DSoS

IST-1999-11585

Dependable Systems of Systems

Revised Version of DSoS Conceptual Model (IC1)

Technical Report CS-TR-746, University of Newcastle upon Tyne

Report N° 35/2001, Technical University of Vienna

LAAS-CNRS Report No. 01441

Report Version: Deliverable IC1

Report Preparation Date: 23 Oct 2001

Classification: Public Circulation (after review)

Contract Start Date: 1 April 2000 **Duration:** 36m

Project Coordinator: Newcastle University

Partners: DERA, Malvern — UK; INRIA — France; CNRS-LAAS — France; TU Wien — Austria; Universität Ulm — Germany; LRI Paris-Sud - France

Project funded by the European Community under the Information Society Technology Programme (1998-2002)

Table of Contents

Revised Version of DSoS Conceptual Model

Cliff Jones[1], Marc-Olivier Killijian[3], Herman Kopetz[2],
Eric Marsden[3], Nick Moffat[4], Michael Paulitsch[2],
David Powell[3], Brian Randell[1], Alexander Romanovsky[1], Robert Stroud[1]

[1]University of Newcastle upon Tyne (UK),
[2]Technical University of Vienna (Austria),
[3]LAAS-CNRS (Toulouse, F), [4]QinetiQ (UK)

1 INTRODUCTION

This document defines the key concepts underlying DSoS. Before coming to their definitions, it is worth emphasising the breadth of systems and issues that the project is addressing.

1.1 Scope of DSoS

There are different ways of building systems: at one extreme there are "green fields" projects where a whole system is constructed from scratch; at the other extreme, systems can be constructed mainly from (large) existing components. It is the objective of the DSoS Project to investigate issues related to the integration of components that are existing complete *systems* in order to generate a new set of dependable services from the resulting *system of systems*. Emphasis is put on systems *of systems* because the latter will typically be non-trivial systems in their own right. This is in distinction to the construction of a system from more-or-less basic components with simple, fixed, interfaces that are fully under the control of the designer of the required system.

Clearly, building a system of systems is a recursive idea in that the required system could be a component of a yet larger system.

One key attribute of *component* systems is that they will normally exist before the design of the required system. Moreover, a *component system* is, typically, an autonomous computer system that provides a useful service to an organization or a set of users. A system of systems may thus span different organizations, each one with their own systems that are, as a result, in different spheres of management control.

The principal problem to be addressed by the DSoS Project is that of ensuring that the result of such integration is an adequately *dependable* system of systems. This objective remains

even when the component systems are less dependable: ways of masking failures in the underlying systems are to be addressed.

A further important requirement of such integration is that the stability of any existing services of the component systems not be compromized.

Furthermore, it is typically not possible for the designer of a system of systems to change the component systems. It is, however, important to recognize that those component systems might continue to evolve without consultation. One potential failure mode of a system is the result of a change to interfaces in its (separately controlled) components. The DSoS project has decided to include within its scope attempts to recover from such failures.

1.2 Breadth of this conceptual model

There does, of course, exist literature on building systems from components. Indeed, in specific domains, members of the DSoS project have made earlier contributions to such approaches. For example, the Time Triggered Architecture approach from TUV[1] is widely recognized as a key contribution to the design of real-time control systems.

What makes the DSoS project goals challenging (and worthwhile) is the decision to tackle a wide class of systems (of systems). The aim of this evolving document is to define a collection of concepts which can embrace, for example, inter-organizational on-line systems *and* real-time control (systems of) systems. The concepts will have to be abstract enough to cover failures as different as timing mismatches in actuators for a car braking system and interface mismatches when a change is made to a component system (not under our control) .

While it is unlikely that a single fault-tolerance approach can be found to attempt to contain all failures, only the identification of the underlying similarities (and the residual differences) can unify the design of systems of systems. Although the different viewpoints bring their own concepts and terminology, it is seen as a key objective of the DSoS *Conceptual Model* to analyse and unify concepts where possible.

There is another potential pay-off of this broad objective: there is real scope for cross-fertilization. For example, the concept of time has been extensively studied in vehicle control systems but has been treated as an afterthought in too much of the rest of computing. Similarly, concepts of ownership and management control are more familiar to the designers of large institutional systems. The DSoS project aims to get synergy from its broad aims.

[1] Technical University of Vienna

1.3 Some pre-definitions

1.3.1 A first look at the notion of "system"

A precise characterization of the concept of system is one of the objectives of Section 3.1 but an intuitive notion will suffice to set the scope of the discussion. A system is normally a collection of components whose behaviour can be discussed by fixing a boundary and its interfaces. Although a system such as a car can be viewed in different ways by its driver and a maintenance mechanic, there is for either purpose a system view which facilitates discussion. Interaction with a system takes place at interfaces which can usefully be thought of as being at the boundary of the system. The concern in DSoS is with systems *of systems* and this brings the key issue of mismatches between interfaces but, before addressing this, it is worth looking more closely at what it means to characterize the operations available at an interface.

Most non-trivial systems possess a state which captures some aspects of the history of interaction with the system (various notions of state are discussed in Section 3.2 below). In the trivial example of a *stack* the state can be viewed as a sequence of values; in a hotel database, the state would include all future bookings. In both cases, interactions at the interface can reflect and influence the state. One could already employ a notion of time here but many computer scientists try to finesse this by implicitly indexing the state by the point in the sequence of operations performed at the interface. Whether or not this is a good idea, it can be seen to be wholly inadequate in the case of systems whose state evolves autonomously. If the interface of a system emits –for example– the temperature of a nuclear reactor, it is essential to discuss the time at which the interaction occurs. (It is also easy to make the case that, if many other systems are interacting with a hotel reservation system, the interactions must also be indexed by time (e.g., the offer of a reservation might be made at time t with a validity period of d)). This notion of time is so central that it is explored more thoroughly in Section 1.3.3.

1.3.2 Legacy systems and architectural style

We distinguish *legacy* component systems from two other types of component system, which in contrast will have been designed in accordance with the chosen architecture for a given system of systems. These are previously-developed *general purpose* components, and components that have been *specially developed* for a particular system of systems. The integration of the component systems is realized via a communication service across special connections, the *linking connections* between *linking interfaces* of the component systems. From the point of view of any linking interface, a component system's specification can be

reduced to the functional and temporal description of those services that are required for the integration, together with, ideally, a description of its dependability guarantees. From this point of view, the other (i.e., local) services of the component systems are not important.

We assume that every autonomous legacy system is developed according to its own rules and conventions concerning data representation, protocol choices, error handling, etc. We call the sum of these conventions the architectural style of the system. It is probable that any two legacy systems that are to be integrated will conform to different non-compatible architectural styles. Any difference in architectural style, something we call a *property mismatch*, could, unless dealt with, give rise to a failure at a linking connection (Garlan, Allen et al. 1995; Allen and Garlan July 1997). It is an important function of the linking connection to reconcile these architectural styles in order that the component systems can communicate without any property mismatch. Furthermore, it may be required that specified independent failure modes of component systems are tolerated by the system of systems, or that mechanisms are provided to increase the dependability of the system of systems. The implementation of these fault tolerance mechanisms is also in the scope of the linking connections.

The subject of interface mismatches is explored in detail below but it is worth emphasising that the concern in DSoS goes far beyond simple types of format mismatches. This partly results from the observation that the component systems of a SoS might be under separate management control. It will be necessary to cope with what might be termed *protocol mismatches* where two systems need to exchange a collection of information but have differing flows of control.

1.3.3 The key role of time

The focus of the conceptual model of the DSoS Project is on the linking interfaces of the component systems, and the linking connections that enable communication between these systems in order to generate the emerging services of the system of systems. The DSoS conceptual model differs from many other models of computation by the explicit inclusion of physical time. *Physical time* is needed if we are to reason about timely failure detection (in particular, of autonomous component systems), performance, and other real-time properties. This point of view is also taken by E. A. Lee in an excellent recent survey on embedded computer systems: "*Time has been systematically removed from theories of computation, since it is an annoying property that computations take time. 'Pure' computation does not take time, and has nothing to do with time. It is hard to overemphasize how deeply rooted this is in our culture. So called "real-time" operating systems have so little to go on that they often reduce the characterization of a component (a process) to a single number, its priority.*

Even most 'temporal' logics talk about 'eventually' and 'always' where time is not a quantifier, but rather a qualifier." (Lee 1999).

1.4 The inspiration from case studies

The problems of composing a system of systems take many forms, since there are many forms of system. For this reason, we have chosen to scope the problem by pursuing case studies which span several different kinds of system.

The first kind of system is exemplified by an embedded real-time system, something that can typically be treated as a black box and defined by its interfaces. Another kind of system is exemplified by an on-line commercial information system, where it is not clear that the black box perspective is appropriate, since any connection could be negotiated by the parties concerned, and the use of the system has important implications outside the system. It is not at all obvious that the same compositional principles apply in both cases — indeed this is something we are investigating.

Of course, these two examples are just different points on a spectrum, with most systems coming somewhere between them. One important dimension of the spectrum is the extent to which the implications of invoking the services provided by the system can or cannot be confined to state variables within the system – others are discussed in Section 2.

1.5 Structure of the document

The purpose of this deliverable is to present a revised version of the DSoS conceptual model, which was first presented in deliverable BC1. As part of this revision, we have attempted to generalize the model by identifying abstract concepts that are applicable to more than one kind of system of systems. We have also developed a taxonomy (see Section 2) in order to explore the range of possible systems of systems, and the different factors that could impact upon the dependability of such compositions of systems. Sections 3 and 4 introduce the set of basic DSoS concepts including a model of time. We then present our initial ideas about formalization in Section 5. Finally, we conclude by summarizing the contents of the deliverable and briefly discussing further work. Annex 1 contains further information about our models of time, and Annex 2 provides a glossary.

2 TAXONOMY OF SYSTEMS OF SYSTEMS

The purpose of this taxonomy is to assist DSoS in its aim of developing a coherent overall understanding of the dependability related problems, and opportunities, that are inherent in a whole spectrum of system of systems. This can be, and is being here, undertaken without regard for the particular domain of application – nor those aspects of dependability required – of the resulting system. In particular, it aims to situate the concerns of DSoS, related to dependability, autonomy and time, in the overall domain of system construction using components that are – or could usefully be regarded as – complete systems in themselves.

In its present state of development, it draws principally on the basic concepts and ideas from the work of distributed systems and system architecture research communities, in effect summarizing this material from a taxonomical viewpoint. It no doubt merits further development and refinement.

This taxonomy of systems of systems is organized into three principal parts. The first involves a classification based on the attributes of an individual system. This concentrates on attributes that are of particular relevance to the fact that the system is being, or might be, used as a component in one or more systems of systems whose dependability is of concern. The second part is based on the attributes of the collection of systems that are incorporated in a system of systems (i.e., on issues that are to do with what has been called a "global architectural structure"). The third part is based on attributes of the connections between the systems that make up a system of systems.

2.1 Attributes of Systems

The attributes of systems that are of particular relevance to the problems of incorporating them into a system of systems relate to a number of readily distinguishable types of issue, including autonomy, controllability, observability, etc.

2.1.1 Autonomy dimension

It is useful to distinguish between several different forms of autonomy, i.e., independence of the considered component system with respect to its existence, its operation and its evolution.

2.1.1.1 Independent existence

We distinguish between:

- component systems that were built especially for a given system of systems,

- component systems that are re-used, either (a) having been built with re-use in mind (component-based engineering; COTS; general-purpose servers and services), or being (b) legacy components.

2.1.1.2 Independent operation

The various component systems involved in a system of systems can either be:

- operating under independent management, in which case their involvement in the system of systems may either be subject to a service delivery contract, or (most problematically) involve no contractual obligations, or
- operating under the same global management as the system of systems of which it is part.

2.1.1.3 Independent evolution

A system of systems may have to cope with the fact that component systems can evolve. In such situations, their component systems can either

- evolve under independent management, in which case their involvement in the system of systems may be subject to a contract that ensures stability of its interfaces or (most problematically) involve no contractual obligations, thus introducing the possibility of dynamic interface mismatch, or
- evolve under the same global management as the system of systems of which it is part.

2.1.2 Controllability dimension

In this dimension, the issue is whether a system has to be treated as a black box whose internal operation cannot be interfered with, or has instead been provided with an "*intercession interface*" (either explicitly by its designer, or implicitly by the enclosing infrastructure).

2.1.3 Observability dimension

In this case, the issue is whether a system has to be treated as a black box whose internal operation cannot be observed, or has instead been provided with an "*introspection interface*" (either explicitly by its designer, or implicitly by the enclosing infrastructure).

2.1.4 Dependability provision dimension

In this dimension we distinguish between:

- provisions w.r.t. internal faults – there may be none, so that it is necessary to rely on external error detection, or the system may have an exception reporting interface (whose use can be supplemented by means of external error detection) or at least have a controlled failure mode (e.g., will only fail by crashing),
- provisions w.r.t. external faults – again there may be none, so that it is necessary to rely on external error detection, or it may have means of detecting one of more classes of threat.

2.1.5 Dependability justification dimension

In this dimension, it would be possible to classify component systems according to:

- the construction, the verification, and the evaluation processes that their designers have employed (if any),
- any quantified guarantees related to reliability, availability, security, safety, and various QOS/performance measures (throughput, latency, WCET…).

2.1.6 Functional dimension

Two aspects of this dimension that are of particular relevance to the task of creating a dependable system of systems concern:

- the designers' knowledge of/confidence in the semantics of the services offered by each putative component system,
- the extent to which these semantics are formally specified.

2.1.7 Other classical (non-SoS-specific) attributes

Other relevant attributes of component systems include their flexibility / adaptability.

2.2 Attributes of Collections of Systems

This dimension of our taxonomy concerns the collection of component systems, in particular the legacy component systems, as a whole – a topic that is termed the "global architecture structure" by (Garlan, Allen et al. 1995)

2.2.1 Integration dimension

Component systems can be integrated together to form a system of systems at various integration levels, such as:

- network-level integration (e.g., TCP/IP…)
- component architecture (e.g., CORBA, COM…)
- web-level integration (e.g., HTTP, SOAP…)

It is also worth distinguishing situations in which the integration is essentially homogeneous from those in which it is heterogeneous, in the sense that different subsets of the set of component systems are integrated together at different levels, perhaps as a result of the whole having been developed incrementally, and indeed opportunistically.

2.2.2 Interaction dimension

As described in some detail below in Section 4.2.2, interactions can be either event-triggered or time-triggered, and a number of different interaction styles are available for constructing systems of systems. These include: client-server, publish/subscribe, multipeer and peer-to-peer, the use of a data sharing repository, mobile code, etc.

With regard to both the triggering method used, and the interaction style, it is useful to distinguish between homogeneous and heterogeneous approaches.

2.2.3 Binding dimension

The binding of names to entities among the component systems can either be static, or dynamic as discussed in Section 4.2.1. In the latter case, it requires the provision of some type of naming service, which in itself may be another component system (e.g., the CORBA naming service).

2.2.4 Timing dimension

We are assuming that all component systems are influenced by the passage of time (perhaps by possessing or having access to some form of clock). The important distinction to make in this dimension is between the situation where all one can rely on is a bounded drift among the set of local clocks, and that in which there is a common notion of time, i.e., of global time (e.g., provided by synchronized clocks).

2.2.5 Mismatch dimension

This important dimension concerns the known (or assumed) property mismatches among a set of component systems, all of which will have to be handled by connection systems if they are not to be a cause of undependability. These mismatches may (i) all be known *a priori*, (ii) may vary among a known set of possibilities, or (iii) may involve the occurrence of new mismatches during operation of the system of systems. These are three increasingly difficult challenging possibilities, requiring the use of ever more sophisticated connection systems.

The differing types of property mismatch, from low-level towards high-level, include:

Physical (Mechanical, Electrical) - In order to be able to transmit bit strings from one system to another system, the mechanical and electrical and coding characteristics at the connection must be compatible.

Syntactic - Caused by incompatible information structures at a connection. This includes issues of data formats, bit and byte ordering (e.g., “endianess”), and the like.

Flow Control – If there is implicit flow control in one component system and explicit flow control in another system then the difficult problem of flow-control reconciliation must be solved in the connection system.

Protocol - Different communication protocols can be used in different parts of the component systems.

Data Representation - Representational issues normally only show up at interfaces, not within a component system. To facilitate the interconnection of systems, rules and conventions concerning data representation and data encoding need to be enforced whenever possible. The specification of a standard format for the representation of time is being investigated; simple representation mismatches might be handled by XML, but this clearly has limitations (being purely syntactic).

Temporal - The duration between a request by a client and the expected response by the server is important from the point of view of the temporal accuracy of the data (in real-time systems) and error detection. The systematic calculation of time-outs and the associated handling of orphan service requests are important research topics.

Dependability - The designer of a system of systems must make assumptions about the reliability, failure modes, and error detection and handling mechanisms of the systems to be incorporated into the system of systems. If a system of systems is to be dependable, these assumptions must be validated. This is an important topic in the DSoS Project.

Semantics - If we investigate high-level interface issues (HLII), then a property mismatch can occur if slightly different meanings are associated with a name. Such a property mismatch is called a semantic mismatch in (Garlan, Allen et al. 1995).

2.2.6 Dependability provision dimension

This relates to the classification of collections of systems according to the fault tolerance mechanisms they employ. Any such mechanisms might either be application-dependent (e.g., transactions, transactional workflow, co-ordinated atomic actions, spheres of control) or application -transparent mechanisms (e.g., providing recoverability, and perhaps making use of replication).

2.2.7 Dependability justification dimension

In this dimension, it would be possible to classify collections of systems according to:

- the construction, the verification, and the evaluation processes that their designers have employed (if any),
- any quantified guarantees related to reliability, availability, security, safety, and various QOS/performance measures (throughput, latency, WCET…).

2.3 Attributes of Connections between Systems

2.3.1 The nature of connectors

Using the classification given by (Garlan, Allen et al. 1995), the issues here concern the protocols and the data models used. Regarding protocols, the primary distinction is between blocking and non-blocking protocols – see Section 4.3 below. The data models used in

transmitting information among component systems will, presumably, be based on the forms of data representation used by these systems.

2.3.2 Type dimension

The connection types that we have identified (see Section 3.3 below) are boundary lines and connection systems, where the latter may deal with various types of mismatch, and provide varying sophistication of mismatch resolution mechanism.

2.3.3 Dependability dimension

Connection systems, though not boundary lines, can be classified in this dimension according to their provisions regarding their own internal and external faults, and the quantitative guarantees (if any) that can be given regarding the dependability of their provisions for coping with mismatches. This classification is therefore essentially the same as that given in Sections 2.1.4 and 2.1.5 above concerning (component) systems.

2.3.4 Flexibility dimension

Connection systems can be developed *generically*, e.g., like CORBA or *specifically*, e.g., a wrapper for a legacy system. Generic connection systems provide means for their customization whereas specific ones are specialized for the particular systems they interconnect.

3 CONCEPTS

In this section we introduce the basic concepts of the DSoS Project by a set of definitions, which though informal are intended to be precise and unambiguous, and explanatory notes.

3.1 Systems and their Behaviour

The definition of a system found in, for example (Laprie 1992) is: *A set of components bound together in order to interact.*

While not disagreeing with this definition, for our purposes we need a definition of system that incorporates a notion of time:

System: An entity that is capable of interacting with its environment and is sensitive to the progression of time.

Fundamental to this definition is the distinction between a system — the object of consideration — and its environment. The environment (itself in principle another system) takes advantage of the existence of a system and produces input information to the system and acts on the output information from the system. Since our main focus is on the information exchanged between a system and its environment, we will abstract from the non-information relevant properties of a system as far as is meaningful and possible.

Typically, the systems in which we are interested have some degree of autonomy in that they are capable of independent behaviour, and in particular of failing. (A standard definition of autonomous is: “Not controlled by others or by outside forces; independent.”)

Our definition of system excludes, for example, a software package without an associated processor. However, we would consider software packages that share the same processor to be separate (but not wholly independent) systems. Our definition of system also includes human organizations, for example (though these are not the focus of our project).

A system can be decomposed into interacting *component systems*. This recursive decomposition will be stopped when the inner details of a component system are of no relevance for the current analysis. Conversely, a set of systems can be composed to form a system of systems; i.e., a new system is generated.

A behaviour (see the definition below) can only be associated with a system if some notion of time is taken into account. Time is also important for the introduction of the concept of a failure. This justifies organising further definitions around a *timeline*.

The introduction of temporal awareness requires a model of time. We assume a model based on Newtonian time. Time progresses along a dense timeline, consisting of an infinite set of instants, from the past to the future.

Instant: A cut of the timeline.

Duration: A section of the timeline.

A duration is delimited by two instants. A more detailed discussion of the DSoS model of time is contained in Annex 1.

Interface: A point of interaction between a system and its environment.

At the physical level, for instance, an interface can exist as a single line (a serial port) or a set of lines (a parallel interface).

An interface can be an output interface or an input interface or both, i.e., a bi-directional interface.

Output Interface: An interface of a system at which information is produced for the environment of the system.

A system without an output interface is meaningless, since it cannot deliver information to its environment and, therefore, has no effect on the environment.

Input Interface: An interface at which information is consumed from the environment of the system.

It is possible to have systems without an input interface, e.g., a clock that produces periodic signals without an explicit input.

> **Example**: A smoke detector is a simple computer-controlled system with two interfaces: an input interface which is connected to a smoke sensor and an output interface which is connected to a central fire alarm station. It is required that, within one second after a critical level of smoke is detected at the input interface, an alarm message must arrive at the central fire alarm station. Crash failures of the smoke detector must also be detected within a second. The smoke detector is a system that has no control input. It samples the state of its environment at points in time that are determined by the internal clock of the smoke detector and sends its observations to the central fire alarm station, either periodically or sporadically when a relevant change-of-state has been detected.

Actuation (Sensing) Operation: The production (recording) by a system at a physical output (input) interface of a single value change at an instant or of a temporally-controlled sequence of value changes during a duration.

The concept of an actuation (sensing) operation is a general concept that encompasses the exchange of information among widely different types of systems (analog systems, digital electronic systems, computer systems).

The description of the communication among computer systems can be simplified by the introduction of the concept of a message. In DSoS, we assume that the idiosyncrasies of any sensors and actuators that interface to the environment of a computer system are, if necessary, encapsulated within transducer systems that can send and receive messages. Hence, the further development of the conceptual model will focus on the operations of sending and receiving messages.

Message: A data structure that is formed for the purpose of communication among computer systems.

Send (Receive) Operation: The sending (receiving) of a message at an interface.

Successful termination of a receive operation always results in the reception of a complete message.

Message Send Instant: The instant when the sending of a message starts at the sender.

Message Receive Instant: The instant when the receiving of a message terminates at the receiver.

A send (receive) operation requires a certain time. The duration between the start-instant of a message-send operation and the termination-instant of the corresponding message-receive operation can be of relevance for the correct operation of a system of systems.

> **Example**: A driver of a car approaching an intersection observes the change of the traffic light from “green” to “yellow”. He/she makes a decision whether to accelerate and cross the intersection during this cycle of the traffic light or to brake and wait for the next cycle. This decision is transmitted to the computer system controlling the car in a message. If the message is stored in a queue for a significant interval of time, the consequent change of the meaning contained in the message can have safety implications.

The appropriate handling of a message at the sender and receiver (update in place, queue) depends on the information content of a message. In order to be able to characterize this

information content we need to introduce the important concept of *state variables* and *state observations*.

State Variable: A state variable is a *relevant* variable, either in the environment or in the computer system, whose value may change as time progresses.

Examples of state variables are the position of an actuator in a controlled system or the size of a queue in a computer system. A state variable has static attributes that do not change during the lifetime of the state variable, in addition to the dynamic attributes that may change. Examples of static attributes are the name[2], the type, the value domain, and the maximum rate of change. The value set at a particular instant is the most important dynamic attribute. Another example of a dynamic attribute is the rate of change at a chosen instant. The information about the value of a state variable at an instant is captured by the notion of an *observation*.

State Observation: A tuple $<Name, Value, t_{obs},>$ consisting of the name of the state variable, the observed value of the state variable, and the instant when the state variable has been observed..

State observations may be transported in messages to a receiver, which may reconstruct the dynamics of the environment based on the incoming messages containing state observations.

Image: A representation of a state variable, e.g., at a receiver of messages containing state observations.

Temporal Accuracy: An image is a *temporally accurate* representation of a state variable at instant t, if the duration between the time-of-observation of the state variable ($t_{obs},$) and the instant t is less than the accuracy interval d_{acc}, an application-specific parameter associated with the dynamics of the given state variable.

An image is thus *valid* at a given instant if it is an accurate representation of the corresponding state variable, both in the value and the time domain (Kopetz and Kim 1990). While a *state observation* records a fact that remains valid forever (a statement about a state variable that has been observed at an instant), the validity of an image is *time-dependent* and is invalidated by the progression of real-time. Delaying a message containing an observation in a queue may affect the temporal accuracy of the information contained in the message.

---

[2] Naming issues will be discussed later in Section 4.2.1.

Event Observation: An event observation records the occurrence of an event. An event is a significant happening, e.g., an *important difference* between a state observation immediately *before* the happening and the state observation immediately *after* the happening.

An event observation can be expressed by the tuple

<Name of the observed event, attributes of the event, time of the event>

For example, the following are event observations: *"The position of control valve A changed by 5 degrees at 10:42 a.m."* or *"An amount of 1000 Euro has been withdrawn from bank account xyz at a particular time?."* An event observation requires *exactly-once semantics* when transmitted to a receiver.

Depending on the information content within a message, we distinguish between a state message and an event message.

State Message: A message that contains only state observations.

State messages are not consumed at sending and require an "update-in-place" semantics on receiving for the proper handling of the meaning of state observations.

In many real-time and multimedia systems, state messages are sent periodically.

Periodic State Message: A state message that is sent periodically at *a priori* known instants. These instants are common knowledge to the sender and the receivers.

The instants when periodic state messages are sent can be fixed either at design time or can be negotiated during the operation of the system.

Event Message: A message that contains only event observations.

Event messages are consumed on sending and stored in a queue at the receiver to implement the *exactly-once semantics* that is required for the proper handling of event information. Event messages are sent sporadically, triggered by the irregular occurrence of events.

Periodic state observations and sporadic event observations are examples of two *alternative* approaches for the observation of a dynamic environment in order to reconstruct *the states and events* of the environment at the receiver (Garlan, Allen et al. 1995). Periodic state observations produce a sequence of equidistant "snapshots" of the environment that can be used by the receiver to infer those events that occur within a minimum temporal distance that is longer than the duration of the sampling period. Starting from an initial state, a complete sequence of (sporadic) event observations can be used by the receiver to infer the complete

sequence of states of the state variable that occurred in the environment. However, if there is no assumed minimum duration between events, the observer and the communication system must be infinitely fast.

If all messages are eventually received and each one contains a complete observation, i.e., *name, value, and time,* then the precise temporal sequence of states and events of a state variable can be reconstructed at the receiver. If this reconstruction is time-constrained—as is the case in many real-time systems and multimedia systems—then the transport delay of the communication system must be bounded. Real-time communication requires a small transport delay and minimal jitter.

In some systems, the *time-of-observation* of a state variable is not contained in the message, but inferred from the *receive instant* of the message. In these systems, the jitter of the communication system influences the temporal precision of the instant of observation. The delay of a *non-time-stamped* observation message in a queue degrades the quality of the delivered observation.

Behaviour: The temporal sequence of send operations of a system in relation to its previous receive operations, and any internal state that it retains.

A system's behaviour is characterized by its send operations, though these of course can be affected by its receive operations, and any internal state that it retains.

Service Specification: The specification of the set of intended behaviours of a system.

In the general case, all the send and receive operations since the startup of the system must be observed at all of the system's interfaces in order to decide whether the service delivered by the system is in agreement with its service specification. This specification should, but in practice may not, accurately reflect the intentions of the relevant stakeholders.

3.2 System Dependability Issues

Our concern is with system dependability, the definition of which term, and of a number of related terms, we base on that of (Laprie 1992).

Dependability: The dependability of a system is the ability to deliver a service that can justifiably be trusted, where the service is the intended behaviour of the system.

In almost all cases, the intended behaviour of a system will depend on its initial state, on the proper reaction of the system to the sequence of receive operations, and, possibly, the passage of time. In principle, different stake-holders, such as the system owners and various system

users, will have different views regarding the intended behaviour of a system, and thus of its dependability.

Failure: A failure is an event that occurs at the instant when the actual behaviour of a system starts to deviate from the intended behaviour.

Ideally, a precise service specification (both in the value domain and in the temporal domain) that specifies the intended behaviour is a prerequisite for the judgment about whether a system has failed or not. In practice, the judgment will sometimes have to take into account the inadequacies of any pre-existing specification. Different judges may thus come to different decisions with regard to whether a system failure has occurred.

State of a System: At a given instant, the values assigned to an internal data structure of a system that synthesizes all cumulative effects of all receive operations at all input interfaces between the startup of the system and this given instant.

In many legacy systems it can be difficult to determine the complete state of a system.

A system consists of a set of interacting components, therefore the system state is the set of its component states.

Declared State: At a given instant, the values assigned to a declared data structure that can be accessed via an interface and that synthesizes all relevant effects of previous receive operations up to the given instant.

Since the declared state can be accessed from the environment of the system, it is possible to observe this declared state and to store it as part of the internal state of another system.

It must be pointed out that a system may well not have a (known) declared state.

Error: An error is that part of the system state that may cause a subsequent failure.

A failure occurs when an error reaches the service interface and can be judged to have adversely affected the service.

Fault: A fault is the *adjudged* or *hypothesized* cause of an error.

A fault is *active* when it produces an error, otherwise it is *dormant*. A fault originally causes an error within the state of one or more components, but system failure will not be deemed to have occurred as long as the error does not reach a service interface of the system.

Fault Containment Region: A set of components that is considered to fail (a) as an atomic unit, and (b) in a statistically independent way with respect to other fault containment regions.

Error Containment Region: A subsystem of a computer *system* that is encapsulated by error-detection *interfaces* such that there is a high probability (the *error containment coverage*) that the consequences of an *error* that occurs within this subsystem will not propagate outside this subsystem without being detected.

Fault Tolerance: Methods and techniques aimed at providing the intended system behaviour in spite of faults.

Fault tolerance is implemented by (a) error detection and subsequent recovery, (b) error compensation, or (c) combinations of both techniques. An error that is present but not detected is a latent error. Recovery transforms a system state that contains one or more errors and (possibly) faults into a state without detected errors, though possibly with faults that could be activated again.

3.3 System Interconnection Issues

Connection: A link between the interfaces of two or more interacting systems.

Architectural Style: A set of rules and conventions governing the connections and interactions between the components of a system.

In order to build a system out of component systems, it is necessary to ensure that the interactions between the component systems conform to a consistent architectural style. This implies that the interfaces via which the component systems interact must be compatible, either directly, or after some form of adaptation.

Properties of an Interface: The set of attributes associated with an interface.

Every interface may be characterized by a set of attributes that control the types of interaction that are possible across the interface, e.g., attributes that refer to the encoding of the information, the structure of the information, the meaning of the information, or the temporal sequence of information exchanges at a particular interface.

Property Mismatch: A disagreement among connected interfaces in one or more of their properties.

If the properties of connected interfaces are in conflict (e.g., different byte orders), then a

failure can occur during system operation. So, directly connecting together non-matching interfaces is a fault.

Boundary Line: A connection between at least two interfaces with matching properties.

Whereas matching interfaces can be connected directly via a boundary line, connecting together non-matching interfaces requires the introduction of a new entity that we call a connection system. The role of the connection system is to resolve the property mismatches between the connected interfaces.

Connection System: A new system with at least two interfaces that is introduced between interfaces of the connected component systems in order to resolve property mismatches among these systems (which will typically be legacy systems), to coordinate multicast communication, and/or to introduce emerging services.

A connection system is delimited by at least two boundary lines, one for each of the component systems that it connects. By definition, there are no property mismatches at any of these boundary lines.

> **Example**: An electric appliance that has been manufactured according to US standards and that is used in Europe has to face property mismatches with respect to the physical dimension of the plug, voltage and frequency. A special connection system (some kind of transformer) that has two boundary lines, one according to US standards and the other according to European standards, can resolve these property mismatches.

At a given level of abstraction, a boundary line does not introduce any relevant properties of its own. For example, if the physical length of a connection introduces a propagation delay between two interfaces that must be considered, then such a connection must be modelled by a connection system and not a boundary line.

> **Example**: If it is of relevance that a wireless connection can be monitored by an intruder, then this connection must be modeled by a connection system with an extra output interface to the intruder.

Connection systems and boundary lines can be viewed at different levels of abstraction. If a property mismatch is not relevant at a given level of abstraction, then the connection system that deals with the mismatch, and the boundary lines over which it communicates with the interacting component systems, can be abstracted away to a single boundary line that connects the component systems directly. Conversely, a boundary line that hides a particular property mismatch can be refined into a connection system (and appropriate connecting boundary

lines) that expose the detail of dealing with that property mismatch.

Figure 1 depicts this expansion of a boundary line into a connection system that is delimited by two boundary lines. This expansion can be continued recursively until the proper level of detail is exposed. In the following sections, we will make use of this expansion whenever appropriate.

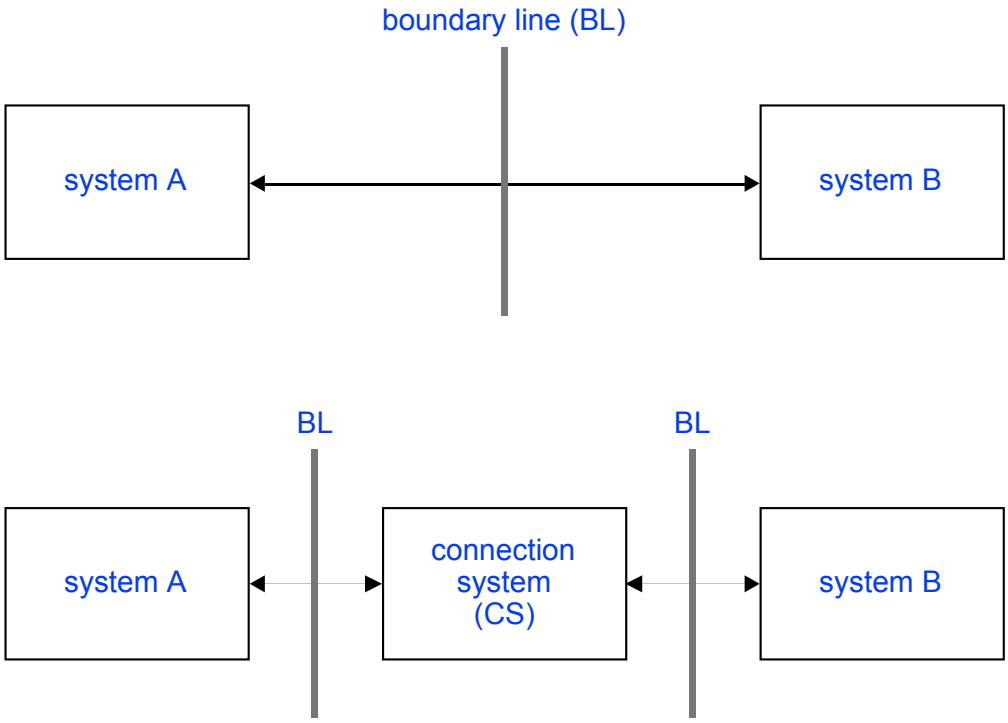

Figure 1 — Expansion of a Boundary Line (BL) into a Connection System (CS) with two Boundary Lines

Communication across a boundary line is only possible if the interacting systems share a set of concepts and a notion of time. The science of semiotics, the study of signs and their relation and interpretation, subdivided into the fields of syntax, semantics, and pragmatics, is relevant in this context. The required common knowledge among the interacting partners must be established either prior to the exchange of a connection data structure or has to be bootstrapped during different phases of the communication. The designer of a connection must be careful to specify all assumptions about this common knowledge that are a prerequisite for a successful communication across the connection. Any mismatch of the concepts or any other properties of the connections among the connected partners will cause a failure of the communication with respect to this specification. Section 2.2.5 identifies a number of types of property mismatch that can occur at a connection.

Linking Interface: An interface of a component system through which it is connected to other component systems within a given system of systems.

Local Interface: An interface of a component system that is not a linking interface within a given system of systems.

An existing legacy system is likely to have many different interfaces. The services of a system can only be accessed via its interfaces. The notion of a linking interface focuses on those interfaces that are needed to generate the emergent services produced by the desired integration. The emergent services can be functional or non-functional. For example, a replication of systems can be introduced for the sole purpose of introducing fault tolerance (and thereby improving the dependability), without a change in the functionality.

Linking Connection: A connection between two or more existing systems that is introduced in order to incorporate these systems into a system of systems with new emergent services.

Interaction: A sequence of message exchanges between connected interfaces.

This sequence of message exchanges must be specified by a protocol that is respected by all these connected interfaces.

Protocol: The set of rules that specifies the interactions between two or more component systems between connected interfaces.

The notion of a protocol is more restrictive than the notion of a service specification. The service specification may cover the behaviour of a system at all of its interfaces, whereas the protocol is focusing on the connected interfaces.

Temporal Composability: The characteristic that ensures that the temporal properties of a component system are not influenced by the integration of the component system into a system of systems.

3.4 Time

The conceptual model of the DSoS Project is notable for the fact that it includes time as an integral feature. This is done for the following reasons:

1. The DSoS Project is concerned with the design of dependable systems of systems. The classification, detection, and handling of failures are thus an important part of the DSoS Project. The simplest external failure mode of a system is a *crash failure* (Laprie 1992);

i.e., a system either operates correctly or does not operate at all. Crash failures can only be detected in the temporal domain.

2. A number of generic services that are required in the design of distributed systems, such as a membership service, can only be defined if the temporal dimension is part of the conceptual model.
3. Many communication protocols that control the interactions among component systems depend on the consistent specification of *time-out values* for their proper and efficient operation. The DSoS conceptual model should provide the capability to develop a calculus for the setting of these time-outs.
4. The DSoS model is to cover the specification, design, and validation of, *inter alia*, so-called real-time systems. In these systems, the validity of real-time information depends on the progression of physical time. For example, it makes little sense to talk about the angular position of a crankshaft in an automotive engine, if the precise instant when this position was measured is not recorded as part of the measurement. In real-time systems, time is an integral part of the concept of an observation. If the DSoS model does not contain a proper model of time, it is not possible to address these core properties of real-time systems.

The inclusion of time in the DSoS model has a number of consequences. The most far-reaching consequence is that, as indicated earlier, DSoS component systems must be physical (typically hardware/software) systems. A stand-alone piece of software has no temporal properties and is, thus, not a proper object of integration in the DSoS context.

In other contexts, such as software engineering, this issue of how to integrate pieces of software together is central. Although a stand-alone piece of software has no temporal properties, these properties might be defined a priori and be required to be respected when the software is installed (along with other software) on a given piece of hardware. Schedulability analysis aims to show that these temporal properties can be respected (in the absence of faults). Violation of the temporal properties at run-time leads to a timing failure for which appropriate detection and tolerance mechanisms might be provided.

Time Measurement

The following three different types of time measurement are supported by the DSoS model:

a) Time Measurement by an Omniscient External Observer

b) Global Time

c) Local Time.

Time Measurement by an Omniscient External Observer: We assume for definitional purposes that there exists an *omniscient external observer* who can observe all events that are of interest in a given context (relativistic effects are disregarded), and that this observer possesses a *unique reference clock z* with frequency f^z, which is in perfect agreement with the international standard of time. The counter of the reference clock is always the same as that of a chronoscopic international time standard (e.g., TAI time or GPS time). We call $1/f^z$ the *granularity* g^z of clock *z*. Let us assume that f^z is very large, say 10^{15} microticks/second, so that the granularity g^z is *1* femtosecond (10^{-15} seconds). Since the granularity of the reference clock is so small and there is only a single reference clock, the digitization error of the reference clock will be disregarded. Whenever the omniscient observer perceives the occurrence of an event *e*, she/he will instantaneously record the current state of the reference clock as the time of occurrence of this event *e,* and will generate an *absolute timestamp* of the event *e*. Since there is only one reference clock, issues concerning the consistency of observations among many observers do not arise. The temporal order of events that occur between any two consecutive microticks of the reference clock, i.e., within the granularity g^z cannot be reestablished from their absolute timestamps. This is a fundamental limit in time measurement. In the DSoS model, we will make use of this time measurement by the omniscient external observer if we want to reason about the temporal relationship between events that cannot be precisely measured within the component systems.

Global Time: A number of distributed systems, particularly distributed real-time systems, synchronize the local clocks of the nodes in order to establish an approximation of a common global time (Kopetz and Ochsenreiter 1987). Suppose a set of *n* nodes exists, each one with its own local physical clock c^k that ticks with granularity g^k. Assume that all of the clocks are internally synchronized with a precision Π, i.e., for any two clocks $j,k \in [1,n]$ and all ticks *i*:

$$\left| z(tick_i^j) - z(tick_i^k) \right| < \Pi .$$

It is then possible to select a *subset of the ticks* of each local clock *k* for the generation of the local implementation of a global notion of time. We call such a selected local tick *i* a *macrotick* of the global time. For example, every tenth tick of a local clock *k* may be interpreted as the global tick, the *macrotick* t_i^k , of this clock. If it does not matter at which clock *k* the macrotick occurs, we denote the tick t_i without a superscript. A global time is thus an *abstract notion* that is *approximated* by properly selected ticks from the synchronized local physical clocks of an ensemble. A global time *t* is called *reasonable*, if all local implementations of the global time satisfy the condition

$$g > \Pi$$

the *reasonableness condition* for the macrotick granularity *g*. This reasonableness condition ensures that the synchronization error is *bounded* to less than one *macrogranule*, i.e., the duration between two macroticks. If this reasonableness condition is satisfied, then for a single event *e* that is observed by any two different clocks of the ensemble:

$$| t^j(e) - t^k(e) | \leq 1,$$

i.e., the global timestamps for a single event can differ by at most one tick. *This is the best we can achieve!* Due to the impossibility of synchronizing the clocks perfectly and the denseness property of real time, there is always the possibility of the following sequence of events: clock *j* ticks, event *e* occurs, clock *k* ticks. In such a situation, the single event *e* is time-stamped by the two clocks *j* and *k* with a difference of one macrotick. The finite precision of the global time base and the digitalization of time cause an unavoidable error in time measurement in a distributed systems that is extensively discussed in (Kopetz 1997).

Local Time: In many distributed systems there exists no global notion of time. In these systems every node has its own local oscillator that establishes a local time base for this particular node.

For a more detailed discussion on the DSoS models of time, refer to Annex 1.

4 INTERFACE AND CONNECTION CHARACTERIZATION

Let us analyze a request-response interaction between, for the sake of simplicity, just two component systems A and B (Figure 2). Component system *A* produces a request D_{AA} according to an architectural style intrinsic to itself. (In our notation the first subscript denotes the producer of the information, the second subscript denotes the architectural style of the information.) The architectural style comprises the set of rules and conventions that are specified in an architecture and must be adhered to by the component systems at their linking interfaces in order to avoid property mismatches at the interfaces. For *B* to understand this request, its architectural style has to conform to *B*'s architectural style. Any such required transformation of D_{AA} to D_{AB} is done by a connection system (CS). Sometime later, *B* responds to the request from *A* with D_{BB}, which is then transformed as appropriate by the connection system and delivered to *A* as D_{BA} at some later instant. If both *A* and *B* conform to the same architectural style, then the connection system may be collapsed to a single boundary line BL (cf. Figure 1, page 24).

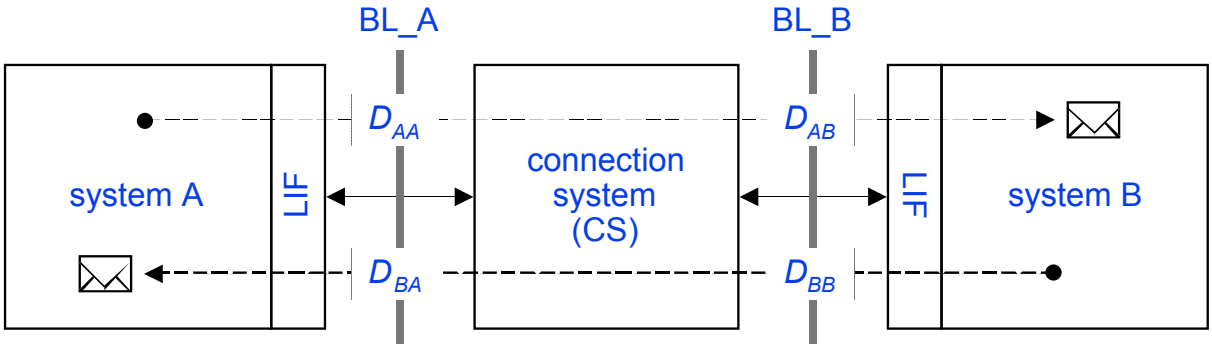

Figure 2 — Request-response interaction through a connection system

A connection system is thus necessary to resolve mismatches when there is communication between component systems with non-matching interfaces. In the software community such a connection system is often called a connector. At a high level of abstraction, a large software system can be described as a configuration of component systems and connectors (Deline 1999): connectors mediate the interaction among components. At this level, Architecture Description Languages (ADL) (Medvidovic and Taylor 2000) have been introduced to model components, connectors, and their configurations.

The integration of a set of component systems into a system of systems is substantially simplified if all component systems conform to the same architectural style. An architectural style prescribes the endorsed properties of the interfaces of connected component systems

such that all significant property mismatches are eliminated. It is possible to solve the mismatch problem by designing a special connection system for every legacy component system that transforms the properties of a legacy system to this uniform architectural style. Such a special connection system is called a *wrapper* (Deline 1999 p.26). A prerequisite for designing wrappers around existing legacy component systems is the definition of a linking architecture that defines the intended architectural style.

The components systems *A* and *B* must process received information and eventually respond, either with an action within their environments, with a response across the linking connection, or with an internal state change. In real-time systems, the duration of the interval between information receipt and the corresponding response must be bounded. The type of data transformation that must be performed within a component system is specific to the given application. One of the research issues in the DSoS Project is to find out which formalism should be used to describe the intended functions of the component systems, as seen through the linking interface. Since, in general, in a legacy component system it will not be possible to describe the whole component system, a focus is placed on devising a formalism that supports the encapsulation abstraction of the functions as required by the linking connection (Gaudel 1994).

4.1 Interface Types

In order to disentangle unrelated functions it is advantageous to specify a distinct interface for every separable service (Kopetz 2000). We have identified three unique functions that occur in many scenarios and should normally be serviced across independent interfaces.

Service Interface: This is the interface that provides the intended service to the environment, namely the systems with which it interacts.

The service interface is the most important interface for the user of the service. To keep the service interface small and understandable, only those objects and functions that are required for the intended emerging service should be visible at the service interface. It is counterproductive for all internal objects of a component system to be visible at the service interface.

In the CORBA world (Siegel 2000), the (syntax of the) services that are provided by an object are defined by the interface definition in a special interface definition language (IDL) that can be mapped into a number of different programming languages. The interface definition specifies the operations that can be performed by the object, the input and output parameters, possible exceptions that may by raised by the object during execution, and possibly, the

declared state of the component.

In real-time systems, the purpose of the *real-time service* (RS) interface is the timely exchange of observations among the component subsystems. An observation states that the state variable possessed the stated value at the indicated instant or an event occured at the instant. In control applications, the temporal access pattern of information at the RS interface is typically periodic, and a small delay and minimal jitter are important for the quality of control. These temporal parameters must be stable in order to support the composability at the RS interface. The user of the observations at the RS interface must know only about the meaning of these observations but does not need any knowledge about the internal structure or operation of the component system that delivers the observation.

The Diagnostic and Management (DM) Interface: The DM interface provides a communication channel to the internals of the component system for the purpose of diagnosis and management.

A maintenance engineer who accesses the internals of a component system via the DM interface must have detailed knowledge about the internal structure, the internal objects and the precise behaviour of the system. The end-points of communication are the internals of a component system on one side and some maintenance system or engineer, possibly sitting at a remote terminal on the Internet, on the other side. The communication pattern is, thus, point-to-point and the messages between the maintained component system and the maintenance system or engineer must be routed transparently through a set of networks. The DM interface should be independent from the service interface, since these two interfaces are directed towards two different user groups and require different knowledge.

In a real-time system, there is usually a need to support on-line maintenance and management while a system is operational. To achieve this objective, any sporadic maintenance and management traffic must coexist with the time-critical real-time traffic without disturbing the latter. The traffic pattern across the DM interface is normally sporadic and not time-critical, although precise knowledge about the instant when a particular value was observed or modified can be important.

The Configuration Planning (CP) Interface: The CP interface is used during the integration or reconfiguration phase to connect a component system to other component systems of a system of systems.

The CP interface is typically point-to-point and not time-critical.

4.2 High-Level Interface Issues

We now consider several issues relating to interactions between component systems.

Issues relating to the interpretation and handling of the information exchanged between the component systems and the dependency of D_{BB} on D_{AB} (cf. Figure 2, page 29) constitute the high-level interface issues (HLII). In particular, the following topics are part of the HLII:

a) Naming

b) Interaction styles

c) State persistence

4.2.1 Naming

Naming is concerned with associating an entity with an identifier within a defined context (Radia and Pachl 1993). To resolve a name means to decide which entity is denoted by the name. The rules that determine which context, out of the many contexts in a large system, must be selected in order to resolve a given name are called *closure mechanisms*. If the same meaning is assigned to a name in different parts of a system, the naming schema is called coherent. Whenever there is an incoherence in naming among interacting component systems, i.e., a naming mismatch, a connection system must be employed to resolve this incoherence.

We distinguish between the following name structures (Hauzeur 1986):

a) Flat name: the names of all entities are unstructured elements of a specified context, the name space.

b) Partitioned name (or compound names): a concatenation of flat names, describing a context, a sub-context, a sub-sub-context and so on until the entity is identified.

Partitioned names are useful in a distributed system, since a section of the name can be used to identify the context, e.g., the particular system, where the name has to be resolved.

Names can be static or dynamic. A static name implies that the name is always associated with the same entity. A dynamic name means that the assignment of names to an entity can vary over the lifetime of the system. However, at any instant, a dynamic name refers to a particular entity out of the selected context. Radia and Pachl investigate how the context for resolving names is selected (Radia and Pachl 1993): "*For a given name n, what context c should be used to yield the correct entity c(n)? An implicit context is needed whenever a name is resolved. An implicit context cannot be avoided, because whenever a context is specified explicitly by a name another implicit context is needed to resolve that name; therefore one*

implicit (nameless) context is needed whenever a name is resolved."

(Saltzer 1978) investigates some of the issues that have to be resolved if two or more parallel and independently operating naming systems are asked to cooperate coherently with each other. These issues are:

a) Sharing objects between systems that have different name space designs.

b) The effect of moving of an object from one system to another system on naming.

c) Naming and consistency of replicated objects.

In principle, there are two possible approaches extending the naming schemes of autonomous legacy systems to support limited interactions in a federated environment (Radia and Pachl 1993):

a) The establishment of cross-links between the local naming graphs in order to create an encapsulated subset of shared entities that can be accessed from both systems.

b) The generation of a new, united name space by the hierarchical integration of the name spaces of the existing legacy systems. This is the approach of the Newcastle Connection (Brownbridge, Marshall et al. 1982).

For the DSoS Project, alternative (a) seems to be more appropriate, because we do not want to expose all names of a legacy system to the other systems in the system of systems but rather restrict the interaction to a well-defined context of shared entities. The problem of how to design name spaces in order to support controlled information transfers across linking connections in a DSoS is an important research topic in the DSoS Project.

There are many different types of entities that are named in a computer system: hardware units, memory references, files, data records, variables, programs, etc. (Some of these entities take the role of a container, the contents of which change dynamically, e.g., a variable.) In a system of systems where it is assumed that the component systems have been developed independently, the same name can — and probably will — carry a different meaning in each one of the component systems. Coherence in naming is essentially impossible to achieve in a system of systems.

When investigating high-level interface issues (HLII), the relationship between a name and its meaning in human communication becomes an issue (Hayakawa 1990). In natural languages a name often refers to a *concept*. According to (Vigotsky 1962), a *concept is a consolidated unit of thought that abstracts and characterizes an aspect of reality*. If a variable name denotes a concept, the associated variable value signifies a particular instance of that concept.

A variable can then be considered as representing an indicative proposition, e.g., temperature = 20 means "*the temperature is 20 (degrees Celsius)*". Many natural languages support syntactic forms to express the subjective truth-value of a proposition (conjunctive, subjunctive) and to place the proposition in the temporal context (tenses). The limited awareness of the temporal validity of information in many computer systems is a cause for many inconsistencies and failures. The notion of an observation (see Section 3.1) tries to make this temporal aspect explicit.

The relationship between variable names in programs and concepts in the natural language of the programmer is exploited by (Caprile and Tonella 1999) for gaining an understanding of the meaning of legacy software.

The explicit inclusion of a flat name in a message leads to the formation of an atomic unit that can be interpreted in any context that can resolve these names. This requires, however, that the context of message names is global to all communicating partners and entails the following consequences:

a) If incoherence in naming is to be avoided, the size of the name space for message names can become huge in large systems. This can cause inefficiencies if small data structures are communicated.

b) One cannot encapsulate communication, i.e., avoid the possibility of interference between communications that are occurring among one set of component systems, and communications among a second separate set of components, unless there is a coordinated scheme of name allocation.

c) The architectural rule of including a flat name in every message cannot be enforced on legacy systems.

The designers of CAN (control area network (CAN 1990)) have decided to follow this approach. However, it soon became apparent that the originally-provided name space in CAN would have to be expanded. Still, naming incoherence can normally not be avoided if multiple CAN domains are deployed in a large system.

> **Example**: Consider the case where the internal parameters of a component system have to be changed by a diagnostic message from a maintenance access point. If the namespace is unstructured, then all other component systems must be designed such that this (internal) diagnostic message name is different from the message names to all other component systems.

4.2.2 Interaction styles

Component systems may communicate using different patterns of interaction. For example, a travel agency may send a query to an airline's flight database and wait for its response. An engine controller in an automobile might raise an interrupt informing all onboard systems that the engine temperature is too high. We classify these forms of coordination of the computational activities of distributed component systems into *interaction styles* (Garlan, Allen et al. 1995)

4.2.2.1 Client-server interactions

The client-server model is a popular approach for organizing software across distributed platforms. In its basic form, clients interact with (human) users and contact the servers to ask for computationally-intensive or data-intensive services (Hauswirth and Jazayeri 1999). This model is based on request-reply interactions between the client and server, which are normally one-to-one and synchronous.

The interaction style of client-server systems may be *connection-based*, in that a state is shared between a client and a server and is modified by their interactions. Conversely, as in basic web-based systems, the interaction may be *connectionless* in that no state information concerning clients is kept by the server between interactions. The management of state dependencies between interactions is in this case delegated to the clients by means of *cookies*, or, less elegantly, through hidden fields in post requests. Alternatively, the server can manage a connection-based interaction by means of a session identifier encoded in the page URL (see, e.g., `www.sun.com`).

In the basic client/server model, clients have a fixed, pre-allocated knowledge of the identity of the servers. Improved flexibility is provided by the use of a naming or trading service, which allows the identity of the most appropriate server to be determined dynamically.

Client-server interactions can be implemented by remote procedure calls (RPC) or by remote method invocations (RMI).

Remote Procedure Call (RPC)

In the remote procedure call (RPC) form of interaction, the arriving message causes the activation of a remote procedure (information push) at the receiving component system. In the Distributed Computing Environment (DCE) of the Open Software Foundation (OSF 1992), remote procedure calls are proposed for communication across heterogeneous platforms. Since the RPC glue can be generated automatically by the middleware, neither the sender nor

the recipient needs to be aware of the remoteness of the call (if the temporal aspects are disregarded). This transparency, which makes RPC calls look similar to local procedure calls, hides the fact that the sender and the recipient may reside in different error fault containment regions. The performance cost penalty of an RPC over a local procedure call can be of the order of more than a thousand (Szyperski 1998). The World Wide Web consortium (W3C) is currently working on the Simple Object Access Protocol (SOAP) for defining remote procedure calls in an Internet setting.

Remote Method Invocation

The main difference between an RPC and a remote object method invocation lies in the late binding of the code to call. An object instance is identified by a unique object reference (name) that can be created dynamically immediately before the call to the object's method.

Method calls can be implemented above an infrastructure that implements remote procedure calls. IBM's System Object Model (SOM) provides a runtime system that dynamically selects the methods to be called on-top of an RPC infrastructure (Forman, Conner et al. 1985).

The most prominent standard for object-oriented computing is the CORBA 3 standard developed by the OMG and described in much detail in (Siegel 2000). The OMG has introduced a special language, the interface definition language (IDL), to specify the syntax of the externally visible interfaces of objects. There exist mappings from IDL to many of the standard programming languages (C, C++, Java, etc.) to support distributed computations in heterogeneous environments.

In the object-oriented world of CORBA, an incoming message can dynamically create a new object by a method call to an object factory. The object factory instantiates the new object dynamically and returns the unique object reference to the caller. By referring to this object reference, the caller can then invoke methods of the newly created object remotely (Siegel 2000).

Other environments for remote method invocation include Microsoft's Distributed Component Object Model (DCOM) and JavaSoft's Java/RMI.

4.2.2.2 Publish/subscribe

In the publish/subscribe interaction style (which is also referred to in the literature as *implicit activation*), interactions are modeled as asynchronous occurrences of, and responses to, events. Systems do not communicate with each other directly but use a publication mechanism to announce that an event has occurred and a subscription mechanism to be

informed about the occurrence of events. This interaction style provides a decoupling between component systems:

- Space decoupling: producers do not need to know who has subscribed to their events, which in turn allows consumers to remain anonymous.
- Time decoupling: subscribers do not need to be alive at the instant the events are produced.

This reduces the static dependencies between component systems, and facilitates system evolution, but at a cost in computational predictability. Indeed, the announcer of an event does not know who will receive this event, in which order it will be delivered to subscribers, and is not informed when they finish handling the event.

The publish/subscribe interaction style depends on the existence of a middleware infrastructure responsible for propagating events from producers to consumers, and for managing subscriptions to classes of events. Different implementations of this infrastructure are possible, depending on the sophistication of the subscription mechanisms that are made available, and on the topology of the underlying interconnection network. For example:

- The multicast mechanisms in the Internet Protocol implement channel-based subscription. A channel is associated with a multicast group, which is identified by a network address.
- USENET, and its underlying NNTP protocol, implements a subject-based subscription mechanism on top of a hierarchical client/server topology. A subject identifies a single newsgroup (such as `comp.object.corba`), or a family of newsgroups (such as `comp.*`). A USENET site receives all articles belonging to the subjects to which it is subscribed.
- Messaging-oriented middleware such as IBM's MQSeries® provide reliable message queues. These queues are a form of channel-based subscription.
- The CORBA Event Service (OMG 2000) defines a publish/subscribe model for inter-object communication that complements the traditional one-to-one RMI semantics of CORBA method invocations. An architectural element called an event channel mediates the transfer of events between the suppliers and consumers as follows:
    - The event channel allows consumers to register interest in events, and stores this registration information.
    - The channel accepts incoming events from suppliers.

- The channel forwards supplier-generated events to registered consumers.

- The CORBA Notification Service (OMG 2000) extends the point-to-multipoint delivery semantics communications of the Event Service to provide additional properties:
  - Event filtering, which allows consumers to register only for specific classes of events. If no consumers are interested in receiving a particular event type then the supplier will not send the event to the notification channel. This can significantly reduce the amount of network traffic required to propagate events, improving the scalability of the service. Event filtering is content-based, using an extension of the constraint language used by the CORBA Trading Service. There is a mechanism that allows new consumers entering the system to discover which types of event are currently available.
  - Quality of service characteristics such as delivery guarantees and priorities. The event aging characteristic allows a supplier to specify a time after which the notification channel should discard an event because it is no longer considered timely. Similarly, it is possible to specify an earliest delivery time for an event. Channels can be made persistent, to ensure delivery of events across crashes. QoS attributes can be assigned at different levels of granularity: per event, per channel or per supplier/consumer. When end-to-end QoS is required, it is the programmer's responsibility to ensure that QoS is consistent across the whole path.

  The Notification Service emerged primarily from the needs of the telecommunications industry.

4.2.2.3 Multipeer

Another style of interaction is multipeer, conveying the notion of spontaneous, symmetric interchange of information, amongst a collection of peer entities. No component system is privileged with respect to its peers, and there is little or no centralized coordination. This paradigm appeared as early as in (Powell, Bonn et al. 1988) where it is called multipoint association. Multipeer interactions are the kind of interaction one might wish among managers of a distributed database or a group of servers. Communication requirements may be heavy in ordering and reliability requirements, and a notion of composition or membership may be required (for example, to provide explicit control over who is currently in the group). Again, the highly interactive nature of the multipeer style of interactions prevents *per se* the

number of participants in real applications from exceeding the small-scale threshold (Veríssimo 2000).

Peer-to-Peer

The peer-to-peer interaction style is a form of multipeer interaction characterized by opportunistic interactions. It has emerged in an Internet setting (Clark 2001), where many systems have intermittent connections to the network. This form of interaction places a strong emphasis on discovery protocols, since a peer entering the network has little information on the existence of other peers and of the services they may be offering. Popular examples of this form of interaction are instant messaging systems such as AIM, and the notorious file-sharing systems Napster and Gnutella.

Another, more ambitious, example of peer-to-peer interaction is Freenet (Clarke, Sandberg et al. 2000), a distributed file storage and retrieval system that addresses a number of reliability and privacy failings of the Internet protocols. Indeed, while the Internet is often cited as an example of a distributed, decentralized and robust architecture, this is only true to a limited extent. The naming system used on the Internet constitutes a single point of failure, and the common publication protocols are lacking certain dependability attributes.

Naming on the Internet is managed by the Domain Name Server (DNS), a hierarchical distributed database which maps from symbolic names to numerical addresses. Though it is distributed, the DNS is centrally controlled (there are a limited number of top level domains), provides limited protection against malicious updates, and has even proven to be liable to fail due to operator error during routine maintenance.

Publication systems such as the Web, while very popular, present several disadvantages from a dependability point of view:

- No built-in mechanism for load balancing: techniques such as caching and mirroring are not transparent to clients.
- Little privacy support: the publisher of a document can determine which clients have requested the document, and when.

Freenet addresses these reliability and privacy problems by implementing a new layer of routing above IP which abstracts from the location of information. It is an adaptive peer-to-peer network of nodes that query one another to store and retrieve files. The files are named by location-independent keys.

Each Freenet node has some local storage that it makes available to the network for reading and writing, and knows of the existence of a number of other nodes in the system. If it receives a request for a file that it does not have locally, it will forward the request to the peer node it thinks is most likely to have that file. When the file is found, it is passed back to the requestor through the chain of proxies (each of which notes that the file is now likely to be available from the requestor). Thus information will tend to migrate towards the nodes where it is most often accessed.

The algorithms for routing requests are designed to be efficient while only requiring local knowledge (which is necessary, since no node is privileged with respect to its peers). A request is presumed to have failed if it has exceeded a certain number of hops. There is no hierarchy or central point of failure. Freenet can be seen as a cooperative distributed file system providing location independence and transparent lazy replication.

4.2.2.4 Data passing via a repository

Another form of interaction between component systems is based on the establishment of a shared memory space that can be accessed by all interacting partners. The sender writes data into the shared memory and it is up to the recipient to decide when to read this data (information pull). To avoid the mutilation of data due to concurrency conflicts, specified atomicity properties must be maintained by the repository (e.g., mutual exclusion for the access of a record). Examples of this form of interaction include:

- Distributed filesystems such as NFS: no constraints on control propagation are necessary for multiple readers. Constraints on control propagation to provide mutual exclusion for multiple writers is assured by a locking protocol.
- AI-type blackboard architectures: a number of knowledge sources interact via a shared data structure. The knowledge sources make changes to this *blackboard* that lead incrementally to a solution to the problem. Control propagation is driven by the state of the blackboard, which triggers activity of knowledge sources.
- Database architecture: data is contained within a number of collaborating component systems. Control propagation to component systems is triggered by incoming requests.
- The temporal firewall model: a time-triggered protocol destined for hard real-time systems. Central to the temporal firewall model is a global time base, available at every node of the distributed system, and a shared data structure that resides in the communication memory (Kopetz and Nossal 1997). We distinguish between an *input firewall* and an *output firewall*. In an input firewall, the shared data structure at the

recipient's site contains state information that must be periodically updated by the producer at instants that have been established *a priori*. The temporal properties of the data at the instant of update, e.g., the temporal accuracy of the data must be precisely defined. In an output firewall, the shared memory must contain a temporally specified data structure at periodic *a priori* defined output instants. At an output instant, the output data is copied and sent to the recipient's input firewall by the communication system. The temporal firewall is a strict data-sharing connection interface without any control signal crossing the firewall. Control error propagation from one component system to another via a temporal firewall is thus impossible by design.

4.2.2.5 Mobile agents

Information communicated to a remote component may be interpreted by the latter as an executable code segment that can be executed in the environment of the component system. An example of this mechanism is a Java applet. A Java applet is allowed to execute in the same process as a client's Web browser. This mechanism poses formidable security challenges, since one must ensure that the imported code segment does not violate the established security policy.

4.2.3 Dependability attributes of interactions

A system may rely on a various non-functional characteristics of the interactions it has with other component systems. For example, a braking system will depend on the time it takes for a "brake" request to propagate to the wheel controllers.

4.2.3.1 Timing guarantees

For real-time systems, the temporal characteristics of an interaction will be important. The timing properties of a client/server type interaction depend on the timing guarantees provided by the communications infrastructure and on the time required by the server system to handle the request. These timing guarantees can be decomposed into *latency* and *jitter*.

4.2.3.2 Delivery guarantees

The reliability of the communications infrastructure is an important factor in the dependability of the overall DSoS. Some communication protocols may not provide delivery guarantees concerning the loss of messages, or their order of delivery.

4.2.3.3 Transactions

Transactions provide the capability of performing multiple actions encapsulated with certain reliability guarantees. There are three candidates for a transactional interaction style — atomic transactions, conversations and coordinated atomic actions — each providing different guarantees (Veríssimo 2000). Atomic transactions are a well-known structuring mechanism that are best suited to competitive interactions. Atomic transactions guarantee the properties of atomicity, consistency, isolation and durability (ACID). The three major currently-available distributed object environments (Corba, COM, and Enterprise Java Beans) all offer transactional services (OFTA 2000).

Conversations (Campbell and Randell 1986) are traditionally used for cooperative systems and employ coordinated exception handling for tolerating faults. Coordinated atomic actions (or CA actions) (Xu, Randell et al. 1995; Xu, Randell et al. 1999) are a structuring mechanism that integrates and extends conversations and atomic transactions. The former are used to control cooperative interactions and to implement coordinated error recovery whilst the latter are used to maintain the consistency of shared resources in the presence of failures and competitive concurrency. Coordinated exception handling is supported by distributed exception resolution algorithms (Xu, Romanovsky et al. 1998).

4.2.4 State persistence

We define the ground state of a node in a distributed system at a given level of abstraction as a state where no task is active and where all communication channels are flushed, i.e., there are no messages in transit (Ahuja, Kshemkalyani et al. 1990). Consider a node that contains a number of concurrently executing tasks that exchange messages with each other and with the environment of the node. Let us choose a level of abstraction that considers the execution of a task as an atomic action. If the executions of the tasks are asynchronous, the situation depicted in the upper section of Figure 3 can arise; at every point in real time, there is at least one active task, thus, implying that there is no point in real time when the ground state of the component system can be defined.

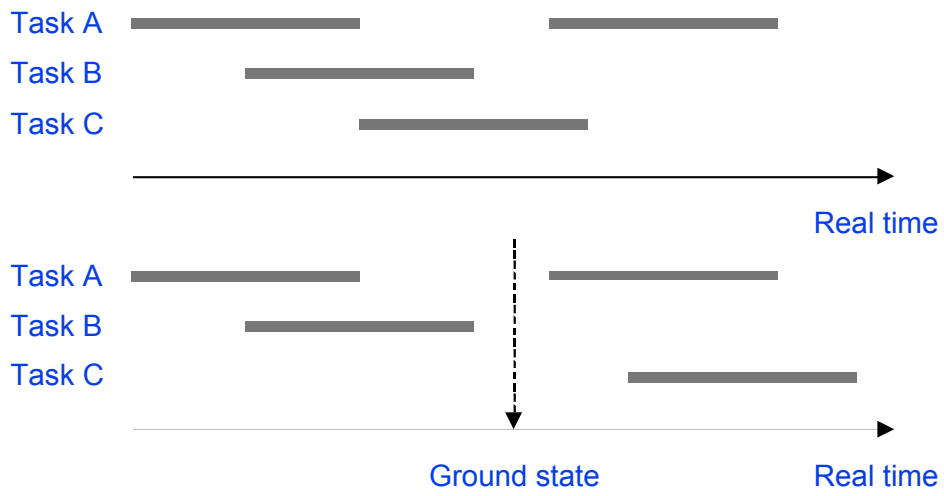

Figure 3 — Task Executions: without (above), and with (below) ground state

In the lower part of Figure 3, there is an instant where no task is active and all the communication channels are empty, i.e., where the system is in the ground state. If a component system is in the ground state, then the internal state of the component system is contained in its data structures and the program counter. The reintegration of a component system after a failure is simplified if a component system periodically visits a ground state that can be used as a reintegration point.

In many large legacy systems, it is not possible to come across an instant where the system is in a ground state. If these systems are structured according to the object paradigm, where methods and states are encapsulated in objects, it may be possible to declare a persistent state for each object or at least for the objects that are visible at the LIFs. In some applications, it might be sufficient to deal only with the persistent state that is visible from the LIF. In their most recent versions, the CORBA Common Object Services (CosServices) specify several services that are related to object persistency. The Persistent State Service (OMG 1999) for instance allows the user to define the declared state of so called "storage objects" using an extended version of IDL (the Persistent State Definition Language, PSDL). The code for these storage objects is then generated automatically in the same way as stubs and skeletons are generated from their IDL descriptions. The Externalization Service (OMG 2000) on the other hand defines interfaces like the Streamable interface, which are to be implemented by the application programmer in order to be able to store an object's state. Furthermore, FT-CORBA (OMG 2000), which is a specialized version of the CORBA specification targeting fault-tolerant applications, defines a similar Checkpointable interface. The Checkpointable interface has two methods, get_state() and set_state(), both of which are intended to be implemented by the application programmer.

4.3 Low-Level Interface Issues

Issues relating to the transport and the syntactic representation of information are considered as low-level interface issues (LLII). In particular, the following topics are part of the LLII:

a) Issues of data representation (e.g., byte order)

b) Transport timing

c) Flow control

Although there are interdependencies between the HLII and the LLII, the HLII focus on the semantic, pragmatic and — in real-time systems — the temporal aspects of the information processing within a component system, while the LLII are concerned with the transport and representation of the information. Real-time aspects are important at both levels: low-level transport timing needs to be carefully considered to ensure high-level temporal properties.

In the following, we analyze the transport and timing of a single message between two component systems A and B residing on different *sites*. These are represented in Figure 4 as *application components* A and B. The application components interact through a network by means of local *communication components*. A communication component may be, for example, a hardware communication controller such as that used in the time-triggered protocol (TTP) (Kopetz, Galla et al. 1999), a CORBA object request broker (ORB), or an HTTP server.

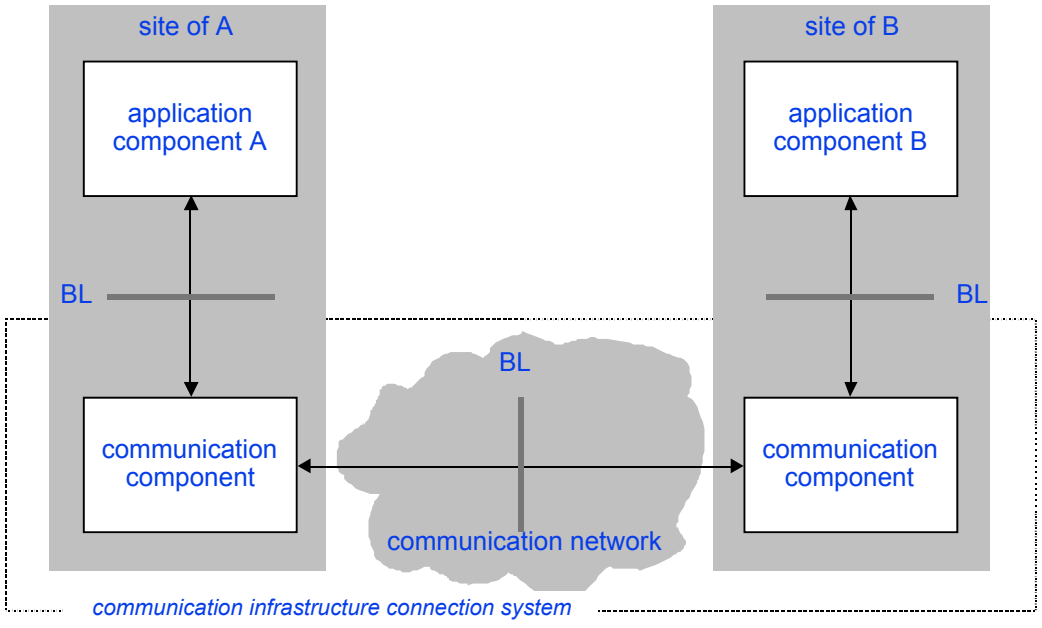

Figure 4 — Transport model between two application components on different sites

Comparing Figure 4 with Figure 1, page 24, it is interesting to note that the communication infrastructure, consisting of the two communication components and the intermediate network, can be viewed as a sort of connection system, the conventions of which must be adhered to at each extremity by application components *A* and *B*. CORBA provides an example of such a connection system, in which the communication components are the object request brokers (ORBs) and the common conventions are specified as interfaces through the CORBA interface definition language (IDL).

Each application component of Figure 4 is interfaced across a boundary line to a communication component that connects across another boundary line to the communication network and, if needed, to an intermediate connection system (Figure 5).

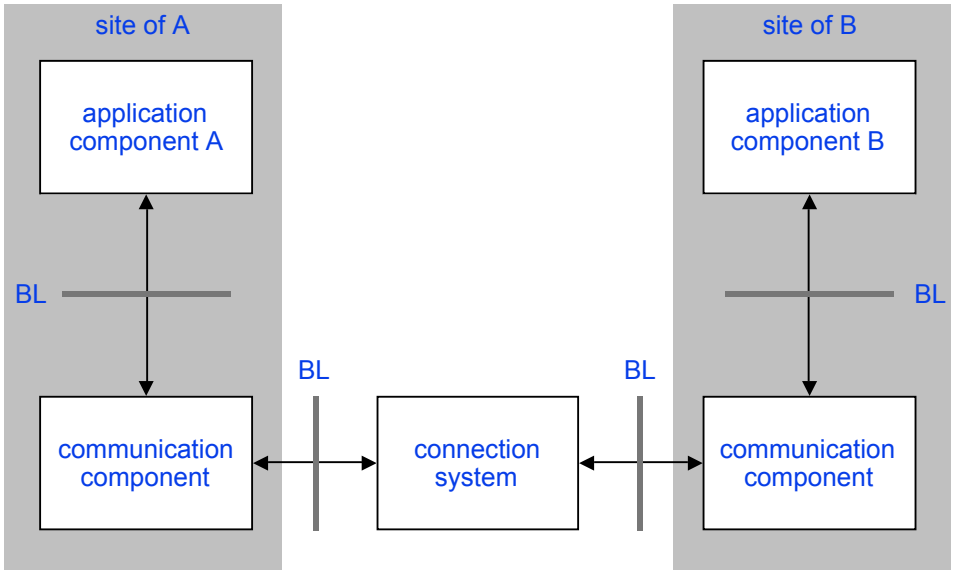

Figure 5 — Transport via an intermediate connection system

The communication components contain memory for the temporary storage, during transmission and acquisition, of communicated data structures. The inclusion of such communication memory in the transport model is justified by the following arguments:

- **Time-to-Space Mapping**: During the transmission of a message, data and control are inextricably linked. In serial communication, for example, it takes some time to assemble the arriving bits into the message data structure. The focus of interest in real-time systems of systems is on the message data structures and the associated control signals that mark the start and end of message transmission (and not on the sequence of the individual bits of a message). We therefore need a communication memory to accumulate the message

data structure out of the incoming bit stream and to act as an information source for the outgoing bit stream.

- **Design of Existing Hardware Controllers**: If we look at the design of existing hardware interfaces, e.g., commercial communication controllers, we always find a memory block associated with the communication controller. Such a memory block is either part of the communication controller or is dynamically reserved for use by the communication controller (e.g., a DMA area in the associated host computer).
- **Expressive Power of the Model**: The inclusion of a communication memory in the DSoS connection model makes it possible to describe the mechanisms of different connection types within the model. In the following section, we will classify connection types by the type of data structure in the communication memory and by the source of the control signals.

A unidirectional *data flow* takes place if the sending system publishes data in the shared communication memory at the recipient's site (i.e., if application component *A* transfers the data to the communication memory at *B*). The data is made available at a given instant. It is up to the recipient to decide when to access this data after the instant of its publication.

A unidirectional *control flow* takes place if the sending system sends a control message to the receiving system. After accepting the signal, the receiving system checks a shared communication memory at the recipient's site to identify the signal and then performs the intended actions. An example of such a unidirectional control flow is the raising of an interrupt after a new message has arrived in the communication memory of the recipient.

4.3.1 Transport timing across the interface

The timing of a unidirectional message send and receive operation across the basic communication interface is shown in Figure 6 and Figure 7. We describe this timing by taking the position of the omniscient observer with the absolute reference clock *z* that can record the occurrence of the significant events and can assign corresponding absolute timestamps *z*(*event*). It is thus possible to express the duration of relevant intervals in the metric of the physical second within our model. At *e1* the application component starts writing a data structure into the send buffer of the communication memory and signals the communication component to start transmitting at *e2*. The communication component has to wait until *e3* before it can start the transmit operation (e.g., because the channel does not become free

before *e3*). At *e4*, the transmission of the message is started at the sender. At *e5*, the transmission is completed at the receiver.

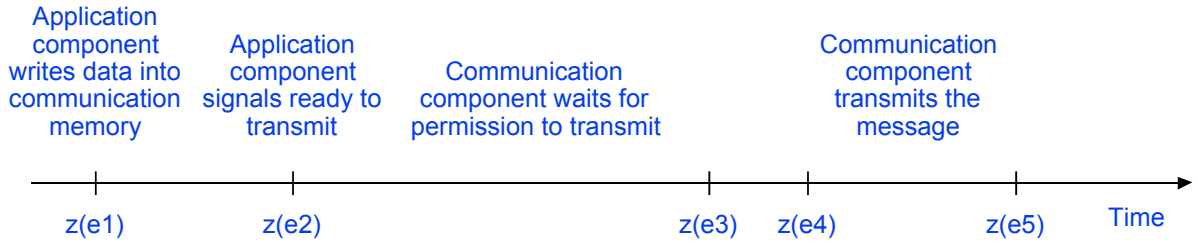

Figure 6 — Timing of a Message Send Operation

The timing of the receive operation is shown in Figure 7. At *e6,* the start of a new frame arrives at the communication boundary line. Sometime later at *e7,* the communication component starts the update of the communication memory. This update is completed at *e8*. During the interval <*e7*,*e8*> the communication controller must have write access to the memory and any concurrent reading operation will be faulted. At *e8* the communication component signals the application component the arrival of a new message. This data structure is read by the application component during the interval <*e9*,*e10*>. At *e10* the transmission is completed, and the message has been delivered to the application component.

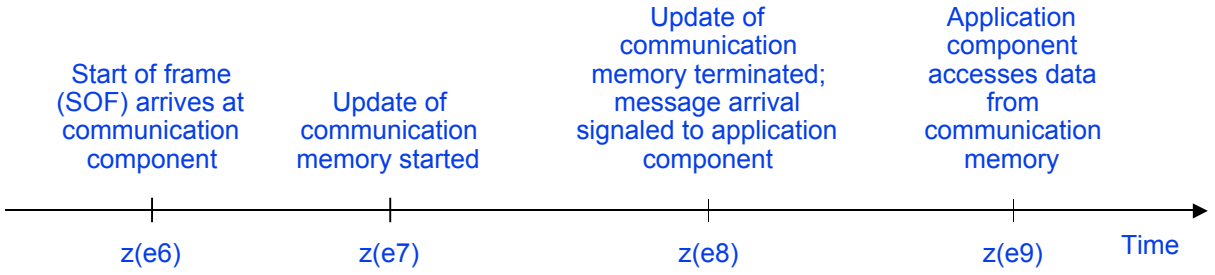

Figure 7 — Timing of a Message-Receive Operation

4.3.2 Flow control

Flow control is concerned with the control of the speed of information flow between a sender and a recipient across a connection in order to ensure that the recipient can keep up with the sender. In any communication scenario, it is normally the recipient, rather than the sender, that determines the *maximum* speed of communication. In the following, two types of flow control are distinguished: *explicit flow control* and *implicit flow control*.

Explicit flow control: In explicit flow control, the recipient sends, after the successful arrival of a message, an explicit acknowledgment message to the sender, informing the sender that

the recipient is now ready to accept the next message. Explicit flow control is based on the sometimes overlooked assumption that the sender must be under the control of the recipient, i.e., that the recipient can exert back pressure on the sender to control the rate of transmission (back-pressure flow control). The most important protocol class with explicit flow control is the well-known class of event-triggered Positive-Acknowledgment-or-Retransmission (PAR) protocols. This protocol class relies on the following principles:

a) The client at the sender's site initiates the communication at an arbitrary instant.

b) The recipient has the authority to delay the sender via explicit flow control across the bi-directional communication channel.

c) A communication error is detected by the sender when the expected acknowledgment signal does not arrive in the specified time window. The recipient is not informed when a communication error has been detected by the sender.

d) Time redundancy is used to correct a communication error, thereby increasing the protocol latency in case of errors.

Explicit flow control protocols are widely used in distributed systems. Such protocols differ, among other attributes, by the point in space where the acknowledgement message originates. If we assume that a message is sent from application component *A* to application component *B* in Figure 4 (page 44), then we can distinguish between the following four possibilities:

a) The acknowledgement message is sent by the communication component at site *A*. This is called a best-effort datagram service. Whenever the network is congested or the recipient *B* is unable to accept the message, the message is discarded.

b) The communication component at site *B* sends the acknowledgement message. The arrival of the acknowledgement message at the sender informs the latter that the message has correctly arrived at site *B*. Communication memory management is under the control of the communication component of the recipient *B*.

c) The acknowledgement message is sent after the acceptance of the message by the application component *B*. This ensures to the sender that the recipient is alive and accepted the message. This alternative is used in CSP (Hoare 1985).

d) The recipient *B* sends the acknowledgement message after it has processed the message. This is the semantics of the Ada rendezvous mechanism. This alternative corresponds to the implementation of an end-to-end protocol (Saltzer, Reed et al. 1984) between sender and recipient. It gives the highest assurance, but the lowest concurrency.

Implicit flow control: In implicit flow control, the sender and recipient agree *a priori*, i.e., before the communication is started, on the transmission rate and the instants when messages are going to be sent. This requires the availability of a global time base. The sender commits itself to send a message only at the agreed instants, and the recipient commits itself to accept all messages sent by the sender, as long as the sender fulfills its obligation. No acknowledgment messages are exchanged during run time. Error detection is the responsibility of the recipient, which knows (by looking at its global clock) when an expected message fails to arrive. In implicit flow control, the number of messages that must be delivered by the communication system is always constant. Communication is unidirectional because there is no need for a return channel from the recipient to the sender. Thus, implicit flow control is well suited to multicast communication. Publish/subscribe protocols and time-triggered protocols (such as TTP (Kopetz, Galla et al. 1999) are based on implicit flow control.

As already indicated, a prerequisite for implicit flow control is the availability of a global time base at sender and recipient. Implicit flow-control is best suited for periodic traffic patterns.

The following table (Table 1) compares the characteristics of explicit and implicit flow control:

	Explicit Flow Control	**Implicit Flow Control**
Best suited for	sporadic traffic	periodic traffic
Control flow	bi-directional	unidirectional
Multicast topology	difficult	simple
Error detection	at sender	at recipient
Error detection latency	large	small
Interface complexity	higher	lower

Table 1 — Characteristics of explicit and implicit flow control

4.3.2.1 Management of communication memory

Existing communication protocols differ in the way they manage the memory for outgoing and incoming messages. We can identify two ways by which communication memory is managed:

Dequeue/enqueue: If the transmitted information is event information, an exactly-once

semantics must be implemented by the communication protocol, because the reception of such information is non-idempotent. Event information is information on the state change of a variable. This requires a strict synchronization of the sender and recipient, i.e., every message sent must be eventually consumed. The message data structures in the communication memory are queues, where the sender enqueues a new message and the recipient dequeues this message. Enqueue/dequeue protocols require explicit flow control and consequently a bi-directional communication channel, even if only a unidirectional data transfer takes place. Multicast communication is difficult to implement with enqueue/dequeue protocols. Many of the explicit flow-control protocols use the enqueue/dequeue model. The enqueue/dequeue model is well suited for systems that have a point-to-point topology and implement information push.

Copy/update-in-place: If the transmitted information contains state information, then the sender can copy a message out of a single send buffer that is updated either periodically or whenever a state change occurs, and the recipient can update-in-place the old version of a message by the new version. The processing of sender and recipient does not have to be strictly synchronized, i.e., the recipient is free to decide when to read the state information, it can read it never, once, or many times, because state information is idempotent. The copy/update-in-place model matches well with implicit flow control. This model is well suited for systems that implement a multipoint topology and the information pull model (Deline 1999), e.g., reading a shared variable or a shared file.

4.3.3 Basic DSoS transport mechanisms

The following two transport mechanisms, event messages and periodic state messages, form the basis of the DSoS conceptual model for the transmission of a message data structure from a sender to a recipient. For a more detailed discussion of the various combinations of information types (event information, state information) and control methods (external control, autonomous control) refer to **(Kopetz 1997)**

4.3.3.1 Event message

An event-triggered (ET) message (or for short, event message) combines a unidirectional data flow with a bi-directional control flow. Since an ET message contains event information, a strict synchronization between sender and recipient is required. The memory data structure is thus a queue. As soon as the message data structure containing the event information is available in a communication memory at the recipient's site, the communication component sends a signal to the receiving system to inform the receiving system that a new message data

structure is available (information push). Since there is only a finite buffer space, the recipient must when appropriate send a control handshake signal back to the sender in order to inform the latter that the message has been consumed and that buffer space has been made available again (back pressure). There are many different protocols governing the information exchange across an event-triggered (ET) connection.

ET messages are used, for instance, in client-server protocols.

4.3.3.2 Periodic state message

A periodic state message sequence (or a time-triggered (TT) message) is characterized by a periodic unidirectional data flow into a shared memory data structure in the communication memory. Flow control is implicit. The recipient accesses this data structure based on its local schedule (information pull). Since a TT message contains state information, a new version of a state message updates-in-place the current version of the state message and no strict synchronization between sender and recipient is required. It is up to the recipient to decide when to read the message, how often to read the message, or not to read it at all. An access to a TT message interface is similar to the access of a variable in memory.

The following table (Table 2) compares the characteristics of these two transport mechanisms.

	Event Message	**State Message**
Information	event information	state information
Flow Control	explicit	implicit
Communication Memory	message queue	shared variable
Synchronization	strict	loose
Interaction type	information push	information pull
Main usage for	client-server protocols	real-time state variables

Table 2 — Characteristics of the transport mechanisms:
Event Message and State Message

From the point of view of coupling across a connection, the state-message model results in the minimum coupling between sender and recipient.

4.3.4 Integration of event- and time-triggered operation

The DSoS conceptual model distinguishes between three different types of interfaces, as described in more detail above (Section 4.1): the *service interface*, the *diagnostic and management interface*, and the *configuration planning interface*. These interfaces serve different functions, have different operational characteristics, provide access to different views of a system and, in large systems, may connect to different management domains. From the point of view of composability of services, only the characteristics of the service interface are relevant.

In real-time systems, the service interface can be time-triggered (TT), while the other two interfaces can be event-triggered (ET). In order to provide access to these interfaces on a single physical communication channel, the operation of event-triggered and time-triggered services must be integrated in such a way that the characteristic service parameters of the time-triggered interface are maintained (see also Section 4.3.3). These characteristic service parameters relate to the temporal properties of known delay and minimal jitter.

There exist three different alternatives for the integration of ET and TT services, as depicted in Table 3. The first alternative, the provision of a basic ET service at the transport layer and the implementation of the TT service on top of the ET layer is implemented in a number of industrial CAN systems that are used for real-time control. In order to reduce the jitter at the critical instant, i.e., when all nodes access the network simultaneously, these systems are normally operated with a very low bandwidth utilization. However, even under these circumstances it is not possible to guarantee a small jitter, which is important in control applications. Another alternative is the implementation of the ET service on top of the TT service. This alternative provides temporal composability and the required jitter guarantee at the transport level. It is, however, not possible to globally share the bandwidth for the ET traffic. The third alternative is a combination of ET and TT media access protocols. In this alternative, which is implemented in the FIP protocol, the timeline is partitioned in two alternating intervals for the TT traffic and the ET traffic. In the TT interval media access is controlled by a TT protocol and in the ET interval media access is controlled by an ET protocol control. The advantages of temporal predictability for the TT traffic and global bandwidth sharing of the ET traffic is bought by an increase in protocol complexity and a loss of temporal composability of the ET traffic.

Characteristic	TT on top of ET	ET on top of TT	ET and TT in parallel
Basic Service	TT operation	ET operation	one slot TT, another slot ET
Media access	TT protocol	ET protocol	ET and TT protocol
Global sharing of bandwidth	yes	no	no for TT yes for ET
Temporal composability	difficult, since global bandwidth allocation	yes	yes for TT part, no for ET part
Jitter	large (critical instant)	small	small for TT, large for ET
Examples	CAN	TTA	FIP

Table 3: Alternatives for the Integration of ET and TT Services

For embedded real-time control systems, DSoS has selected the middle alternative of Table 3, ET on top of TT, as the preferred alternative, because it supports temporal composability for the ET and TT traffic. In this alternative, event message channels are constructed on top of the basic time-triggered communication service by assigning an *a priori* specified number of bytes of selected time-triggered messages to the event-triggered transport service. These periodically transmitted bytes form a dedicated communication channel for the transmission of the dynamically generated event information. In order to implement the event semantics (see Section 2.3) at the sender and receiver, two message queues must be provided in the CNIs: the sender queue at the sender's CNI and the receiver queue at the receiver's CNI. The sender *pushes* a newly produced event-message on the sender queue, while the receiver must check the receiver queue to *pull* and consume the event message. An alternative design could produce an interrupt whenever a new event message arrives at the receiver, but such a design is not recommended since it violates the principle of providing an *information pull interface* at the receiver and could interfere with the principle of *stability of prior services* (by providing more interrupts than a node can handle).

At the conceptual level, four interfaces are provided at every node. An input and output interface for state messages (update-in-place on input, no consumption on output) and an

input and output interface for event messages (input queue and output queue) as depicted in Figure 8. State messages and event messages are stored in the memory element of the DSoS interface model.

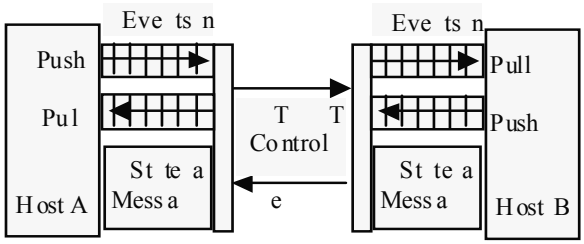

Figure 8: Model TT-ET Interface

In most real-time architectures, the basic communication service is a broadcast service (e.g., in CAN, TTP) that connects the *n* nodes of a cluster. Every transmitted event message thus generates *(n-1)* event messages at the receivers. To handle these *message showers*, two additional services should be provided by the middleware to avoid a queue overflow at the receiver: a *filter service* and a *garbage collection service*. The *filter service* selects the incoming event messages according to filtering criteria established by the receiver and accepts only those event messages that pass the filter. The *garbage collection* service eliminates decayed event message from the receiver queue based on the age of the message. A maximum *queue-storage duration* must be statically assigned to each event message for this purpose. After this duration has elapsed, the message is eliminated from the receiver queue. The event-message channels are used in the TTA to implement the non-time-critical DM and CP services. It is possible to implement widely-used event-based protocols, such as TCP/IP or CAN, on the TTA event channels.

Event message channels should not be used for time-critical or safety-critical functions. In case of a *rare-event* peak-load scenario, the event-message service may be delayed or stopped in order to maintain the safety-critical time-triggered service. It follows that the host tasks servicing the event channels can be scheduled according to the "best-effort" paradigm. Care must be taken that any software interaction between the event-service and the safety-critical time-triggered service inside the application software of the host is fully understood and no negative consequences on the replica determinism of the time-triggered service can occur.

5 TOWARDS FORMALIZATION

Formalization of SoSs developed according to the DSoS conceptual model requires the identification and/or development of formal techniques for description and validation of SoSs[3]. This section presents some preliminary ideas about how to proceed towards these objectives. A fuller treatment will be presented in the final version of the DSoS conceptual model (deliverable CSDA1), in response to feedback from the other workpackages.

According to the DSoS conceptual model, linking interfaces (LIFs) of systems and, when they exist, connection systems between such interfaces provide the means by which systems are linked together into systems of systems. Therefore, connection systems and component system LIFs are critically important to the dependability of a SoS. In addition, of course, the dependability of component systems themselves is critical. The LIFs, connection systems and component systems of a SoS must be described in a way that allows the intended dependability of the SoS to be validated.

Section 5.1 identifies features of formal description and reasoning techniques that are relevant to the formal validation of SoS dependability properties.

Section 5.2 outlines the architecture description language (ADL) that is proposed for the DSoS project in IC2(2001), and summarizes its extensibility.

Section 5.3 summarizes OMG IDL (the interface definition language defined by the Object Modeling Group), and discusses its ability to describe changing interfaces.

Section 5.4 discusses techniques for the formal description of interactions between component systems of a SoS, and quickly focuses on the suitability of the process algebra CSP for describing interactions. Suitability is partly determined by the amenability to formal validation of systems described using CSP.

Evidence for that suitability is provided in Section 5.5, which describes a CSP model of the CORBA General Inter-ORB Protocol (GIOP), together with initial validation results obtained using the model checker FDR2. Section 5.6 reviews known attempts to formalize Coordinated Atomic (CA) actions.

[3] Note that we do not consider formalisation of the DSoS conceptual model itself.

5.1 Formal Validation of SoS Dependability

This section attempts to identify major characteristics of formal validation techniques, in particular those that influence their suitability for dependability validation of SoSs. Clearly, these characteristics are determined by the DSoS validation context: what properties are to be validated, and of what systems?

Section 5.1.1 outlines the DSoS validation context. Section **Error! Reference source not found.** describes the high-level activities that formal validation of SoS dependability involves. Sections 5.1.3 and 5.1.4 list various characteristics of SoSs and dependability properties that we consider to be relevant to description/validation.

As an aside, note that we use the term 'validation' rather than 'verification'. The difference between these terms is the implied balance between reliance on assumptions and formal reasoning. Validation relies more on assumptions, and less on formal reasoning, than verification. It does not matter which term we use; it only matters that all the assumptions are identified and deemed acceptable.

5.1.1 DSoS validation context

The DSoS Validation context is the *dependability* validation of *systems of systems*.

Recall that the *dependability* of a system is its ability to deliver a service that can justifiably be trusted, where the service is the intended behaviour of the system. So dependability has two aspects: 'intended service' and 'trustworthiness'.

SoSs are systems built by the integration of existing complete component systems. Furthermore, these existing component systems are typically non-trivial and outside the control of the SoS designer.

Note that some component systems may be designed by the SoS designer, specifically for the purpose of integrating the existing component systems. Such 'integration' components can be expected to be under the control of the SoS designer.

5.1.2 Validation activities

In general, formal validation of a system relies on:

1) formal descriptions of the system and of the desired system properties;

2) formal reasoning to show that the described system satisfies the described properties.

In the context of DSoS, formal validation of the dependability of a SoS relies on:

1) formal descriptions:
   a) of the SoS;
   b) of the intended services of the SoS (functional properties);
   c) of the acceptance criteria for trustworthiness (non-functional properties);
2) formal reasoning:
   a) to show that the described SoS provides the described intended services;
   b) to show that the described SoS service provision is trustworthy (according to the described criteria).

5.1.3 Characteristics of SoSs and implications for description/validation

Here we identify characteristics of SoSs that are relevant to description/validation, either because they affect the suitability of ADLs or IDLs for describing SoSs and their interfaces, or because they affect formal validation of SoSs thus described. For each characteristic, we identify its implications for the capabilities of (a) IDLs and ADLs (when used to describe SoSs), and (b) formal validation techniques used in the DSoS context.

To help clarify the discussion, consider the example SoS depicted in Figure 9It has component systems A, C, D and E, and connection system B between A and C. The arrows represent the flow of information across connections. (Note that connection systems might reasonably be considered to be component systems themselves, though they are introduced by the SoS designer and are therefore likely to be much more controllable than are legacy component systems.)

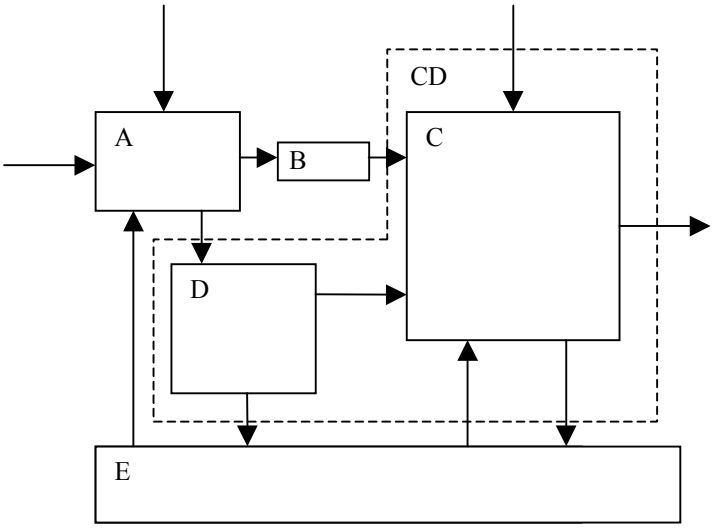

Figure 9. A system-of-systems (dotted system is an abstraction of C and D)

5.1.3.1 SoS hierarchical structure

Formal description and reasoning techniques generally benefit greatly from the ability to describe and reason about systems in a way that corresponds to the system architecture. A system architecture provides a structured way to understand, and validate, system behaviours. The characteristic feature of SoS architectures is their hierarchical nature. Architecture description languages rely heavily on hierarchical description techniques, and thereby provide a good opportunity for validation activities to exploit SoS hierarchy. The obvious strategy is to prove SoS-level properties by proving an appropriate set of properties of the component systems – we may call these 'component-level properties'. (We discuss the existence and discovery of an appropriate set of component-level properties in Section 5.1.4.2.) Once one has found a particular formal validation technique that can exploit system structure (particularly SoS hierarchy), the question arises whether exploitation can be automated, at least in part. This can be achieved by automating one or more of the following tasks:

1) generate formal descriptions of architectural components (from ADL/IDL descriptions of the SoS and its interfaces);
2) generate formal descriptions of an appropriate set of component-level properties (from SoS-level properties);
3) formally validate that the components satisfy the identified component-level properties.

5.1.3.2 Non-controllable component systems

An important feature of SoSs is their reliance on component systems that are outside the sphere of control of the SoS engineers. Such component systems may change their functionality or reliability (availability/performance) without warning; a DSoS is required to cope with such behaviour.

ADLs, IDLs and formal description/validation techniques applied to SoSs must be able to describe/validate SoSs designed to cope with component systems not controllable by the SoS engineers. In particular, these techniques must be able to describe/validate systems that employ dependability mechanisms, such as those described in IC2: CA Actions and Wrappers.

We anticipate that redundancy will often be used to cope with possible change of services provided by component systems, and of their trustworthiness. Another strategy is to track advertized service/trustworthiness changes, either for complete component systems or individual interfaces. The ability to track such changes necessitates corresponding flexibility of ADLs/IDLs.

5.1.3.3 SoS lifecycle

It is often advantageous to build a system model in the design phase of a system, and develop that model throughout the lifetime of the system. Maintaining a model in this way is advisable in order that an up-to-date system description is available for dependability validation throughout the SoS lifecycle. For example, where safety properties are concerned, it is typically necessary to maintain a safety case while the system remains in service.

Frequently, SoSs evolve by the inclusion of extra component systems that satisfy some compositional property, also satisfied by the SoS or one of its component systems. In such cases, it is advantageous if the validation techniques can exploit compositionality, allowing the SoS-level property of any future system (thus evolved) to be validated by simply validating the compositional property of the new components.

5.1.3.4 Complexity of SoSs

One reason for the complexity of SoSs is that they attempt to combine the behaviours of component systems to provide emergent SoS-level functionality. Component systems may interact in complex ways. Further complexity results from the requirement to cope with run-time errors in components and connection systems, in order to ensure SoS dependability.

The likely complexity of SoS behaviour strongly indicates the need for validation techniques to include largeness avoidance and tolerance strategies to avoid the need to construct large models of system behaviour, and tolerate such a need.

5.1.4 Characteristics of SoS properties and implications for description/validation

Here we identify characteristics of desirable SoS properties that are relevant to description/ validation, either because they affect the suitability of ADLs or IDLs for describing SoSs and their interfaces, or because they affect formal validation of SoSs thus described. For each characteristic, we identify its implications for the capabilities of (a) IDLs and ADLs (when used to describe SoSs), and (b) formal validation techniques used in the DSoS context..

5.1.4.1 Types of properties

Recall that dependability means trustworthy delivery of intended services. So validation of functionality properties and availability/reliability/performance properties is required.

5.1.4.2 Decomposability of properties

For any SoS, we can expect that some properties will be decomposable into necessary and sufficient independent properties of its parts (the component systems, connection systems and connections). We call such properties *decomposable.*

In principle, all desired properties of a SoS can be decomposed into ('pushed down to') sufficient, independent properties of its parts[4]. However, some properties may not be decomposable into independent properties of its parts that are both necessary and sufficient. For example, consider the system depicted in figure 2, a two-place buffer built from two one-place buffers A and B.

[4] In the extreme, one can decompose any SoS-level property into the property FALSE for each part. Then, it is trivially the case that if FALSE holds then the SoS-level property is true.

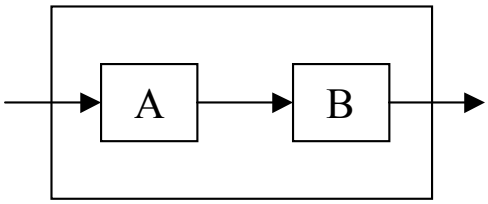

Figure 10: A two-place buffer with two different component one-place buffers

Suppose we wish to check the property that the system always transfers data in and out again in less than 10 seconds. We can choose the necessary and sufficient properties: A takes x<10 seconds to transfer the data and B takes <10-x seconds, but these are not independent. In fact, we cannot find properties of the delays across the (possibly different) one-place buffers A and B that are necessary and sufficient, yet independent of each other. We might choose sufficient independent properties, but would lose necessity (e.g., A takes at most 4 seconds to transfer its data, and B takes less than 6 seconds, or (simplest) A and B each take less than 5 seconds).

A very useful validation strategy for decomposable SoS-level properties is the validation of the necessary and sufficient independent component properties. By definition, these are sufficient to imply the SoS-level property. The necessity of the component properties means that the validation effort is focussed precisely on the SoS-level property desired.

Notice that the context of SoS validation is likely to be such that component systems are outside the sphere of control of SoS engineers. This reduces the relevance of property decomposability, since decomposition of a SoS-level property may lead to component properties that are not satisfied by the available component systems. On the other hand, in many cases it will be possible to employ wrappers around component systems (see Section 4 of IC2 (2001)), which enable the use of non-ideal component systems by changing the demands made on them by the SoS.

Possible strategies for validating a non-decomposable SoS-level property include:

1. Map the SoS-level property to some sufficient and independent (so not necessary) component properties, and try to validate these component properties, perhaps using some simple technique. If unsuccessful then try to validate successively less restrictive, but still sufficient, component properties and/or use more sophisticated techniques. Continue iterating until the SoS-level property is validated or we choose to abandon the attempt.

2. Map the SoS-level property to some necessary and sufficient (so not independent) component properties, and use some technique to validate these properties together, rather than in separate steps.

A good example of the systematic decomposition of a high-level property into a set of low-level properties occurs in the rely/guarantee paradigm (Misra and Chandy 1981), (Jones 1981, June), (Jones 1983), (Collette and Jones 2000). This paradigm decomposes a high-level property into a set of low-level 'rely/guarantee' properties, each of which is of the form "if rely(cond) then guarantee(prop)". These properties mean that if 'cond' is true, then 'prop' is also true. A properly constructed set of rely/guarantee properties will together imply the high-level property. Steps are: (1) decompose the high-level property into a set of rely/guarantee properties of components (with simple consistency conditions), (2) check these rely-guarantee properties.

5.1.5 Abstraction

Roughly speaking, abstraction is a procedure whereby one or more particular types of information about an entity are deliberately ignored, in order to postpone consideration of some aspect(s) of that information. A comprehensive discussion of abstraction and related issues appears in *Abstraction, Encapsulation and Information Hiding* (http://www.toa.com/pub/abstraction.txt). It is common to speak of information *at a given level of abstraction*, i.e., the information that remains after the information of the type(s) ignored by the given abstraction is removed.

Within the DSoS Project, we consider abstraction of information about systems, including SoSs. In this context, there are various types of abstraction, including:

1. **architecture abstraction**
   The information ignored regards the distinctions between some component systems (this corresponds to treating those component systems together as a single system). For example, one might ignore the distinction between components C and D in Figure 9and so treat them as a single, combined system (dotted system CD in the figure).

2. **data abstraction**
   The information ignored regards the data values stored by component systems and/or communicated between them or between the SoS and its environment. For example, one might choose to abstract away from the value of an integer output, only recording whether the value is even or odd.

3. **communication abstraction**
   The information ignored regards the means by which communications are achieved, between component systems or between the SoS and its environment. For example, one might abstract away from the details of a communication protocol, retaining only the information regarding the information transferred by a successful run of that protocol (e.g., ignore ACKs).

Of course, the usefulness of an abstraction depends upon its purpose. Here, we consider abstractions for the purpose of assisting the modeling of SoSs, particularly in order to facilitate validation of dependability.

5.2 Architecture Description

5.2.1 Proposed UML-based ADL

Architecture Description Languages (ADLs) are notations enabling the rigorous specification of the structure and behaviour of systems (Medvidovic and Taylor 2000). Several ADLs proposed in recent years are all based on the same principle: specifying the structure of systems using the following basic concepts: components, connectors and configurations (described below).

The DSoS report "Architecture and Design: Initial Results on Architectures and Dependability Mechanisms for Dependable SoSs" (2001) proposes an ADL defined in relation to standard UML elements. The proposed ADL is being developed by the definition of a set of core extensible language constructs for the specification of components, connectors and configurations. The intention is that these extensible constructs will enable a variety of ADLs to be mapped into UML.

The definition of the proposed DSoS ADL environment is based on UML for a number of good reasons, detailed in (2001). One of the most important reasons is the prevalence of UML as a notation – it is widely used by Industry and is therefore likely to minimize resistance by Industry to the take-up of the emerging DSoS methodology.

5.2.2 Components, connectors and configurations

It is now accepted by the vast majority of the software architecture community that the description of a system architecture should be based on Components, Connectors and Configurations. These terms are discussed in detail in Chapter 1 of the DSoS State of the Art Survey (2000). Briefly:

- ***Components*** abstractly characterize units of computation or data stores.
  In general, the specification of a component gives the behavioural specification of the component together with the component's interfacing points with other architectural elements.
- ***Connectors*** abstractly characterize composition patterns among components.
  A connector thus prescribes the interaction protocol that takes place among the components that are composed through it.
- ***Configurations*** define the structures of systems by composing collections of component instances through bindings *via* connector instances. A system's software architecture is then defined as a configuration together with the component and connector types that are instantiated within the configuration.

5.2.3 Extensibility of the proposed ADL

Extensibility is a major consideration in the design of the proposed ADL, as evidenced by its definition using core extensible language constructs. This should enable its use for rigorous architectural description of a wide range of SoSs, since these constructs can be extended to provide particular descriptive abilities necessary for particular systems. It is harder to anticipate all possible architectural features of a system than it is to provide the flexibility to extend an ADL to cope with particular features as the need arises, and not providing this flexibility would unnecessarily constrain the types of systems that can be described (and therefore that can be validated).

The proposed ADL is based on UML, which in turn is meant to be a standard base for the development of a family of languages, called UML profiles. Profiles are defined using UML standard extension mechanisms (e.g., stereotypes, constraints, etc.). Those mechanisms can be used to extend the definitions of the base ADL elements, as needed.

5.3 Interface Description

This section summarizes OMG IDL and discusses its strengths and weaknesses with respect to the provision of suitable capabilities for describing the interfaces of component systems of a system of systems.

5.3.1 Summary of OMG IDL

Establishing a communication between two subsystems requires that all properties match. If we focus on the data properties there are a number of different aspects:

- Representation of data
- Structure of data
- Typing of data
- Meaning of data

In order to support the communication among heterogeneous systems the Object Management Group (OMG) has defined a semiformal Interface Definition Language (IDL) to avoid data property mismatches at the representation layer and structure layer. The syntax of this language is similar to the programming language C and so are the basic data types. The following list contains some of the types available in OMG IDL:

- *boolean*: may have two values only (TRUE and FALSE)
- *char*: 8 bits value for characters
- *octet*: 8 bits unsigned value (is not subject of conversions)
- *short* and *unsigned short*: 16 bits integer value
- *long* and *unsigned long*: 32 bits integer value
- *long long* and *unsigned long long*: 64 bits integer value
- *float*: IEEE single-precision floating point
- *double*: IEEE double-precision floating point
- *long double*: IEEE double-extended floating point

Additionally to these basic types it is possible to define user-defined types like *struct* or *union*, or use several instances by using *sequence* or *array*.

In order to avoid the restriction that static typing imposes, two additional types exist in OMG IDL: *any* and *DynAny*. When the *any*-type is used for a method any predefined type in the IDL-file can be used. The *DynAny*-type allows the use of types not predefined in the IDL-file.

In OMG IDL a method may have a valid return value or raise/signal an exception. This mechanism for reporting errors is supported by a number of programming-languages natively (e.g., C++, Java). Other languages provide mechanisms to emulate this behaviour. In real-time systems, exceptions are not widely used because of their interrupt-like nature.

Attributes in conjunction with the associated methods are combined as objects. It is important to mention that objects can only be declared but not defined, i.e., only parameters (input- and

output-parameters) and results of methods (regular result or an exception) are stated, but the algorithms cannot be described.

It must be stressed that OMG IDL defines the system appearance of the exchanged data and operations but not the meaning associated with the structures. It is a background assumption that the client has an informal understanding of the meaning. A formalization of this meaning is an important open issue.

The OMG standard defines language mappings from OMG IDL to C, C++, Java, Smalltalk, COBOL, and Ada. For Java even a reverse mapping is defined (Java to OMG IDL). Although not defined in an OMG standard there exist additional language mappings for some other programming languages like Perl, Common Lisp, Eiffel, or Python. Thus it is possible to compose a system from parts that may be written in different programming languages.

Different computer architectures may use a different representation of data (e.g., byte order or different character-sets). The Common Data Representation (CDR) defines representations for all data types available in OMG IDL. Thus the receiver of a message is able to convert the message into its preferred representation. This strategy minimizes the number of format conversions when messages are exchanged within an architecture. This allows the integration of different computer architectures to be transparent to the user. A syntactic property mismatch as defined in chapter 2.3.2 can't occur.

OMG IDL supports synchronous interfaces and asynchronous interfaces. The synchronous interface allows a client to wait for a result or to continue immediately (in which case no result may be retrieved). The asynchronous interface allows an event-triggered (callback) or a time-triggered (polling) retrieval of the result. The different types of flow-control provide the flexibility to choose the interface most suitable for the application.

CORBA provides two ways of using a method of another object: The Static Invocation Interface (SII) and the Dynamic Invocation Interface (DII).

The SII can be used like procedures or functions in most programming languages. Although this interface is restricted to methods known at compile-time, it provides some advantages. If one abstracts from the temporal properties, one need not be aware if the function or procedure is defined locally in a library or if it is a CORBA-method, which will call an object on a remote location. Furthermore, this interface provides type checking at compile-time. A static invocation requires two steps:

1. The object providing the required method must be identified. This can be done by simply knowing the reference to the object or by using the CORBA "Naming Service" or "Trading Service".
2. Then the request is invoked and the results are received.

The DII allows more flexibility by allowing a client to use methods of objects that were designed after compilation of the client. Thus calling a method requires four steps:

1. The object providing the required method must be identified. This step is the same as the first step in the SII.
2. The interface definition must be found out. For this purpose CORBA provides the service "Interface Repository" where the interface definition of objects can be registered.
3. The invocation is constructed.
4. The request is invoked and the results are received. This step is similar to the second step in the SII.

5.3.2 Extensibility of OMG IDL

For an SoS to cope with changing component interfaces, an IDL used to describe those interfaces must have an introspection mechanism that provides information on the syntax of a new interface, and a mechanism for dynamically constructing requests and responses against this interface. In CORBA, these are provided by the Interface Repository and the DII and DSI modules.

There are two approaches to making a syntactic change to an existing interface without breaking backward compatibility: (1) dynamic/latent typing, where clients ignore attributes that they don't understand; and (2) static typing, where new clients bind to a new interface that inherits from the old interface. The CORBA approach includes direct provision for the second approach, but the first approach can be achieved by use of data types such as strings and Anys.

CORBA also provides for signalling of a service change. A client realises it has a stale object reference when it receives an exception, either raised by the remote ORB to signal that the object whose invocation was requested no longer existed, or raised by the remote application. Leasing is also provided by CORBA, where object references automatically expire after a certain time. A client can be recompiled to support a new interface, or alternatively it can obtain a syntactic description of the new interface from the Interface Repository and dispatch

requests against that interface using the Dynamic Invocation Interface. The client obtains an object reference for the new service by using the naming or trading service.

5.4 Interaction Description

This section focuses on the formal description of the interactions between component systems of a SoS for the purpose of formal validation of SoS dependability properties.

5.4.1 Formal validation techniques

Formal validation techniques fall into two major categories, which we refer to as theorem proving and model-checking techniques:

1. In theorem proving techniques a logic (e.g., the HOL logic (Gordon and Melham 1993)) is defined, and a model of the system to be validated is expressed using this logic. Conjectures are then generated that represent the satisfaction, by the system, of the desired properties. A *theorem prover* (e.g., Isobelle/HOL (Paulson 1994)is then used to prove these conjectures, thus formally proving that the properties hold of the system (under the assumption that the system and properties are represented correctly).
2. In model-checking techniques a model of the system is constructed. The modelling language typically either (a) employs a sequential imperative programming language, together with a shared-variable computational model (e.g, Promela (Holzmann 1993), SMV (McMillan 1993)); or (b) is a process algebra (e.g., CCS (Milner), CSP (Hoare 1978), (Roscoe 1998)). In case (a), system properties are typically expressed as logical expressions using some sort of temporal logic over execution paths of the system. In case (b), system properties are usually expressed as refinements between a model of the property and the system model (when using CSP), or as bisimulations (when using CCS). In all cases, a *model-checker* is then used to formally check the properties of the system (under the assumption that the system and properties are represented correctly).

Theorem provers typically require a large amount of expertise if they are to be used effectively. Model-checkers typically require less expertise, but expertise is still highly desirable in all but the most straightforward of application domains.

We focus this discussion on model checking using the process algebra CSP (Hoare 1978), in particular the version of CSP described in (Roscoe 1998). This version of CSP (strictly, its machine-readable form CSPm (Scattergood 1998)) is supported by the model-checker FDR2 (1992-99). The interested reader is referred to (2001) for details regarding the use of PROMELA and SPIN in the DSoS Project.

5.4.2 Modeling systems using CSP

The process algebra CSP is well suited to the formal description of SoSs. In general, CSP models consist of a number of processes composed using process operators. These operators include choice operators (which model the ability of a process to behave in alternative ways, either deterministically or non-deterministically) and parallel operators (which model a process as a set of component processes executing in parallel and interacting over one or more channels).

It is very natural to model component systems and connection systems of a SoS as CSP processes, and model the SoS itself as a parallel combination of these processes. A component system will have some interface(s) through which it is expected, by the SoS designer, to communicate with other component systems and with the environment of the SoS. Models of these interfaces would form part of the component system models.

5.4.3 Validation by compositional reasoning

In the context of CSP, compositionality means that if an implementation process IMPL is a refinement of a specification process SPEC, and a system that contains SPEC is a refinement of another specification process CSPEC, then the containing system, except with SPEC replaced by IMPL, is a refinement of CSPEC.

In CSP, compositionality can be expressed as follows:

CSPEC [M= C[SPEC] /\ SPEC [M= IMPL then CSPEC [M= C[IMPL]

where [M= represents refinement in semantic model M (for example, P [T= Q means P is traces refined by Q, i.e., all the traces of Q are traces of P) and C[.] represents the given process operating in context C. A CSP context is effectively a function, defined using any CSP operators, from one or more CSP processes to a CSP process.

The compositionality of CSP is very useful for splitting validation into a collection of simpler validation steps. In particular, a traces refinement SPEC [T= P1 || P2 can be validated by checking SPEC [T= P1 and SPEC [T= P2 separately. (Note that this is because P [T= P || P is true for all processes P; the corresponding properties are not true in the stable failures and failures/divergences semantic models of CSP (Roscoe 1998)).

5.5 CSP Models of CORBA Protocols

This section presents some CSP models of a CORBA middleware protocol (GIOP) and describes the formal validation activities that have been performed using the models. This work demonstrates the potential to use CSP and FDR2 (a model-checker for CSP) for validating dependability properties of real protocols that are likely to be used in SoSs.

5.5.1 Common object services and CORBA facilities

The *Common Object Request Broker Architecture* (CORBA) is the *Object Management Group's* (OMG's) middleware for enabling the provision of services between distributed, heterogeneous objected-oriented systems.

CORBA supplies developers with a basic suite of *Common Object Services* (COS). The COS include naming services, event and time services, transaction and concurrency services, and a basic security service. These are 'low-level services' which facilitate the development of objects independently of the application domain.

In addition, CORBA provides its user community with an evolving set of *Common Facilities* and *Domain Facilities*. The former are facilities that one would expect to be useful to any IT-based business domains; they include facilities for Systems Management and User Interfacing. The latter are business-domain specific facilities, including facilities for electronic commerce, accounting, medical and healthcare, and telecommunications.

5.5.2 ORBs and GIOP

The *General Inter-ORB Protocol* (GIOP) is the CORBA protocol by which the CORBA *Object Request Brokers* (ORBs) communicate method invocations on behalf of the CORBA objects that they host. In keeping with the CORBA doctrine for architecture and vendor neutrality, GIOP is a transport-neutral protocol, i.e., it is designed to run over any connection-oriented transport-level protocol that meets a minimum set of requirements. The GIOP protocol makes no assumptions about how different vendors' ORBs are implemented, or about their runtime environment.

The 'base-line' transport-level protocol of GIOP is the TCP/IP protocol suite; the particular mapping of GIOP to TCP/IP is called the *Internet Inter-ORB Protocol* (IIOP). An ORB must support IIOP in order for its vendor to claim compliance with the CORBA standard. However, ORBs may support other mappings of GIOP to 'environment-specific' or proprietary protocols.Such mappings are called *Environment Specific IOPs* (ESIOPs). One of

the most important ESIOPs is the *DCE-ESIOP*, designed to facilitate communications between CORBA-compliant and DCE-compliant systems.

5.5.3 The CSP modeling of GIOP

5.5.3.1 Modeling context

The CSP GIOP modeling presented in an accompanying report owes much to the research reported in (Kamel and Leue November 1998). In that research, the GIOP protocol was modeled in PROMELA, and basic properties verified of the system using the Spin model checker (Holzmann 1997).

The aim of our CSP modeling is twofold: firstly, to demonstrate that CSP and FDR are sufficiently advanced to model and verify complex object interactions via GIOP; secondly, to demonstrate the use of CSP Data Independence techniques to formally extrapolate the results of the modeling – which is necessarily finite-state – to arbitrarily large systems. The latter is not easily achievable in PROMELA/Spin. We did not expect our modeling of the GIOP protocol to reveal any hitherto unknown significant design flaws or holes – given that CORBA is popular and widely used, it is likely that, by now, any problems that do exist are already known and/or are of a highly pathological nature. Even so, it sometimes happens that formal validation discovers an error; it is often beneficial to formally validate in order, one hopes, to confirm expectations of correct behaviour.

For the purposes of realism, and for brevity, it is expedient that we make assumptions about the underlying transport-level protocol – for example, addressing information is necessarily transport-specific. Our modeling assumes that the underlying protocols are TCP/IP – so, in effect, we are modeling the IIOP mapping of GIOP.

There is plenty of scope for further refinement of the models presented here – for example, the incorporation of TCP/IP idiosyncrasies, message fragmentation, and different threading models.

5.5.3.2 Overview of the model

For this deliverable, we have modelled basic ORB processing of GIOP messages.

The OMG allows vendors considerable leeway in the way in which they implement ORBs. This is reflected in our models. We have endeavoured to design a simple, but ‘fair’ ORB that will guarantee that all invocation requests are eventually serviced under non-pathological conditions. Those conditions are detailed in the annotations of the CSP scripts.

Our *base case* model is of a two-ORB CORBA environment in which up to five objects may be *instantiated* on either machine, and those objects can *relocate* at will. (Object *relocation* is defined in the annotations of the GIOP1.csp, see below.)

By introducing the concept of object relocation early on, we were able to resort to the Data Independence theory of (Lazic and Roscoe July 1998) in order to extrapolate our results for an arbitrary number of objects (five is the *threshold* cardinality of objects in our models).

Without free object relocation, we would have had to resort to more advanced data independence arguments, such as D.I. Induction (Creese and Roscoe June 2000) with no guarantee of success. However, free object relocation can lead to pathological cases in which a server object persistently re-locates and a (prospective) client ORB cannot 'catch up' with it. This anomaly was described in (Kamel and Leue November 1998). In that study, the authors proposed a solution based on constraining the number of times an object is allowed to relocate. We propose an alternative solution that imposes no constraints on the number of relocations. This solution relies on the explicit 'fairness' that has been built into our ORBs. We have not, however, formally verified our proposed solution (i.e., through CSP/FDR).

Finally, we have described how the results of certain 2-ORB CSP models (such as those we present in this report) can, in principle, be extrapolated to arbitrary n-ORB implementations by resorting to simple *compositionality* arguments only. Such an argument, however, would necessitate a significant weakening of our ORB functionality: client objects themselves would have to re-send requests to relocated objects, rather than rely on the ORBs to automatically do this for them. Such a weakening of the ORBs would, among other things, mean that we could not legitimately impose our 'persistent object-relocation' solution without, potentially, introducing deadlock into the CORBA environment.

5.5.3.3 Running the CSP scripts

The GIOP modelling is composed of three CSP_M scripts:

- GIOP1.csp – the definition of the GIOP message datatype;
- GIOP2.csp – the model of the TCP/IP communication layer;
- GIOP3.csp – the model of an ORB's processing of the GIOP protocol.

GIOP3.csp is the top-level script that is loaded into FDR. This script instantiates certain data independent types (primarily the objects) whose threshold cardinalities can only be calculated by analysis of the implementation and specification models present in that script.

The assertions in GIOP3.csp are designed to prove two properties:

- that our ORB implementations cannot, in themselves, introduce deadlock into the CORBA environment;
- that method invocations are 'fairly' processed by the ORBs in transport-failure-free conditions, and that methods are processed *once-only*.

The GIOP3.csp script was run through FDR version 2.66 on a Dell Precision 420 machine running Red Hat Linux 7.0. All the assertions passed. The Data Independence theory of Lasić and Roscoe allow us to formally extrapolate the assertions to an arbitrary number of objects (but not an arbitrary number of ORBs).

5.5.3.4 Data independence thresholds

Here we describe calculation of the thresholds on the cardinality of objects, *MaxNoOfObjects*, and the cardinality of the Request Identifiers nametype, *request_IDs*. These thresholds are calculated from a semantic analysis of the *CORBA_ENVIRONMENT*, *SPEC_3_1* and *SPEC_3_2* models (presented in the GIOP3.csp script) according to Theorem 15.2.3 of (Roscoe 1998), as follows:

- *CORBA_ENVIRONMENT*, *SPEC_3_1* and *SPEC_3_2* are Data Independent in the object type (i.e., {1.. *MaxNoOfObjects*}) and in the *request_IDs* type. (A formal description of what it means for a CSP process to be *Data-Independent* in a particular type can be found in Section 15.2.2 of (Roscoe 1998).)
- *SPEC_3_1* and *SPEC_3_2* both satisfy *Norm* (see Section 15.2.2 of (Roscoe 1998) for the definition of *Norm*).
- Section 15.2.2 of (Roscoe 1998) defines W^{Impl} to be the maximum number of values of the D.I. type that the implementation (in this case *CORBA_ENVIRONMENT*) ever has to store for future use. For the objects and for the request identifiers, this is 4 (the ORB in the role of 'client' stores one of each value, in the role of 'server' it stores one of each value, and there are two ORBs in *CORBA_ENVIRONMENT*).
- W^{Spec} is a specification's equivalent of W^{Impl}. For both types, W^{Spec} is 0 in *SPEC_3_1* and in *SPEC_3_2*. (Both specifications involve a single input of both types before recursing.)
- Section 15.2.2 of (Roscoe 1998) defines $L_?^{Impl}$ to be the largest number of values of the D.I. type that can be input in *any single* visible event of the implementation. For the objects, $L_?^{Impl}$ is 1 (the *user_invoke!this?* event in the ORB in the role of client, and, implicitly, on the receipt of a *Request* message by the ORB in the role of server). For the

request identifiers, too, $L_?^{Impl}$ is 1 (implicitly, on the receipt of a *Request* message by the ORB in the role of server).

- L^{Spec} is a specification's equivalent of $L_?^{Impl}$. For both *SPEC_3_1* and *SPEC_3_2,* L^{Spec} is 1 for both the object and request identifier types (via the *send!host_machine?_...* input).
- Section 15.2.2 of (Roscoe 1998) defines $L_{|\sim|}^{Impl}$ to be the largest number of values of the D.I. type that can be non-deterministically chosen in any single non-deterministic choice in the implementation over the type. As *CORBA_ENVIRONMENT* has no non-determinism, this is trivially 0 for both the objects and request identifiers.

Theorem 15.2.4 of (Roscoe 1998), then states that the threshold value of the types in the *traces* and *failures* models are given by:

$$W^{Spec} + W^{Impl} + \max(L^{Spec}, L_?^{Impl}, L_{|\sim|}^{Impl})$$

That is, 0+4+max(1,1,0) = 5. This means that GIOP3.csp's deadlock freedom test and the assertions involving *CORBA_ENVIRONMENT, SPEC_3_1* and *SPEC_3_2*, if true for #objects=5 and #request identifiers=5, are true for any number (>=5) of objects and request identifiers.

5.6 Modelling CA-Actions

The Coordinated Atomic (CA) action concept is an approach to structuring complex concurrent activities in a distributed environment, aimed at supporting fault-tolerance in object-oriented systems.

Several models have been proposed for formalising the CA action concept with the intention either to give a more complete and rigorous description of the concept or to verify systems designed using CA actions.

These are four approaches falling into the first category.

- The concept of Dependable Multiparty Interactions has many similarities with that of CA actions, and is formally specified using Temporal Logic of Actions TLA (Zorzo 1999). There were several earlier attempts to specify the CA action semantics using TLA (for example, the one reported in (Schwier, Henke et al. 1997)).
- The COALA framework (Vachon 2000) was proposed to allow system developers to model complex distributed/concurrent systems. Within this work a formalization of the CA action concept is developed using CO-OPN/2: an object-oriented language based on Petri nets and partial order-sorted algebraic specifications.

- The ERT model (ERT stands for extraction, refusals and traces) is used for formalising the CA action concept (Koutny and Pappalardo 1998). Refusals and traces are terms that come from semantic models of CSP; term extraction refers to a specific technique used to relate systems specified at different levels of abstraction.
- A mathematical framework, based on Timed CSP, for representing the use of CA actions in real-time safety-critical systems is proposed in (Veloudis and Nissanke 2000). It allows the interactions between concurrently functioning pieces of equipment to be modelled – and their behaviour to be reasoned about – in an abstract way. The framework models dynamic system structuring using CA actions and explicitly uses events representing synchronization between items and the control system to allow the action context to be changed dynamically. Although the framework was not developed for dealing with erroneously behaving action participants, it helps gain a better understanding of the CA action concept and can be used in developing general models incorporating mechanisms supporting system safety.

The following research belongs to the second category.

- A formal approach is used to model and verify a safety-critical system designed using CA actions in (Xu, Randell et al. 1999). To model-check the system controlling a fault-tolerant Production Cell, the state transition system corresponding to a CA action based design is expressed in SMV (Symbolic Model Verifier) and the properties of the system to be analysed are expressed in CTL.

Currently, we are working on the modelling of CA actions using the most recent version of the ERT model (Burton, Koutny et al. 2001). We expect to improve both the precision and generality of the CA action model and to prepare tools that will allow compositional reasoning about designs based on CA actions.

6 SUMMARY AND FUTURE WORK

In this deliverable, we have presented a revised version of the DSoS conceptual model, and illustrated some of the concepts using a series of examples. The revised Section 1 positions the objectives of the document.

The taxonomy that we have presented in Section 2 attempts to summarize the large number of different classificatory dimensions that could be of use for characterizing and comparing different systems of systems.

General DSoS concepts were introduced in Sections 3 and 4. With respect to the previous version of the conceptual model, extensive work has been carried out to improve consistency of the concepts and the corresponding definitions, and to widen the scope of the types of SoS that they embrace. Indeed, as indicated in Section 1.2, one of the greatest challenges of the conceptual model is to provide useful definitions that cover the wide range of systems considered.

Section 5 gives our initial views on formalization.

Several aspects need to be developed further in future revisions of the conceptual model. For example:

- The taxonomy presented in Section 2 needs to be tested against examples of systems of systems, and appropriately extended.
- Whereas the current conceptual model has several times discussed faults that need to be taken into account in dependable systems of systems, an appropriate fault model has yet to be explicitly defined. The fault model needs to cover classic faults in distributed systems, together with specific SoS fault types like dynamic mismatch.
- One of the major objectives of the DSoS Project is to investigate how to build dependable systems of systems out of (undependable) legacy component systems. The dependability enhancement will be provided by special connection systems that perform error detection and reconfiguration or fault masking. This topic is covered in Work Package 2 (Architecture and Design) and the appropriate fundamental concepts (e.g., wrappers for error confinement, CA Actions) will be included in the final version of the conceptual model.

ANNEX 1. MODELS OF TIME

In the following paragraphs we develop further the models of time that are part of the conceptual model of the DSoS Project.

Events and States: The flow of real time can be modeled by a directed *timeline* that extends from the past into the future (Whitrow 1990). A cut of the timeline is an *instant*. Any occurrence that happens at an instant is called an *event*. There can be many events happening at a single instant. Instants are totally ordered, events are only partially ordered. Information that describes an event is called *event information*. Event information is *non-idempotent* and requires exactly-once semantics when transmitted to a consumer. The present instant, *now*, is a very special instant that separates the past from the future (the presented model of time is based on Newtonian physics and disregards relativistic effects). An interval on the timeline is defined by two instants, the *start event*; and the *terminating event* of the interval. The *duration* of the interval is the time of the terminating event minus the time of the start event, measured in some metric (see below). Any property of an object that remains valid during a finite duration is called a *state attribute* and the corresponding information *state information*. State information is *idempotent* and requires an at-least once semantics when transmitted to a consumer. A change of state is an event. An observation is an event that records the state of an object at a particular instant, the point of observation. An event observation can be expressed by the atomic triple:

<Name of the observed event, attributes of the event, time of the event>

A *trigger* is an event that causes the start of some action, e.g., the execution of a task or the transmission of a message. Depending on the triggering mechanism for the start of communication and processing activities in each node of a distributed computer system, two distinctly different approaches to the design of real-time computer applications can be identified (Kopetz 1993; Tisato and DePaoli 1995): the event-triggered and the time-triggered approach. In the *event-triggered* (ET) approach, all communication and processing activities are initiated whenever a significant change of state, i.e., an event other than the regular event of a clock tick, is noted. In the *time-triggered* (TT) approach, all communication and processing activities are initiated at predetermined instants. While ET systems are flexible, TT systems are temporally predictable.

Physical Clock: A (*physical*) *clock* is a device for measuring time. It contains a counter, and a *physical oscillation mechanism* that periodically generates an event to increase the counter. A clock partitions the time line into a sequence of nearly equally spaced intervals, called the

granules of the clock, which are bounded by special periodic events, the ticks of the clock. Whenever an observer perceives the occurrence of an event e, she/he will instantaneously record the current state of the clock (the current granule) as the time of occurrence of this event e, and, will generate a timestamp for *e*. A *Clock* (*event*) denotes the timestamp generated by the use of a given clock to timestamp an event. The granularity of any digital clock leads to a digitalization error in time measurement. Since any two clocks will have slightly different physical oscillation mechanisms, the time-references generated by two clocks will drift apart, if the clocks are not periodically resynchronized. Even if the clocks are properly synchronized, there is always the possibility that an external event is observed by two clocks with a tick difference. This tick difference, which is unavoidable in a distributed system, can cause the loss of replica determinism (Poledna 1995) of two replicated systems.

Dense time: Assume a set of events {E} that are of interest in a particular context. This set {E} could be the ticks of all clocks, or the events of sending and receiving messages of the nodes of a distributed system. If these events are allowed to occur at any instant of the timeline, then we call the time base *dense*. To arrive at a consistent view among a set of nodes about the order of the events that occur on a dense time base of a distributed system, the nodes must execute an *agreement protocol.* The first phase of an agreement protocol requires an information interchange among the nodes with the goal that every node acquires the differing local views about the state of the observation of every other node. At the end of this first phase, every node possesses exactly the same information as every other node. In the second phase of the agreement protocol, each node applies a deterministic algorithm to this consistent information to reach the same conclusion—the commonly agreed value. In the fault-free case, an agreement algorithm requires an additional round of information exchange as well as the resources for executing the agreement algorithm (see also (Kopetz 1997)). Agreement algorithms are costly, both in terms of communication requirements, processing requirements, and — worst of all — in terms of the additional delay they introduce into a control loop. It is therefore expedient to look for solutions to the ordering problem that do not require these additional overheads.

Sparse Time: If the occurrence of significant events that are to be observed is restricted to some active intervals of duration ε with an interval of silence of duration Δ between any two active intervals, then, we call the time base *ε/Δ-sparse*, or simply *sparse* for short (Kopetz 1992). If a system is based on a sparse time base, there are time intervals during which no significant event is allowed to occur. If the intervals ε and Δ are properly chosen (see, e.g., (Kopetz 1997), then, it is possible to establish a consistent order of the significant events among a set of properly synchronized nodes without the execution of an agreement protocol.

It is evident that the occurrences of events can only be restricted if the given system has the authority to control these events, i.e., these events are in the sphere of control of the computer system (Davies 1979). For example, within a distributed computing system the sending of messages can be restricted to some intervals of the timeline and can be forbidden at some other intervals. The occurrence of events outside the sphere of control of the computer system cannot be restricted. These external events are based on a dense time base.

If there is a global time available among a set of DSoS component systems, we assume that the macrotick granularity of this global time base is a negative power-of-two of the physical second. Considering the reasonableness condition, the achieved precision determines which negative power-of-two of the second is selected for the macrotick granularity. By restricting the macrotick granularities to the negative powers-of-two of the full second it is ensured

- that a consistent time base for the measurement of events in the different component systems of a distributed system is established and
- that a full second tick can be generated by a simple binary counter that counts the macroticks of the global time base.

ANNEX 2. GLOSSARY

This glossary contains an alphabetized list of all the terms for which explicit definitions are given above.

Actuation (Sensing) Operation: The production (recording) by a system at a physical output (input) interface of a single value change at an instant or of a temporally-controlled sequence of value changes during a duration.

Architectural style: a set of rules and conventions governing the interactions between the components of a system.

Behaviour: The temporal sequence of send operations of a system in relation to its previous receive operations, and any internal state that it retains.

Boundary Line: A connection between at least two interfaces with matching properties.

Connection System: A new system with at least two interfaces that is introduced between interfaces of the connected component systems in order to resolve property mismatches among these systems (which will typically be legacy systems), to coordinate multicast communication, and/or to introduce emerging services.

Connection: A link between the interfaces of two or more interacting systems.

Declared State: At a given instant, the values assigned to a declared data structure that can be accessed via an interface and that synthesizes all relevant effects of previous receive operations up to the given instant.

Dependability: The dependability of a system is the ability to deliver a service that can justifiably be trusted, where the service is the intended behaviour of the system.

Duration: A section of the timeline.

Error Containment Region: A subsystem of a computer *system* that is encapsulated by error-detection *interfaces* such that there is a high probability (the *error containment coverage*) that the consequences of an *error* that occurs within this subsystem will not propagate outside this subsystem without being detected.

Error: An error is that part of the system state that may cause a subsequent failure.

Event Message: A message that contains only event observations.

Event Observation: An event observation records the occurrence of an event. An event is a significant happening, e.g., an *important difference* between a state observation immediately *before* the happening and the state observation immediately *after* the happening.

Failure: A failure is an event that occurs at the instant when the actual behaviour of a system starts to deviate from the intended behaviour.

Fault Containment Region: A set of components that is considered to fail (a) as an atomic unit, and (b) in a statistically independent way with respect to other fault containment regions.

Fault Tolerance: Methods and techniques aimed at providing the intended system behaviour in spite of faults.

Fault: A fault is the *adjudged* or *hypothesized* cause of an error.

Image: A representation of a state variable at the receiver.

Input Interface: An interface at which information is consumed from the environment of the system.

Instant: A cut of the timeline.

Interaction: A sequence of message exchanges between connected interfaces.

Interface: A point of interaction between a system and its environment.

Legacy System: An existing system that provides a service to an organization or set of users.

Linking Connection: A connection between two or more existing systems that is introduced in order to incorporate this system into a system of systems with new emergent services.

Linking Interface: An interface of a component system through which it is connected to other component systems within a given system of systems.

Local Interface: An interface of a component system that is not a linking interface within a given system of systems.

Message Receive Instant: The instant when the receiving of a message terminates at the receiver.

Message Send Instant: The instant when the sending of a message starts at the sender.

Message: A data structure that is formed for the purpose of communication among computer systems.

Output Interface: An interface of a system at which information is produced for the environment of the system.

Periodic State Message: A state message that is sent periodically at *a priori* known instants. These instants are common knowledge to the sender and the receivers.

Properties of an Interface: The set of attributes associated with an interface.

Property Mismatch: A disagreement among connected interfaces in one or more of their properties.

Protocol: The set of rules that specifies the interactions between two or more component systems across connected interfaces.

Send (Receive) Operation: The sending (receiving) of a message at an interface.

Service Interface: This is the interface that provides the intended service to the environment, namely the systems with which it interacts.

Service Specification: The specification of the set of intended behaviours of a system.

State Message: A message that contains only state observations.

State Observation: A tuple <*Name,* t_{obs}*, Value*> consisting of the name of the state variable, the instant when the state variable has been observed (t_{obs}), and the observed value of the state variable.

State of a System: At a given instant, the values assigned to an internal data structure of a system that synthesizes all cumulative effects of all receive operations at all input interfaces between the startup of the system and this given instant.

State variable: A state variable is a *relevant* variable, either in the environment or in the computer system, whose value may change as time progresses.

System: An entity that is capable of interacting with its environment and is sensitive to the progression of time.

Temporal Accuracy: An image a *temporally accurate* representation of a state variable at instant *t*, if the duration between the time-of-observation of the state variable and the instant *t* is less than the accuracy interval d_{acc}, an application-specific parameter associated with the dynamics of the given state variable.

Temporal composability: The characteristic that ensures that the temporal properties of a component system are not influenced by the integration of the component system into a system of systems.

The Configuration Planning (CP) Interface: The CP interface is used during the integration or reconfiguration phase to connect a component system to other component systems of an system of systems.

The Diagnostic and Management (DM) Interface: The DM interface provides a communication channel to the internals of the component system for the purpose of diagnosis and management.

References

(1992-99). Failures-Divergence Refinement: FDR2 User Manual, Formal Systems (Europe) Ltd.

(2000). State of the Art Survey,. Newcastle upon Tyne, University of Newcastle upon Tyne.

(2001). Architecture and Design: Initial Results on Architectures and Dependable Mechanisms for Dependable SoSs. Newcastle upon Tyne, University of Newcastle upon Tyne.

Ahuja, M., A. D. Kshemkalyani, et al. (1990). A Basic Unit of Computation in a Distributed System. 10th IEEE Distributed Computer Systems Conference, IEEE Press.

Allen, R. J. and D. Garlan (July 1997). "A Formal Basis for Architectural Connection." ACM Transactions on Software Engineering and Methodology.

Brownbridge, D. R., L. F. Marshall, et al. (1982). "The Newcastle Connection, or - UNIXes of the World Unite!" Software Practice and Experience **12**(12): 1147-1162.

Burton, J., M. Koutny, et al. (2001). Implementing Communicating Processes in the Event of Interface Difference. ICACSD'01, Newcastle upon Tyne, U.K.

Campbell, R. H. and B. Randell (1986). "Error Recovery in Asynchronous Systems." IEEE Trans. Software Engineering **SE-12**(8): 811-826.

CAN (1990). Controller Area Network CAN, an In-Vehicle Serial Communication Protocol. SAE Handbook 1992, SAE Press. **SAE J1583:** 20.341-20.355.

Caprile, C. and P. Tonella (1999). Nomen est omen: Analyzing the Language of Function Identifiers. Sixth Working Conference on Reverse Engineering, IEEE Press.

Clark, D. (2001). "Face-to-Face with Peer-to-Peer Networking." Computer **34**(1): 18-21.

Clarke, I., O. Sandberg, et al. (2000). Freenet: A Distributed Anonymous Information Storage and Retrieval System. ICSI Workshop on Design Issues in Anonymity and Unobservability, Berkeley, CA, International Computer Science Institute.

Collette , P. and C. B. Jones (2000). Enhancing the Tractability of Rely/Guarantee Specifications in the Development of Interfering Operations. Proof, Language and Interaction. M. Tofte, MIT Press**:** 277-307.

Creese, S. J. and A. W. Roscoe (June 2000). Data independent induction over structured networks. In International Conference on Parallel and Distributed Processing Techniques and Applications. PDPTA '00, Las Vegas, USA.

Deline, R. (1999). Resolving Packaging Mismatch. Computer Science Department. Pittsburgh, Carnegie Mellon University**:** 178.

Forman, I. R., M. H. Conner, et al. (1985). "Release-to-release binary compatibility in SOM." ACM Sigplan Notices, Proceedings of OOPSLA 1995 **30**(10): 426-438.

Garlan, D., R. Allen, et al. (1995). Architectural Mismatch or Why it's hard to build systems out of existing parts. Proc. ICSE 17, Seattle.

Gaudel, M.-C. (1994). Formal Specification techniques (invited survey). Proc. 16th IEEE-ACM International Conference on Software Engineering (ICSE'16), Sorrente.

Gordon, M. J. C. and T. F. Melham (1993). Introduction to HOL: A theorem proving environment for higher order logic., Cambridge Univ. Press.

Hauswirth, M. and M. Jazayeri (1999). A Component and Communication Model for Push Systems. Joint 7th European Software Engineering Conference (ESEC) and 7th ACM SIGSOFT Int. Symp. on the Foundations of Software Engineering (FSE-7), Toulouse, France, ACM.

Hauzeur, B. M. (1986). "A Model for Naming, Addressing, and Routing." ACM Transactions of Office Information Systems **4**(4): 293-311.

Hayakawa, S. I. (1990). Language in Thought and Action, Harvest Original, San Diego.

Hoare, C. A. R. (1978). "Communicating Sequential Processes." Comm. ACM **21(8)**: 666--677.

Hoare, C. A. R. (1985). Communicating Sequential Processes, Prentice Hall.

Holzmann, G. (1993). "Design and validation of protocols: a tutorial." Computer Networks and ISDN Systems(25): 981-1017.

Holzmann, G. (1997). The model checker SPIN. IEEE Transactions on Software Engineering.

Jones, C. B. (1981, June). Development Methods for Computer Programs including a Notion of Interference. Programming Research Group Technical Monograph 25. Oxford, Oxford University.

Jones, C. B. (1983). Specification and Design of (Parallel) Programs. Proceedings of IFIP'83, North-Holland**:** 321-332.

Kamel, M. and S. Leue (November 1998). Validation of remote object invocation and object migration in CORBA GIOP using Promela/Spin. Proceedings of the 4th International SPIN Workshop. Paris, France, Ecole Nationale Supérieure de la Télécommunication.

Kopetz, H. (1992). Sparse Time versus Dense Time in Distributed Real-Time Systems. Proc. 14th Int. Conf. on Distributed Computing Systems, Yokohama, Japan, IEEE Press.

Kopetz, H. (1993). "Should Responsive Systems be Event-Triggered or Time-Triggered?" IEICE Trans. on Information and Systems (Special Issue on Responsive Computer Systems).

Kopetz, H. (1997). Real Time Systems: Design Principles for Distributed Embedded Applications. Boston, Kluwer Academic Publishers.

Kopetz, H. (2000). Software Engineering for Real-Time: A Roadmap. Software Engineering Conference 2000, Limmerick, Ireland, IEEE Press.

Kopetz, H., T. Galla, et al. (1999). Specification of the TTP/C Protocol. TTTech, Vienna, Austria, http://www.ttpforum.org.

Kopetz, H. and K. Kim (1990). Temporal Uncertainties in Interactions among Real-Time Objects. Proc. 9th Symp. on Reliable Distributed Systems, Huntsville, AL, USA, IEEE Computer Society Press.

Kopetz, H. and R. Nossal (1997). Temporal Firewalls in Large Distributed Real-Time Systems. Proceedings of IEEE Workshop on Future Trends in Distributed Computing, Tunis, Tunesia, IEEE Press.

Kopetz, H. and W. Ochsenreiter (1987). "Clock Synchronisation in Distributed Real-Time Systems." IEEE Trans. Computers **36**(8): 933-940.

Koutny, M. and G. Pappalardo (1998). The ERT Model of Fault-Tolerant Computing and Its Application to a Formalisation of Coordinated Atomic Actions. Newcastle upon Tyne, Department of Computing Science, University of Newcastle upon Tyne.

Laprie, J. C., Ed. (1992). Dependability: Basic concepts and terminology — in English, French, German, Italian and Japanese. Dependable Computing and Fault Tolerance. Vienna, Austria, Springer-Verlag.

Lazic, R. and A. W. Roscoe (July 1998). Verifying Determinism of Concurrent Systems Which Use Unbounded Arrays., Oxford University Computing Laboratory.

Lee, E. A. (1999). Embedded Software--An Agenda for Research, University of California, Berkely.

McMillan, K. L. (1993). Symbolic Model Checking., Kluwer Academic Publishers,.

Medvidovic, N. and R. N. Taylor (2000). "A Classification and comparison Framework for Software Architecture Description Languages." IEEE Transactions on Software Engineering **26(1)**(Jan. 2000): 70-93.

Milner, R. A Calculus of Communicating Systems. LNCS 92, Springer-Verlag.

Misra, J. and K. M. Chandy (1981). "Proofs of Networks of Processes." IEEE Trans. Software Eng. **7**: 417--426.

OFTA (2000). Software Architecture and Component Re-use. Paris, Masson.

OMG (1999). CORBA Persistent State Service V2.0, Joint Revised Submission, Object Management Group.

OMG (2000). CORBA Externalization Service V1.0, Object Management Group.

OMG (2000). CORBAServices: Common Object Service Specification: Event Service Specification, Object Management Group.

OMG (2000). CORBAServices: Common Object Service Specification: Notification Service Specification, Object Management Group.

OMG (2000). Fault Tolerant CORBA Specification V1.0, Object Management Group.

OSF (1992). Introduction to OSF DCE, Open System Fondation. Englewood Cliffs, N.J, Prentice Hall.

Paulson, L. C. (1994). Isabelle -- A Generic Theorem Prover. . .LNCS 828, Springer Verlag,.

Poledna, S. (1995). Fault-Tolerant Real-Time Systems, The Problem of Replica Determinism. Hingham, Mass, USA, Kluwer Academic Publishers.

Powell, D., G. Bonn, et al. (1988). The Delta-4 Approach to Dependability in Open Distributed Computing Systems. 18th IEEE Int. Symp. on Fault-Tolerant Computing Systems (FTCS-18), Tokyo, Japan, IEEE Computer Society Press.

Radia, S. and J. Pachl (1993). Coherence in Naming in Distributed Computing Environments. 13th Int. Conference on Distr. Computing Systems, IEEE Press.

Roscoe, A. W. (1998). The theory and practice of concurrency, Prentice Hall.

Saltzer, J., D. P. Reed, et al. (1984). "End-to-End Arguments in System Design." ACM Transactions on Computer Systems **2**(4): 277-288.

Saltzer, J. H. (1978). Naming and Binding of Objects. Operating Systems: An Advanced Course. New York, Springer Verlag**:** 1-105.

Scattergood, B. (1998). Tools for CSP and Timed CSP. Oxford, Oxford University.

Schwier, D., F. v. Henke, et al. (1997). Formalisation of the CA Action Concept Based on Temporal Logic. DeVa - Design for Validation, ESPRIT LTR 20072. **2nd year:** 3-15.

Siegel, J. (2000). CORBA 3--Fundamentals and Programming, OMG Press--John Wiley.

Szyperski, C. (1998). Component Software, Addison Wesley.

Tisato, F. and F. DePaoli (1995). On the Duality between Event-Driven and Time Driven Models. Proc. of 13th. IFAC DCCS 1995, Toulouse France.

Vachon, J. (2000). COALA: a Design Language for Reliable Distributed Systems. Lausanne, Switzerland, EPFL.

Veloudis, S. and N. Nissanke (2000). Modelling Coordinated Atomic Actions in Timed CSP. Formal Techniques in Real-Time and Fault-Tolerant Systems. M. Joseph, Springer**:** 228-239.

Veríssimo, P. (2000). Topological Model. Malicious and Accidental-Fault Tolerance for Internet Applications: Reference Model and Use Cases, LAAS-CNRS**:** 67-74.

Vigotsky, L. S. (1962). Thought and Language. Boston, Mass., MIT Press.

Whitrow, G. J. (1990). The Natural Philosophy of Time. Oxford, Clarendon Press.

Xu, J., B. Randell, et al. (1995). Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery. 25th Int. Symp. on Fault-Tolerant Computing (FTCS-25), Pasadena, CA, USA, IEEE Computer Society Press.

Xu, J., B. Randell, et al. (1999). Rigorous Development of a Safety-Critical System Based on Coordinated Atomic Actions. 29th Int. Symp. on Fault-Tolerant Computing (FTCS-29), Madison, WI, USA, IEEE Computer Society Press.

Xu, J., A. Romanovsky, et al. (1998). Co-ordinated Exception Handling in Distributed Object Systems: from Model to System Implementation. Proc. 18th IEEE International Conference on Distributed Computing Systems, Amsterdam, Netherlands.

Zorzo, A. F. (1999). Multiparty Interaction in Dependable Distributed Systems (PhD Thesis). Department of Computing Science. Newcastle upon Tyne, University of Newcastle upon Tyne.