Quality-of-Service Directed Targeting Based on the ODP Engineering Model

Georg Raeder and Shahrzade Mazaher

NR – Norwegian Computing Center
P.O.Box 114 Blindern, 0314 Oslo, Norway
Georg.Raeder@nr.no, Shahrzade.Mazaher@nr.no

When developing large distributed systems, the final transformation steps toward executable components (targeting) are usually complex and platform dependent. The ODP Engineering Model provides a framework for alleviating this situation, but methods and tools that realize its potential are still lacking. This paper gives an example of a *distribution configuration language* and shows how it can be used, together with SDL, to achieve platform-independent distributed targeting. Furthermore, targeting decisions should be made on the basis of quality-of-service constraints, such as performance and reliability requirements, and the paper shows how our language can serve as a basis for stating and checking such constraints. Finally, the relation of our work to dynamic reconfiguration is discussed.

Keywords: Distribution configuration, configuration languages, distributed targeting, quality of service, open distributed processing.

1. INTRODUCTION

Targeting is a complex issue in distributed systems. Targeting involves all the steps taken to transform platform-independent application code to software components that can be executed in a concrete environment. While this is a relatively trivial task for a monolithic program, targeting distributed systems involves non-trivial choices and trade-offs as to how the system is to be split into distributed components, achieving the desired performance, reliability, etc.

Thus, methods and tools have to be developed for *distribution configuration*, allowing system implementors to express how a system is to be partitioned and assigned to platform entities, such as computers and operating system processes.

There are many degrees of freedom in such a configuration. To control it, a framework of *quality of service* (QoS) must be developed so that end-user requirements can be readily translated to technical constraints that will guide the distribution configuration.

To make these steps possible in more than an ad hoc manner, we need a *model* of the target environment, i.e., an abstract framework that 1) will help us tie together the various concepts of applications, target platforms, and end-user requirements, and 2) will do so in a manner independent from concrete platform implementations.

This paper presents a language for distribution configuration, and it also describes an approach to using the language for checking that QoS constraints are satisfied. The language is based on the Engineering Model of the Open Distributed Processing (ODP) draft international standard [6] as its abstract framework.

The paper assumes familiarity with ODP. Knowledge of SDL, the ITU-T Specification and Description Language, would also be helpful, but is not essential.

1.1. Configuration of distributed systems

Most contemporary work on configuration of distributed systems (see [7] for a collection of recent reports on this topic) deals with *logical* configuration of system components. In terms of ODP, this concerns configuration at the Computational Model level. Typically, these approaches are based on a modules/ports model, where individual components (*modules*) are encapsulated and send and receive messages to and from local *ports*. Configuration consists of creating and deleting instances of these modules and connecting their ports to each other. This is an example of programming-in-the-large, as distinct from the programming-in-the-small of each module.

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. The components of a distributed application have to be mapped to the physical nodes of the infrastructure and they have to be structured according to these engineering concepts.

Configuration of distributed systems may be *static* (done "off-line"). Executables are typically generated statically, and an initial distribution configuration may also be specified statically. Dedicated systems that do not evolve may in fact have their engineering configuration statically determined.

In many cases, however, the system needs to change *dynamically*. Functional (logical) changes, in terms of new modules being instantiated and connected, or parts being shut down, can often be pre-specified in terms of change scripts (see, e.g., [1, 3, 8]). Non-functional changes, such as reconfigurations to counter load imbalances, usually cannot be anticipated in this way. *Management* objects could be supplied for this purpose, with each application object providing a corresponding management interface (a form of this is presented in [5]). A major challenge is how one can keep a system consistent according to some criteria in the face of ad hoc changes. One would like to describe an initial, consistent configuration, and then make sure that every reconfiguration operation takes the system from one consistent state to another, for example by means of checking *constraints* that always must hold [2, 3].

Engineering configuration in ODP belongs to the dynamic category, since the state of the network in an open system typically will vary in a non-deterministic manner. Achieving a desired level of quality of service must therefore be subject to continual evaluation and reconfiguration. Introducing this type of management in ODP seems to imply more centralization (via management objects), but it is nevertheless very important for achieving performance and reliability.

1.2. Quality of service

Quality of service (QoS) encompasses a wide range of essentially non-functional requirements that users may have on a system, such as timeliness (delay bounds, jitter bounds), throughput (bandwidth), cost, dependability, security, etc. ([12]; [10] Vol. 5). As the system design is refined into components, so will the QoS requirements be broken down into constraints on individual components, such as ODP Computational Model objects, and on the engineering infrastructure that supports them. There may also be a translation from user-level requirements, such as the quality perception of moving images, to technical constraints, such as a minimal number of frames per second.

The current ODP standards documents refer to QoS, but without detailing the subject. In [12] a framework is proposed for how QoS can be integrated in the ODP Computational Model

(CM). Briefly, interface definitions are extended with

- QoS clauses that express the quality of service *required* by this interface type from its environment,
- QoS clauses that express the quality of service *provided* by this interface type to its environment, given that its requirements on the environment are fulfilled, and
- a list of the types of interfaces that implementations of this interface type may invoke operations on.

The QoS clauses are expressed in a language based on first-order logic. The last item above is necessary when performing QoS analyses, which essentially consist of checking whether the QoS provided by servers match that required by their clients. This gives rise to an extended type conformance relation.

These CM-level QoS constraints must be further interpreted in terms of the supporting infrastructure. Whether an object can provide the level of QoS specified depends on its engineering and technology support. Section 2.3 of this paper describes our approach to engineering-level QoS analysis and its role in guiding distribution configuration.

1.3. The context

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.

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 behavior, it is possible to generate code automatically from SDL diagrams. SCORE is developing a translator from SDL-92 to C++ [11]. The SDL input to this translator has to adhere to certain guidelines to narrow the semantic gap between SDL and ODP ([10] Vol. 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 designed a *distribution configuration language* (Sdcl) for expressing this type of information. We are currently prototyping tools for mapping an application onto an ODP-based platform (such as ODS) according to a Sdcl specification. Figure 1 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 QoS constraints [4]. An ideal tool environment would contain means for automatically deriving Sdcl configuration descriptions from QoS requirements. 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.

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

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

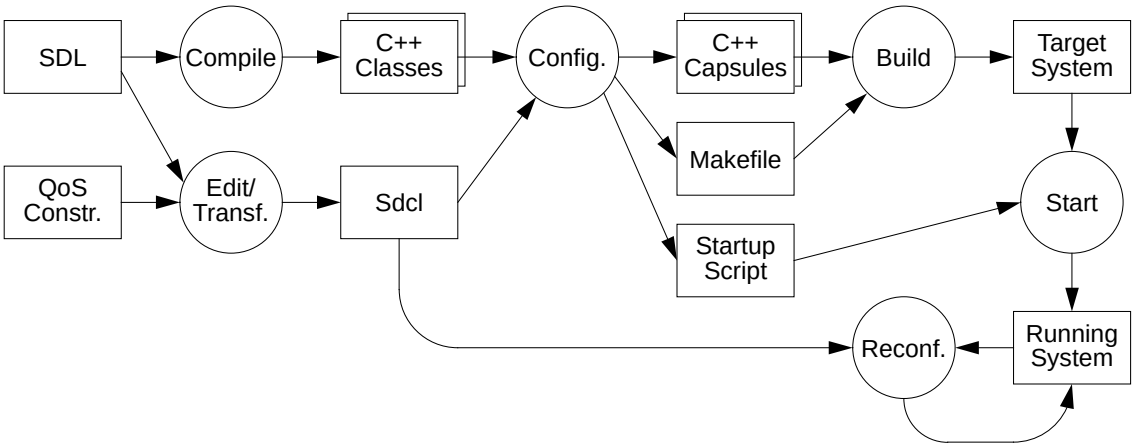

Figure 1. The distribution configuration tool chain.

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.

1.4. The approach

Our goals are 1) to build practical tools to automate the task of configuring distributed systems, and 2) to include QoS concerns in this tool support. We can split the task of configuring a particular system as follows:

1. Specify the QoS requirements and provisions of the interfaces of each object class in terms of engineering-level constraints.
2. Specify capsule files, i.e., the executables that should be generated and which object classes they should have compiled in.
3. Specify an initial configuration, indicating the capsules to start, where to start them, and which object instances they should initially contain, and check it against the QoS constraints.
4. Create a management object that maintains the QoS specifications of the system, and that can check requests for reconfigurations against these constraints.

Steps 1–3 involve *static* specifications that can be processed off-line. This paper mainly concerns these aspects. The management object in step 4 can also be statically generated, but the object itself handles requests for *dynamic* reconfigurations, and must make decisions based on QoS requirements, current system configuration, and the current platform status (node loads, network traffic, etc.). The following sections detail our contributions covering these steps.

2. AN ODP-BASED DISTRIBUTION CONFIGURATION LANGUAGE

Distribution configuration comprises several kinds of information. In this section, we will list the main issues of concern, and we will show via examples how our language, Sdcl, expresses these aspects.

2.1. Specifying capsule files

A logical system structure must in some way 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. This is the core task of distribution configuration, and it should therefore form the central feature of a notation for this purpose.

To this end, our language supports the definition of types based on the engineering concepts, and that of an initial configuration in terms of the assignment of instances of the defined types to explicit physical nodes:

- We interpret ODP capsule and cluster definitions as *templates* used for code generation. The modules generated from these templates can be instantiated several times and in several different places in the target system.
- ODP nodes, on the other hand, correspond to physical system entities. A node description therefore gives a snapshot of part of the system, in terms of the capsules and clusters instantiated on it. In particular, a set of *initial* node configurations can be used for starting up the system.

Assume that we have the SDL-92 design shown in Figure 2.[3] The figure depicts a system[4] consisting of five process types—*Customer*, *Subscriber*, *Terminal*, *Session*, and *Database*—along with their instances *cu*, *sb*, *tr*, *sn*, and *db*. There is one instance each of *Customer* and *Database*, whereas the other three types may have any number of instances. The communication patterns are as indicated by the arrows (dotted arrow means process creation).

Figure 3 (a) presents the capsule and cluster types used in the distribution configuration. It also specifies the interfaces that each object class (i.e., SDL process type) supports and 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[5] that it comprises, defining the object classes that the capsule can support at run-time. For example, objects of type *Session* can be instantiated in two different capsule types.

2.2. Specifying an initial configuration

Figure 3 (b) shows an initial configuration of the system in terms of the structure of the nodes comprising it. For exposition, we have instantiated two subscribers and their terminals. We have distributed the system over one central computing node (containing one capsule) plus one node for each terminal (containing a capsule with the terminal object plus an empty session capsule).

[3]The example is derived from a Virtual Private Network (VPN) test service elaborated in SCORE. Briefly, the *Customer* represents the service, aided by a *Database* of system information. *Subscriber* handles subscriptions and sets up calls. A *Terminal* is instantiated for each connected terminal. A *Session* is instantiated for each call.

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

[5]An object class directly within a capsule is a shorthand for a cluster containing only that class.

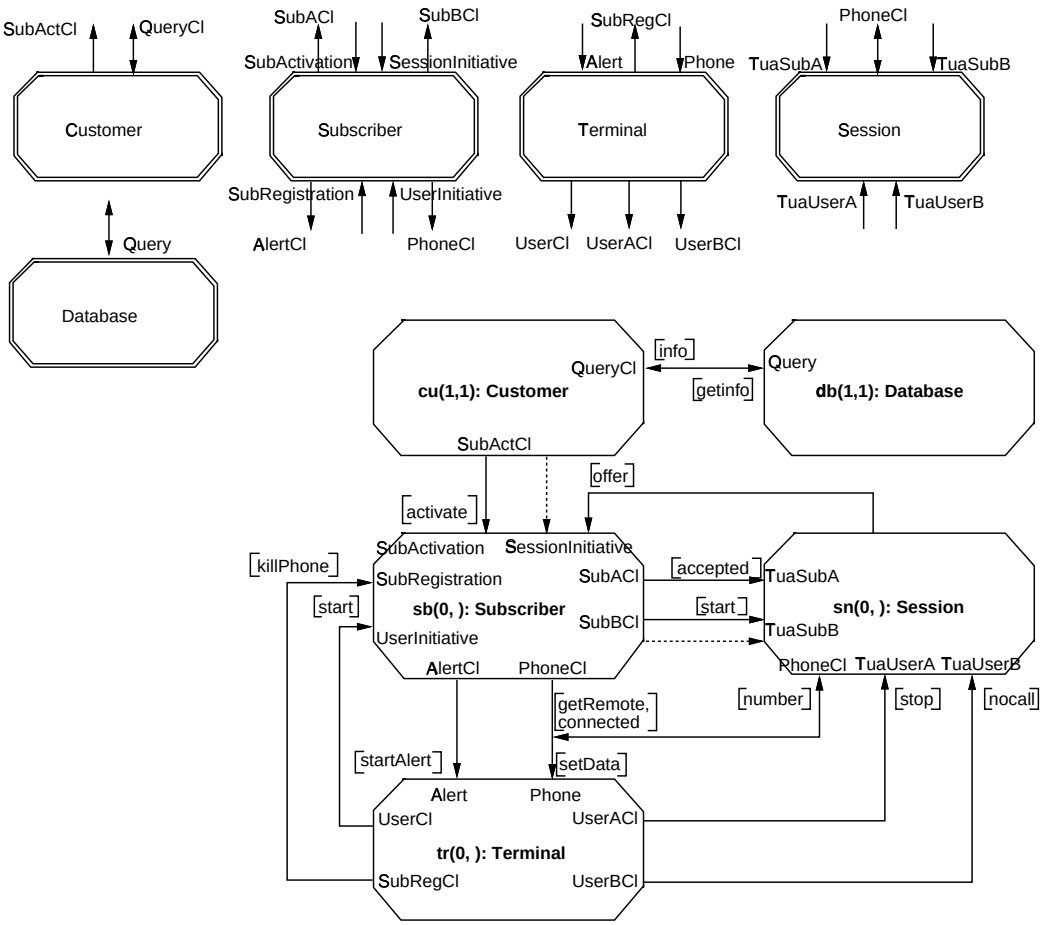

Figure 2. The example SDL-92 system.

Those instances that are 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. These may actually be generated from the SDL specification. 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, there may be many other engineering aspects we wish to express. We have focused on the following aspects, also depicted in Figure 3 (b).

Location

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

```
class Customer
    requires SubActivation, Query;
class Subscriber
    requires Phone, Alert, TuaSubA, TuaSubB;
    provides SubActivation, SubRegistration,
        UserInitiative, SessionInitiative;
class Terminal
    requires TuaUserA, TuaUserB
        SubRegistration, UserInitiative;
    provides Phone, Alert;
class Session
    requires Phone, SessionInitiative;
    provides TuaSubA, TuaSubB,
        TuaUserA, TuaUserB;
class Database
    provides Query;

cluster CustSubCls is
    class Customer;
    class Subscriber
end

capsule CustomerCap is
    cluster CustSubCls;
    class Session
end

capsule TerminalCap is
    class Terminal;
    class Session
end
```

(a)

```
initial
    node vpn on "vpn.nr.no" is
        CustSub: CustSubCap[
            Cs: CustSubCls[
                cu: Customer; sb(1:2): Subscriber]]
    end

    node ola on "ola.nr.no" is
        Term: TerminalCap[tr: Terminal];
        SessionMgr: SessionCap[]
    end

    node per on "per.nr.no" is
        Term: TerminalCap[tr: Terminal];
        SessionMgr: SessionCap[]
    end

    exist
        db: Database with "C=no; O=nr"
    end
end

vpn.CustSub.Cs.cu
    uses vpn.CustSub.Cs.sb(1).SubActivation;
vpn.CustSub.Cs.cu
    uses vpn.CustSub.Cs.sb(2).SubActivation;
vpn.CustSub.Cs.cu uses db.Query;
vpn.CustSub.Cs.sb(1) uses ola.Term.tr.Alert;
vpn.CustSub.Cs.sb(1) uses ola.Term.tr.Phone;
ola.Term.tr uses vpn.CustSub.Cs.sb(1).SubRegistration;
ola.Term.tr uses vpn.CustSub.Cs.sb(1).UserInitiative;
(Analogous clauses for "per" as for "ola")
```

(b)

Figure 3. Sdcl code for type definition (a) and initial distribution (b).

Instantiation vs. binding

When introducing a new system, it might be the case that instances of the same types as some of its components are already running in the network. An issue is thus whether to create a new instance 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 3 (b), the *db* object is specified to exist already, and no instance of this object will therefore be created. Instead, its client *cu* 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.

2.3. QoS constraints and dynamic reconfiguration

The distribution configuration of an application must satisfy the QoS (non-functional) requirements imposed on it. As the system is elaborated through design and implementation, the original user-level requirements must be broken down into more specific constraints. At the computational level, QoS requirements are stated by putting constraints on the computational interfaces of objects ([12]; [10] Vol. 5). During implementation, these constraints should be further translated into *engineering-level constraints*.

Engineering constraints would handle aspects such as the location of objects, whether to replicate objects for added reliability or performance, performance of channels connecting the objects of the application, etc. Checking configurations against these constraints, expressed in terms of the supporting infrastructure, may be expected to be relatively simple.

Engineering constraints may conveniently be included as an *invariant* part of the Sdcl specification. An initial configuration of the system should satisfy these constraints. Furthermore, a (dynamic) reconfiguration should essentially update the Sdcl description (the initial configuration part) that could then be checked against the invariant. The capsule/cluster class part of the Sdcl description is used for capsule generation to *facilitate* that appropriate reconfiguration is possible (capsules with the proper contents must be available).

The constraints would take the form of first order logic formulae, and could contain functions such as:

node(x), capsule(x), cluster(x) – The node, capsule, or cluster of an object x.

numcaps(x), numclus(x), numobjs(x) – The number of capsules, clusters, or objects in a node or capsule x.

type(x) – The type of an object x.

proc(x) – A performance factor, given as the time taken to complete a specific computation on the infrastructure of object x.

comm(x,y) – A performance factor, given as the time taken to complete a specific communication between the objects x and y.

A set of constraints for our example system could be:

node(cu) = "vpn.nr.no"
$\forall$ x $\in$ Subscriber : cluster(x) = cluster(cu)
$\forall$ y, z $\in$ Session, y $\neq$ z : capsule(y) $\neq$ capsule(z)
10 $\geq$ 4·proc(tr) + proc(sb) + 2·proc(sn) + comm(tr,sb) + comm(tr,sn)

The first constraint ties the object *cu* to a physical node, i.e., the object has to permanently be placed on that node (due to security reasons, for example). The second constraint implies collocation of objects (which is directly reflected by the *cluster* construct of Sdcl). The third constraint expresses the condition that no two objects of type *Session* should be placed in the same capsule (e.g., for reliability reasons).

The last constraint expresses a relation between the performance factors of the potential nodes and capsules where the objects would be placed and of the communication channels between them. Consider Figure 4, which is a partial ODP realization of the system in Figure 2. Assume that there is a requirement from the environment (via an interface *Call* not shown in Figure 2)

on *tr*, stating a maximum response time of 10ms for invocation of a certain operation. Assume further that executing this operation uses 4 processing units in *tr* plus one invocation on each of *sb* (delay *x*) and *sn* (delay *y*). The executions of the latter two operations use 1 and 2 processing units, respectively. Adding communication delays, we get:

$$10 \geq 4 \cdot proc(tr) + x + y \wedge x \geq proc(sb) + comm(tr,sb) \wedge y \geq 2 \cdot proc(sn) + comm(tr,sn)$$

or:

$$10 \geq 4 \cdot proc(tr) + proc(sb) + 2 \cdot proc(sn) + comm(tr,sb) + comm(tr,sn)$$

All the *proc* and *comm* parameters should be obtained from a database of network information and from the Sdcl description that states the clusters, capsules, and nodes within which the objects are contained. The objects *tr*, *sb*, and *sn* should be distributed over the network such that the corresponding values for *proc* and *comm* parameters satisfy the above formula. For example, this analysis will uncover whether *tr* can fulfill its requirements when using an *sn* in another capsule, or whether only the one in its own capsule is fast enough.

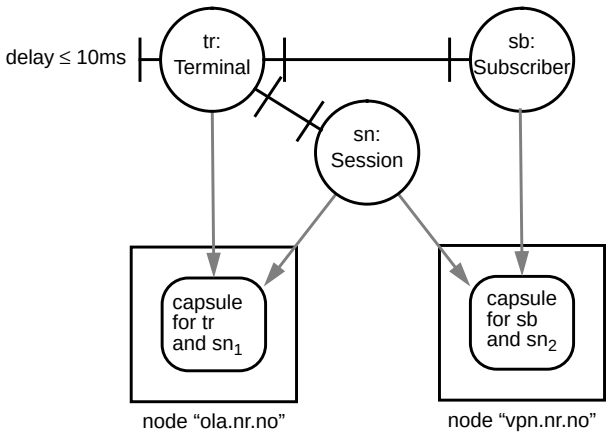

Figure 4. Engineering constraints.

Thus, the expression of the QoS of the system will involve a system of equations which cannot be solved until a concrete platform is chosen. The external system requirements (the 10ms in the example) provide boundary conditions on the equations. In theory, one could automatically derive a configuration satisfying the equations, but in practice, one will manually propose a configuration (using Sdcl) and check that it satisfies the equations (possibly having to go through some iterations).

Many QoS characteristics can be expressed in a manner similar to this response-time example, but the combination rules for the equations for, e.g., a probabilistic dependability measure, would vary. The analysis is crude, and it must be based on average or worst case figures, but we conjecture that automated tools supporting even this level of analysis would be quite useful in complex systems.

The need for constraints checking implies the existence of management objects that can query the objects in the system to verify the system state, and access a database of network information for relevant data. (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 the configuration part of it against the specified engineering constraints.

3. CONCLUSIONS

This paper has presented a language for specifying the distribution configuration of a system, for targeting onto a platform conformant to ODP. Realizing that the major factor influencing this kind of configuration is the QoS requirements of the system, we also described how the language can be used as a basis for QoS-directed targeting. Finally, we outlined an approach to managing the dynamic reconfiguration of the system based on the initial configuration and a set of engineering-level constraints derived from QoS requirements.

3.1. The role of the ODP Engineering Model

As we have seen, the distribution configuration specification defines 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 handle 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 such an abstract target model, platform independence extends only to the computational model (SDL) level.

3.2. The role of SDL

In the SCORE tool chain (Figure 1), 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 1.1. In fact, SDL block diagrams (such as Figure 2) can be thought of as a programming-in-the-large configuration language, with process diagrams supplying programming-in-the-small 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" [9]. (One way to model ODP in SDL is documented in [10] Vol. 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.

3.3. 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 the new version of Sdcl, and integrate it with the other tools of SCORE. The work will continue through 1995.

3.4. Further work

In order to achieve predictable, consistent, and appropriate quality of next-generation systems and services in open distributed environments, a QoS-directed development approach is needed. To this end, an ODP-based framework (methods and tools) for stating QoS constraints and carrying them through analysis, design, and implementation into engineering-level properties should be developed. These properties should guide the generation of service implementations so that they meet the QoS constraints (e.g., by selecting proper library components and choosing an appropriate distribution of components in the network). The properties could also be used in the management domain to control dynamic reconfigurations of services.

The design of a distribution configuration language, and its use in conjunction with engineering-level QoS constraints, as presented in this paper, is one step in this direction.

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). We particularly thank our SCORE colleagues from PTT Research (The Netherlands) and Tele Danmark Research (Denmark) for the stimulating collaboration on ODP targeting. This paper represents the view of its authors.

REFERENCES

1. M.R. Barbacci, C.B. Weinstock, D.L. Doubleday, M.J. Gardner, and R.W. Lichota. Durra: a Structure Description Language for Developing Distributed Applications. *Software Engineering Journal*, 8(2), March 1993.
2. T. Coatta and G. Neufeld. Configuration Management via Constraint Programming. In *Proc. Int'l Workshop on Configurable Distributed Systems*. IEE, U.K., 1992.
3. M. Endler and J. Wei. Programming Generic Dynamic Reconfigurations for Distributed Applications. In *Proc. Int'l Workshop on Configurable Distributed Systems*. IEE, U.K., 1992.
4. 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.
5. C. Hofmeister, E. White, and J. Purtilo. Surgeon: a Packager for Dynamically Reconfigurable Distributed Applications. *Software Engineering Journal*, 8(2), March 1993.
6. ISO/IEC JTC1/SC21/WG7. Basic Reference Model of Open Distributed Processing – Part 3: Prescriptive Model. ISO/IEC 10746-3.
7. J. Kramer, editor. *Proc. Int'l Workshop on Configurable Distributed Systems*. The Institution of Electrical Engineers, U.K., 1992.

8. J. Magee, N. Dulay, and J. Kramer. Structuring Parallel and Distributed Programs. *Software Engineering Journal*, 8(2), March 1993.
9. 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.
10. 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.
11. SCORE-METHODS AND TOOLS. Report on Methods and Tools for Service Creation (Final Version). Deliverable D204—R2017/SCO/WP2/DS/P/028/b2, RACE project 2017 (SCORE), December 1994.
12. J.-B. Stefani. Computational Aspects of QoS in an Object-based Distributed Architecture. In *Proc. 3rd International Workshop on Responsive Computer Systems*, 1993.