contributing to the transparency offered by such architectures.

Sdcl expresses a static system configuration, but we have shown how it can form the basis of a constraint-oriented approach to system reconfiguration, and we have discussed some implications this will have on the field of system management.

Acknowledgements

This work has been supported by the Research Council of Norway and the CEC through the RACE II project 2017 SCORE (Service Creation in an Object-oriented Reuse Environment). This paper represents the view of its authors.

References

[1] ANSA. A Model for Interface Groups. Technical Report AR.002.00, Architecture Projects Management Ltd., Cambridge, UK, August 1991.

[2] L. Hazard, F. Horn, and J.-B. Stefani. Notes on Architectural Support for Distributed Multimedia Applications. Technical Report CNET/RC.W01.-LHFH.001, ISA Project, March 1991.

[3] ISO/IEC JTC1/SC21/WG7. Basic Reference Model of Open Distributed Processing. ISO/IEC 10746.

[4] ISO/IEC JTC1/SC21/WG7. Basic Reference Model of Open Distributed Processing – Part 2: Descriptive Model. ISO/IEC 10746-2.

[5] ISO/IEC JTC1/SC21/WG7. Basic Reference Model of Open Distributed Processing – Part 3: Prescriptive Model. ISO/IEC 10746-3.

[6] J. Kramer, editor. *Proc. Int'l Workshop on Configurable Distributed Systems.* The Institution of Electrical Engineers, U.K., 1992.

[7] S. Mazaher and G. Raeder. SDL and Distributed Systems—a Comparison with ANSA. In O. Faergemand and A. Sarma, editors, *SDL '93, Using Objects. Proc. 6th SDL Forum.* North-Holland, 1993.

[8] J.R. Nicol, C.T. Wilkes, and F.A. Manola. Object Orientation in Heterogeneous Distributed Computing Systems. *IEEE Computer*, 26(6), June 1993.

[9] OMG. Object Management Architecture Guide. Object Management Group, Inc., Framingham, MA, U.S.A., November 1990.

[10] OMG. The Common Object Request Broker: Architecture and Specification. Object Management Group, Inc., Framingham, MA, U.S.A., 1991.

[11] SCORE-Methods and Tools. Report on Methods and Tools for Service Creation (First Version). Deliverable D203, R2017/SCO/WP2/DS/-P/027/b2, RACE project 2017 (SCORE), December 1993.