DSoS

IST-1999-11585

Dependable Systems of Systems

Final Version of DSoS Conceptual Model

(CSDA1)

Technical Report CS-TR-782, University of Newcastle upon Tyne

Report N° 54/2002, Institut für Technische Informatik, Technical University of Vienna

LAAS-CNRS Report No. 02441

Technical Report QINETIQ/KI/TIM/TR030434, QinetiQ Ltd.

Report Version: Deliverable CSDA1

Report Preparation Date: April 2003

Classification: Public Circulation (after review)

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

Project Coordinator: Newcastle University

Partners: QinetiQ, 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

Final Version of DSoS Conceptual Model

Marie-Claude Gaudel[1], Valérie Issarny[2], Cliff Jones[3], Hermann Kopetz[5], Eric Marsden[4], Nick Moffat[6], Michael Paulitsch[5], David Powell[4], Brian Randell[3], Alexander Romanovsky[3], Robert Stroud[3], François Taiani[4]

[1]LRI (Paris, F),
[2]INRIA (Rocquencourt, F),
[3]University of Newcastle upon Tyne (UK),
[4]LAAS-CNRS (Toulouse, F),
[5]Technical University of Vienna (Austria),
[6]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 systems. It is the objective of the DSoS Project to investigate issues related to the integration of 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 organisation or a set of users. A system of systems may thus span different organisations, each one with their own systems that are, as a result, in different spheres of management control. This is precisely why our sense of SoS differs from "just a large system": however many component systems there

are, a system composed of systems that are not controlled by the same organisation typically poses more difficult challenges than does a system that is managed in one piece.

The principal problem 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 addressed.

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

Furthermore, it is typically not possible for the designer of a system of systems to change the component systems. It is, however, important to recognise that those component systems might continue to evolve without consultation. One potential failure mode of a system of systems is the result of a change to interfaces of its (separately controlled) component systems. 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 recognised 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 information systems (of systems). The aim of this document is to define a collection of concepts that can embrace, for example, inter-organisational 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-fertilisation. For example, the concept of time has been extensively studied in vehicle

[1] Technical University of Vienna

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.

1.3 Some pre-definitions

1.3.1 A first look at the notion of "system"

A precise characterisation of the concept of system is one of the objectives of Section 3 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. Before addressing this, it is worth looking more closely at what it means to characterise the operations available at an interface.

When discussing systems that are themselves composed of systems, a choice of terminology arises. One may choose to refer to a 'system of systems' and its 'component systems' or to a 'system' and its 'subsystems'. This document, in common with other Project deliverables, uses the former terminology for describing systems in which the components are systems; the latter terminology is used when referring to systems in general.

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* component systems, and component systems that have been *specially developed* for a particular system of systems. The integration of the component systems is realised 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, and incompatible, 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 [Allen and Garlan 1997; Garlan et al. 1995]. 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].

Most non-trivial systems employ a state which captures aspects of the history of interaction[2] with the system (various notions of state are discussed in Section 3.4 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 bookings for future dates. 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; for example, the offer of a reservation might be made at time *t* with a validity period of *d*). The notion of time is so central that it is explored more thoroughly in Section 1.3.3.

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. We have considered a wide variety of SoSs and have chosen two particular SoSs that represent extremes.

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

This deliverable presents the final version of the DSoS conceptual model, which was first

[2] Notice that the state will not normally record (or even represent) the whole of the history of interactions – this point is returned to below.

presented in deliverable BC1 [Kopetz 2000a] and then revised in deliverable IC1 [Jones et al. 2001].

This version is a considerable advance compared to IC1. The main differences are:

1) Section 3 is now more complete: several concepts have been revised, others have been added, and their presentation has been re-organised.
2) Section 4 now contains a discussion of the characteristics of linking interfaces that we consider particularly important for systems of systems dependability.
3) Section 5 has been completely re-written.

Section 2 presents a taxonomy that explores the range of possible systems of systems, and the different factors that could impact upon the dependability of such compositions of systems. Section 3 introduces the set of basic DSoS concepts relating to systems of systems and their dependability, including a model of time. Section 4 discusses a range of interface and connection issues for dependable systems of systems. Section 5 discusses approaches to formalisation. Finally, we conclude by summarising 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.

It is our intention to try to distil the concepts described in this deliverable into a form that can benefit a wider community. We feel there is clear potential for exploiting synergies with the wider community.

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.

It draws principally on the basic concepts and ideas from the work of distributed systems and system architecture research communities, in effect summarising this material from a taxonomical viewpoint.

This taxonomy of systems of systems is organised 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 system 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 (b) being 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.

This topic is explored in more detail in PCE4 [Dobson and Periorellis 2002].

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.

Strategies for coping with evolving interfaces are discussed in detail in [Jones et al. 2002].

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 controlled by a means other than through its normal service interface, or has instead been provided with an "*intercession interface*" (either explicitly by its designer, or implicitly by the enclosing infrastructure). This dimension therefore correlates with the intercession dimension of connections (Section 2.3.5).

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 or more classes of threat.

Unless the component system has been purpose-built for integration in the DSoS, and particularly in the case of COTS component systems that were not built with dependability issues in mind, system integrators may not have sufficient information on these factors available. The validation work reported in deliverable IC3 [Marsden 2001] aims to synthesize information on these factors, through the use of fault injection techniques.

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, and 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 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 non-dependability. 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 component system to another component 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 component 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 component systems.

Data Representation - Representational issues normally only show up at interfaces, not within a component system. To facilitate the interconnection of component 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 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

Similar comments can be made here as are made in Section 2.1.5. 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 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 3.6 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.

2.3.5 Intercession dimension

A connection may allow some form of intercession, i.e. it may allow a component system to affect, configure, or control its behaviour. For instance, a connection system that provides support for mobility of component systems may allow an interacting system to request the use of a lower bandwidth communications protocol when it switches from a fixed to a wireless network.

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, together with explanatory notes.

3.1 Outline

We start by defining what we mean by a **system**. Systems of systems are then defined as special types of system. Time is fundamental to the concept of a system in DSoS. This is largely because useful notions of behaviour and failure require some notion of time, or at least an ordering of events that corresponds to the progression of time. So the simple concepts of 'instant' and 'duration' are defined before proceeding further.

Apart from concerns such as cost, the ultimate thing the user of a system cares about is its **behaviour**. We begin by considering only the simplest notion of system behaviour – traces of activity at system interfaces. Other types of behaviour are considered later in the section.

System **state** is then defined, abstractly, as essentially 'that which determines potential behaviour'. This definition might appear vague at first, but this reflects its generality and is not a weakness. The meaning is in fact precise for any given notion of behaviour. So we present abstract state as a convenience for describing system behaviour. The strongly related concept of 'stored state', which records some aspect of the history of a system, is also defined.

Next, we define some system **dependability** concepts. The definitions are based on some standard definitions of fault/error/failure [Laprie 1992], but have been tailored to suit the DSoS context, influenced by [Jones 2002].

We are then ready to define some concepts that help describe **system interconnections**. Among the most important of these concepts are 'linking interface', 'property mismatch' and 'linking connection'.

Other types of behaviour are discussed next. The impact of the new types of behaviour on the preceding definitions is explored.

The section ends with a more detailed discussion of the DSoS model of **time**.

Our intention is that the concepts described here make sense at different levels of abstraction. Clearly, systems (and systems of systems) are sometimes usefully viewed according to different abstractions, and it is desirable to relate alternative views. Most obviously, one would hope that an implemented system (or SoS) meets its specification, if one exists. Specifications are often just abstract views of system (or SoS) behaviour: they

disregard irrelevant (or unimportant) aspects of behaviour. Non-determinism is considered below – even though all implementations are deterministic – because non-deterministic specifications are often useful. Another reason for emphasising abstractions is that some of the concepts defined here differ only according to the viewpoint; this is evident, below, for various concepts derived from state.

3.2 Systems

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 may be sensitive to the progression of time.

By 'sensitive to the progression of time' we mean the system may react differently, at different times, to the same pattern of input activity, and this difference is due to the progression of time. A simple example is a time-controlled heating system, where the temperature setpoint depends on the current time; such a system may react differently, at different times, to the same sensed temperature.

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 the system: it 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 organisations, for example (though these are not the focus of our project).

A system can be decomposed into interacting *components*, which are sometimes *systems* that can themselves be decomposed. 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.

System of Systems (SoS): A system constructed from autonomous component systems, where *autonomous* means independence with respect to existence, operation and/or evolution (see Section 2.1.1).

A system may comprise an information processing subsystem and a mechanical

subsystem. For example, a smart sensor system comprises a microcontroller and an electromechanical sensing element. The microcontroller calibrates the sensed information and presents this information at the smart sensor interface in a standardized message.

Time is important for a system's failure to be observable by some other system. 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.

Note that the decision to use a dense time model does not prevent one abstracting time to a simpler model, such as integer time, when the system architecture permits. More detailed discussions of the DSoS model of time are contained at the end of this section and in Annex 1.

3.3 Behaviour

Ultimately, what most concerns the user of a system is its behaviour. We focus on the simplest notion of system behaviour in this subsection – traces of activity at system interfaces. Other types of behaviour are considered later in the section.

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

By the environment of a system we mean everything other than the system.

At the physical level, for instance, an interface can exist as a single line (a serial port) or as 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 of a system 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, at most 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.

In a distributed system based on message exchanges among system components, the interface specification can comprise three parts: the syntactic specification, the temporal specification, and the meta-level specification [Kopetz 2002c]:

- A syntactic specification concerns the specification of the data elements that cross the interface; it bridges the gap between the logical level and the informational level [Avizienis 1982]. An incoming (outgoing) syntactic interface specification specifies the structure of incoming (outgoing) messages at this interface (e.g., numbers, operations, and text) and assigns names to the chunks of a structure.
- An incoming (outgoing) temporal interface specification specifies when a message is expected (must be sent): instant, phase, and frequency.
- A meta-level interface specification bridges the gap between the informational level and the user's level [Avizienis 1982]. The meta-level specification establishes the meaning of information chunks in messages that cross the interface. (The chunks are generated according to the syntactic specification.) This is done by defining the semantics of the information chunks and by providing a conceptual model of the interface – the *interface model* – that relates the names of the chunks to the user's conceptual world. This conceptual model must be expressed in concepts that are familiar to the user of the interface services.

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 (including analog systems, digital electronic systems, and computer systems).

The description of 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 below focuses on the operations of sending and receiving messages.

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

In order that errors in a message may be detected, an output assertion and an input assertion can be associated with a message. Such an assertion is a predicate on values of the message, and relevant state variables, that defines an application-specific acceptance criterion [Meyer 1988].

Using such assertions, it is possible to classify messages as shown in Table 1.

Attribute	**Explanation**	**Antonym**
valid	A message is *valid* if its checksum and contents are in agreement.	invalid
checked	A message is *checked at source* (or, in short, *checked*) if it passes the output assertion.	not checked
permitted	A message is *permitted* with respect to a receiver if it passes the input assertion of that receiver. The input assertion should verify, at least, that the message is *valid*.	not permitted
timely	A message is *timely* if it is in agreement with the temporal specification	untimely
value-correct	A message is *value-correct* if it is in agreement with the value specification	not value-correct
correct	A message is *correct* if it is both timely and value-correct.	incorrect
insidious	A message is *insidious* if it is permitted but incorrect	not insidious

Table 1 — Message Classification

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 decides 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 in a message to the computer system controlling the car. If the message can be stored in a queue for a significant interval of time, the consequential uncertainty about the state of the light at the message receive instant can have safety implications.

Behaviour: A sequence of (perhaps timestamped) send and receive operations of a system.

Sometimes it is useful to abstract away from time and consider untimed trace behaviour of a system. In other cases it is not appropriate to abstract away from time. We postpone consideration of other types of behaviour – such as liveness and nondeterminism – to Section 3.7, but note here that, in general, the appropriate notion of behaviour depends on the context. (For example, one might not care whether the system is available, only that it never performs some dangerous action. Alternatively, one might only care about its behaviour at a given interface.)

A system's behaviour is characterised by its send operations, though these of course can be affected by preceding operations.

The appropriate handling of a message at the sender and receiver (update in place, or 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 concepts of *state variables* and *state observations*.

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

Here, a relevant variable is one that can influence system behaviour (recall that behaviour is, in general, defined with respect to some abstraction, or view, of the system). 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[3], the type, the value domain, and the maximum rate of change. The value that is 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 a *state observation*.

State Observation: A record of the value of a state variable. It may be represented as 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 was observed. The recorded time t_{obs} may be NULL, in which case the observation has no timestamp.

When we wish to refer specifically to state observations with timestamps, or not, we will do so explicitly.

State observations may be transported in state messages (defined below) to a receiver, which may reconstruct the dynamics of the environment based on the incoming state messages. State observations, whether timestamped or not, are idempotent and may be communicated using the update-in-place technique. [Powell 2002].

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

Value Accuracy: An image is a *value accurate* representation of a state variable if the interpretation of the image value by the user is in agreement with the semantic content of the state variable at the instant of observation.

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.

Accuracy: An image is an *accurate* representation of a state variable (it is *valid*) at a given instant if it is value accurate and temporally accurate.

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 temporal accuracy and hence the validity of an image is *time-dependent*, so an image may be 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.

---

[3] 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 the state observation immediately *before* the happening and the state observation immediately *after* the happening. An event observation can be represented by the tuple

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

where the *time of the event* field may be NULL, in which case the observation has no timestamp.

As with state observations, when we wish to refer specifically to event observations with timestamps, or not, we will do so explicitly.

For example, *"the temperature of the boiler has increased by 2°C"* is an event observation without a timestamp, and the following are timestamped 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 1:35 p.m."*

Event observations that do not include a timestamp require exactly-once semantics because they are not idempotent. In contrast, timestamped event observations can be considered to be idempotent, since consumers can assume that two events with the same timestamp are duplicates. Timestamped event observations therefore require at-least-once semantics at the consumer [Powell 2002].

Idempotency: An observation is *idempotent* if the effect of processing it more than once can be made the same as the effect of processing it once.

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.

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 either be fixed at design time or negotiated during the operation of the system.

Event Message: A message that contains only event observations.

The two most common approaches for the observation of a dynamic environment are to use either time-triggered state observations or event-triggered event observations. Both approaches allow the relevant aspects of states and events of the environment to be reconstructed at the receiver [Tisato and DePaoli 1995]. Periodic state observations

produce a sequence of equidistant "snapshots" of the environment. These snapshots can be used by the receiver to infer those events that occur within a minimum temporal distance longer than the duration of the sampling period. Starting from an initial state, a complete sequence of event observations can be used by the receiver to infer a 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 (or of an event) is not contained in the message, but inferred from the *receive instant* of the message. In these systems, the jitter of the communication system affects the accuracy of the inferred instant of observation. The varying delay of a *non-timestamped* message in a queue degrades the quality of the delivered observation.

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.

3.4 State

There are two approaches to defining the 'state' of a system: state can be defined according to either (what are called here) the forward looking style or the backward looking style. It is important to recognise that these approaches lead to *different* concepts of state, which are useful for different purposes. A comprehensive literature review [Peti 2002] has recently compared accepted notions of state from a diverse range of engineering disciplines; it is noticeable that these definitions all conform to one or other of the styles described here.

In the forward-looking style of definition:

> At a given instant, the state of a system is a notional attribute of the system that is sufficient to determine its potential behaviour;

In the backward-looking style of definition:

> At a given instant, the state of a system is the total information explicitly stored (in state variables) by the system up to the given instant.

'Potential behaviour' is intended to mean the possible behaviour of the system after the given instant, at all its interfaces, including responses to possible input operations at its interfaces. (Also, recall that for the moment we are only considering trace behaviour, timed or untimed.)

The first definition is more popular with system modellers than with system implementers. It corresponds to the view of a system as a labelled transition system (an LTS) or automaton. Strictly speaking, one would *model* the system as an LTS, where each 'node' in the LTS represents (or *denotes*) the behaviour of the system from some point in its execution onwards. (Indeed, the nodes of an LTS are often called 'states', though this is really just shorthand for the fact that they represent states of the system.)

The second definition is typically favoured by system implementers. It corresponds to the view of a system as a physical entity that stores information about its interaction with the environment, and uses stored information to influence its future behaviour. This concept of state is therefore often called 'stored state' (or 'internal state'). Note that stored state only consists of values explicitly stored in state variables; it does not include static information about the system.

Variant definitions of the concept of a stored state can cause confusion[4]. However, as it stands, the second definition above allows stored state to include autonomously changing state, such as the time recorded by an internal clock (which is not part of the behaviour of the system), and it does not restrict the stored state to relevant stored state – information about the history of the system that can possibly influence its future behaviour.

Consider a physical system that implements a pure function by alternately inputting a single value and then outputting the corresponding function value. Such a 'pure function system' can be viewed at a number of 'levels of abstraction'. Now consider an abstract version that ignores the delay between corresponding inputs and outputs: it repeatedly inputs a value and outputs the corresponding function value 'simultaneously'. No information about the preceding behaviour of such a system is needed for it to continue

[4] For example, instead of the phrase 'total information' in the second definition, something like 'total information about the behaviour of the system' may be used; and the phrase 'that can influence potential behaviour' may be appended to this definition.

operating[5], and so no stored state is necessary. Yet the potential behaviour of this system is well defined. Recall that we define 'state' as essentially 'that which determines potential behaviour'. In this case, potential behaviour is determined by the definition of the abstract 'pure function system' itself, a crucial part of which is the function definition.

For all systems, stored state is data, which must be acted upon by the system. So, stored state alone cannot determine potential behaviour; it can only determine potential behaviour when considered together with the system definition. So, one could say that stored state represents the dynamic part of the system state.

In the light of these considerations, we propose the following definitions of "abstract state" and "stored state". Each may be called, simply, 'state' when the variant is clear from the context:

(Abstract) State of a System: At a given instant, a notional attribute of the system that is sufficient to determine its potential behaviour.

An alternative way of defining abstract state is as 'a representative of the equivalence class of histories', where two histories of system activity are considered equivalent precisely when the potential behaviour of the system after one history is indistinguishable from that after the other history. In such a definition, the 'history' of a system must capture all its activity, both internal and external – behaviour at its interfaces is not enough. These notions of abstract state are essentially equivalent.

(Stored) State of a System: At a given instant, the total information explicitly stored by the system (in state variables) up to the given instant.

In concrete implementation terms, the stored state of a system is the set of values assigned to the internal *state variables*. Although not implied by the above definition, we will usually assume that the stored state of a system is relevant to its future behaviour. One could define 'relevant stored state' and 'irrelevant stored state' but, though the distinction is real[6], these terms are not necessary for our purposes.

---

[5] though some input/output pairs could be recorded to avoid some recalculation of the function

[6] it may happen, presumably in error, that a particular state variable is never relevant to the potential behaviour of the system. Even in the absence of design/implementation errors, the value of a state variable might not be relevant in some circumstances. (It often happens during execution of a program that a state variable is never read without first being written.)

For abstract state and (relevant) stored state to be well defined, the notion of behaviour must be well defined. This is because these concepts are defined with respect to a particular notion of behaviour. For the same reason, an abstraction of behaviour induces an abstraction of abstract state and of stored state. Note that a particular state variable may be irrelevant to *a given abstract notion of* potential behaviour, so state variable relevance is important when applying abstractions.

Stored state can be structured – state variables can be grouped – according to criticality of service: some state variables may be more critical than others because they might be relevant to more critical behaviour.

In many legacy systems it can be difficult to determine the complete abstract state of a system; the (dynamic) stored state and the (static) system definition may both be difficult to ascertain.

A system consists of a set of interacting subsystems. Therefore the system state space is the Cartesian product of its subsystem state spaces. (Note that the same is not generally true for *reachable* state spaces, because subsystem states are typically correlated by their interactions.)

Table 2 summarises and relates some useful concepts of 'state' (those not yet defined are defined below).

	Abstract	Stored	Declared
System	Abstract State A notional attribute that determines potential system behaviour	Stored State Stored data that is relevant to future system behaviour	Declared State The value of a declared data structure containing state variables relevant to essential future system behaviour
Interface	Abstract Interface State A notional attribute that determines potential behaviour at this interface	Stored Interface State Stored data that is relevant to future behaviour at this interface	Declared Interface State The value of a declared data structure containing state variables relevant to essential future behaviour at this interface

Table 2 — Some concepts of state for systems and interfaces

Some explanation of Table 2 is appropriate. Observe that the concepts in the first row relate to system behaviour, and those in the second row relate to system behaviour at a particular interface. The columns describe abstract, stored and declared concepts of state. The concepts 'declared state' and 'declared interface state' are defined with reference to

essential behaviour, by which we mean some part of the full behaviour considered essential for the application. Clearly, one could parameterise these concepts in terms of essential behaviour. We assume here that essential behaviour is understood, but remark that different ideas about what constitutes essential behaviour can cause problems when constructing systems of systems.

Notice that all the concepts in Table 2 vary according to the underlying notion of behaviour. Even with a fixed underlying notion of behaviour, these concepts are related in interesting ways. For example, the 'Abstract' concepts give the same information about the potential behaviour as do the 'Stored' concepts when the latter are taken together with a definition of the system. Further, in each row the 'Declared' variant of state is a subset of the 'Stored' variant. The 'Interface' concepts specialise the 'System' concepts by abstracting the behaviour to behaviour at the interface.

Declared State: At a given instant, the value assigned to a declared data structure that can be accessed via an interface and that records all the stored state that is relevant to (i.e., that can influence) the future essential behaviour of the system.

So 'declared state' is simply a structured representation, made available at a system interface, of the stored state that is relevant to essential behaviour. In principle, there may be any number of suitable declared data structures, but there will typically be one, or none, known to an engineer who wishes to include the system within a system of systems. Observe that a declared state is simply a representation, made available by the system at an interface, of the (relevant) stored state, for a given notion of behaviour. Since a 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 stored state of another system. As with state, and stored state, abstractions of behaviour induce corresponding abstractions of declared state.

(Abstract) Interface State: The (abstract) state of a component system as viewed from a particular interface. It is a notional attribute of the interface that is sufficient to explain future behaviour of the component system across this interface.

The abstract interface state for a given interface is determined by the abstract state of the component (when both are defined in terms of the same underlying notion of behaviour).

(Stored) Interface State: The (stored) state of a component system that is relevant to future behaviour at a particular interface. Together with a definition of the system, it is sufficient to explain the behaviour of the component system across this interface.

Stored interface state consists of data that might, or might not, be made explicitly available to an interfacing system. The stored interface state for a given interface will typically be a subset of the full stored state of the component.

Example: One would expect the (abstract or stored) interface state of a particular Amazon.com session to be independent of the identities of the users running simultaneous sessions, though their identities etc. are part of the server state.

We will argue that a linking interface ideally makes the interface state available, and that this interface state should be as simple as possible. We will also argue that the DSoS sparse model of time simplifies the interface state.

Declared Interface State: At a given instant, the value assigned to a declared data structure that can be accessed via an interface and that records all the stored state that is relevant to (i.e., that can influence) the future essential behaviour of the system at the given interface.

The declared interface state is the declared state that is part of the interface model of the considered interface.

Interface Model: the model of the concepts a user has in mind when he/she relates the meaning of the chunks of information in a message (which are the results of the syntactic specification) to his/her conceptual world [Kopetz 2002c].

3.5 Dependability

Our concern is with system dependability, the definition of which term, in similar fashion to a number of related terms, we base on that of [Laprie 1992]. In addition, the definitions of fault, error and failure below owe much to [Jones 2002], especially where ideal terminology is described. (There is a choice between defining real-world concepts and defining idealised counterparts; the latter are more readily formalised but less generally applicable. We have chosen to formulate real-world definitions, and indicate idealisations in the text.)

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.

Ideally, dependability is defined with respect to a *specified* behaviour and a *specified* dependability metric, but these may be informal (and in practice often will be). It is helpful to make these specifications explicit, because this helps prevent differences in this essential information being ignored during development of a system of systems.

In principle, different stakeholders, such as the system owners and various system users, will have different views regarding the intended behaviour of a system – they will desire conformance to different specifications of behaviour, where available. Consequently, the stakeholders will have different views of its dependability.

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

Ideally, failure of a system is defined with respect to an explicit specification of (intended) system behaviour. This explicit reference to a specification – which may be formal or informal – is intended to ensure that the appropriate relationship exists between validated properties and required properties of component systems. Absence of component system failure, from the perspective of a SoS designer, must follow from absence of failure from the perspective of the component system implementer. That is: the (validated) properties of individual component systems should imply the respective requirements placed on them by the SoS designer.

Many systems are designed such that failures can only occur at outputs. Other systems may be able to refuse to accept an input from an interfacing system that is ready to send it an output, perhaps by failing to send an 'input buffer not full' message, or failing to send an acknowledgment.

Service Failure: A failure at a service interface of the system.

Ideally, a precise specification (both in the value domain and in the temporal domain) of the intended behaviour for a particular stakeholder is available for the judgement about whether a system has failed for this stakeholder. 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 about whether a system failure has occurred.

A useful distinction can be made between 'failure to produce correct output when provided with correct input', and 'failure to produce correct output because provided with incorrect input'. In the former case, the given system has been affected by an internal fault, whereas in the latter case it has failed due to error propagation from the system that provided the erroneous input.

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. Errors can exist in any part of a system's state, i.e., in its (static) definition or in its (dynamic) stored state.

Fault: A fault is the cause of an error.

A judgement is needed to diagnose a fault. This judgement will depend on precisely what is considered as the system, and at what level of detail the system is considered.

A fault is *active* when it produces an error, otherwise it is *dormant*. On activation, a fault causes an error within the stored 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.

Error Containment Region: A well-defined subsystem of a computer system that contains error-detection mechanisms 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.

In the above definition, *well-defined* means accessible only through well-defined (linking) interfaces.

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.

If the failure of a fault containment region is unconstrained, an error containment region must comprise at least two fault containment regions in order to be able to detect the consequences of a single fault [Kopetz 2002a].

Although (components of) different fault containment regions are considered to fail independently, their respective failure rates may differ.

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.

3.6 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 of systems out of component systems, it is necessary to ensure that architectural styles match at any interfaces between which direct interactions occur. 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.

An important interface property is whether it is *elementary* or *composite* [Kopetz 1999]:

Elementary Interface: An interface across which only elementary interactions can occur. An *elementary interaction* is one where all messages are transmitted according to the information push model (i.e., the consumer of each message exerts no control – no back pressure – on its transmission).

Composite Interface: An interface across which composite interactions can occur. A *composite interaction* is one where at least one message is transmitted according to the information pull model (i.e., the consumer of some message exerts control – back pressure – on its transmission).

Elementary interfaces are inherently simpler than composite interfaces, so elementary interfaces are preferable for the linking interfaces (see below) of dependable systems of systems.

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 dimensions of the plug, the voltage and the 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 modelled 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 the expansion of a boundary line into a connection system 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.

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.

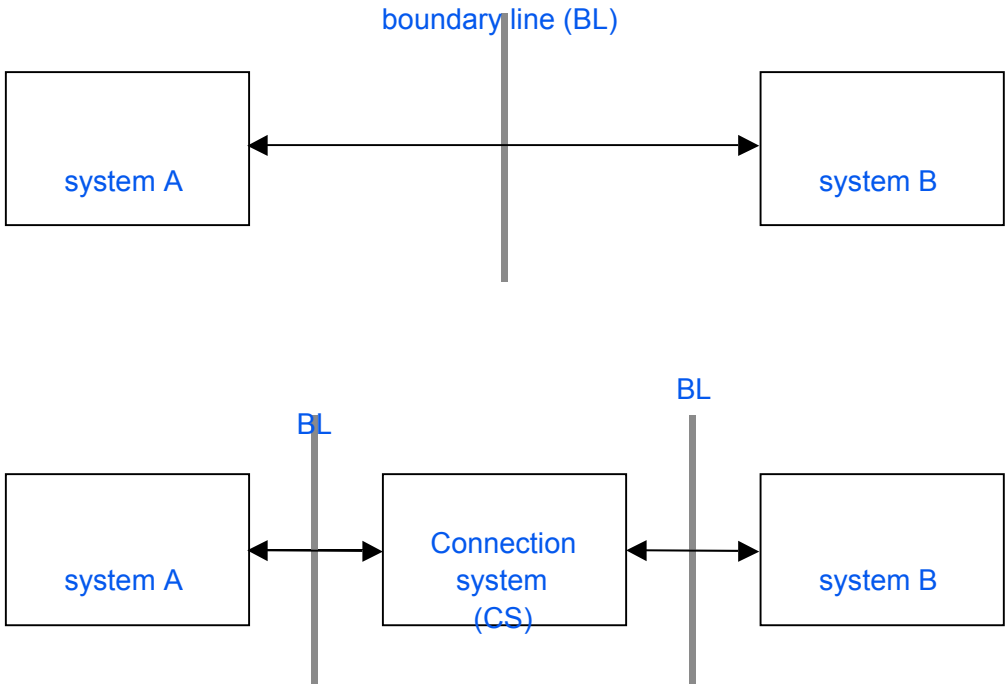

Figure 1 — Expansion of a Boundary Line (BL) into a Connection System (CS) with two Boundary Lines

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

Well chosen linking interface specifications together include all information about the interface behaviour of all the component systems in a system of systems. In this way, they insulate the development of, and reasoning about, the SoS from the implementation of any component systems that are not already available.

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, systems can be replicated 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 resolve property mismatches and thus 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: A set of rules that specifies the interactions between two or more component systems between connected interfaces.

The notion of a protocol is more limited 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 focuses on only the interfaces connected by the protocol.

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 [Kopetz and Bauer 2003]**.**

3.7 Other Notions of Behaviour

System behaviour can only be defined with respect to a particular computational model. Even with the notions of send and receive operations here, the appropriate concept of behaviour varies according to the context of the discussion.

It often happens that the only aspect of system behaviour that matters, in the context where the system is used, is the set of possible (timed or untimed) traces of (input and output) operations that the system may perform. Trace behaviour is typically the simplest behaviour of interest to system stakeholders. It allows so-called safety properties to be expressed.

Sometimes, however, it is important to know when the system can 'refuse' to perform some operation (in a well-defined sense). This allows one to distinguish between a system that can perform some important trace (but may refuse to do so), and one that must perform that trace (it cannot refuse to do so). For example, it matters whether a clock can refuse to send ticks, or refuse to send the current time value, to an interfacing system. In the extreme, systems can deadlock.

Another aspect of behaviour that sometimes matters is the occurrence of internal events, particularly when internal 'livelock' is possible. If one wishes to distinguish between possible livelock and possible deadlock of a system, then the notion of behaviour must be enriched to take account of internal events.

At a low enough level of abstraction, all physical systems are deterministic[7]. However, if details are hidden then the resulting abstract system may behave nondeterministically. In

[7] We ignore quantum mechanical effects.

order to allow (abstract) specifications of system behaviour, it is often useful to be able to represent nondeterministic behaviour. Contrary to what one might at first think, non-determinism is not incompatible with the notion that state 'determines' potential behaviour. With a suitable definition of behaviour, potential behaviour captures the non-determinism.

3.8 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 will 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:

.

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* , 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 synchronised 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 synchronisation 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 synchronising 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 digitisation of time cause an unavoidable error in time measurement in 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 of 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 33).

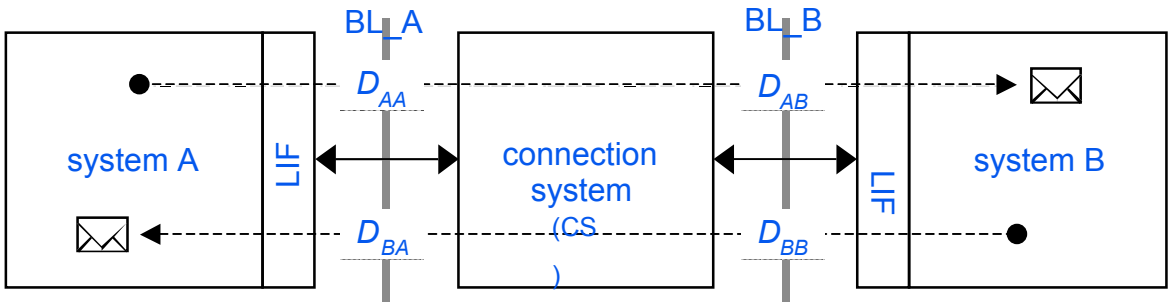

Figure 2 — Request-response interaction through a connection system

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 component 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.

4.1 Interface Types

In order to disentangle unrelated functions it is advantageous to specify a distinct interface for every separable service [Kopetz 2000b]. 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 order to realize the emergent services of a system of systems, a set of component systems is linked together by connecting them via their service interfaces. The service interfaces thus become the linking interfaces (LIFs) of the components.

The specification of a service interface must define, either directly or indirectly, all properties introduced in Section 2.2.5, from low level to high level. Many of the low level properties of this service can be defined indirectly by referring to a designated architectural style (see Section 1.3.2). The high-level properties of a service interface can be specified by presenting a *service interface model*. In general, such a service interface model will contain parts that are amenable to formal specification (e.g., the syntax of the input and output messages) and parts that can only be expressed by reference to natural language concepts (e.g., the meaning of the data in the input and output messages). The service interface model can also include a *declared interface state* (see Section 3.4) that

records the cumulative effect of the history of the component system, including events at all its interfaces, on the future behaviour at this service interface.

Ideally, the specification of the service interface should be self-contained such that the user of the component services finds *all* information needed to use the services of the component system in the service interface specification. All issues related to the composition of a set of component systems can then be investigated by referring only to the specification of the LIFs of the component systems without any knowledge of their internal structure and operation.

A single component system may support many LIFs that are, from the point of view of each LIF user, separate, although they might not be independent. In a SoS the interface state of a LIF may change autonomously (by the progression of time or, more generally, by the occurrence of some internal event) or at least independently of the interfacing system (because of activities at other separate LIFs).

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 of systems, the purpose of the *real-time service* (RS) interfaces of component systems is the timely exchange of observations among the component systems. An observation states that the state variable possessed the stated value at the indicated instant or an event occurred 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.

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.

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 39) constitute the high-level interface issues (HLII). In particular, the following topics are among 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 sub-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 have investigated 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 on naming of moving an object from one system to another system.

c) Naming and consistency of replicated objects.

In principle, there are two possible approaches to 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 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 component 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 [Jones et al. 2002].

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 a (timestamped) observation (see Section 3.2) 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] to gain 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 component systems, 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]) 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 of 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 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, (human) users interact with clients, which 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.

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 service or a 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 or 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 (RMI)

The main difference between an RPC and a remote object method invocation lies in the late binding of the code to the 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 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 2000b] 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 2000c] 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 et al. 1988] where it is called multipoint association. Multipeer interaction is 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 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 System (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 [Wayner 1997].

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 is 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 data structure that resides in the communication memory of each node [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.

4.2.3 Dependability attributes of interactions

A system may rely on 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 non-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 et al. 1995; Xu 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 et al. 1998].

4.2.4 State persistence

We define a 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 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.

A ground state is a declared state of a component system assuming all tasks are inactive and no messages are in transit. Thus, strictly, a system can have a number of ground states.

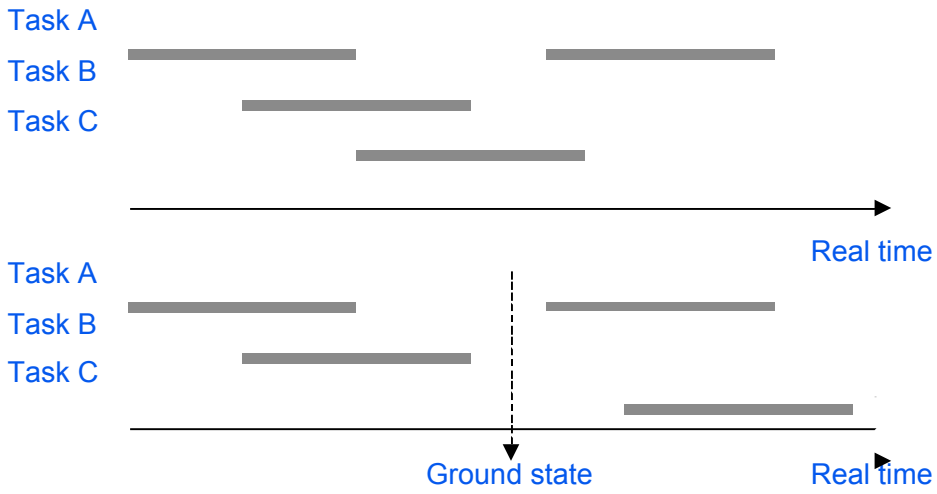

Figure 3 — Task Executions: without (above), and with (below) ground state

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 2000a] 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 2000d], 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 among 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, for example, be a hardware communication controller such as that used in the time-triggered protocol (TTP) [Kopetz et al. 1999], a CORBA object request broker (ORB), or an HTTP server.

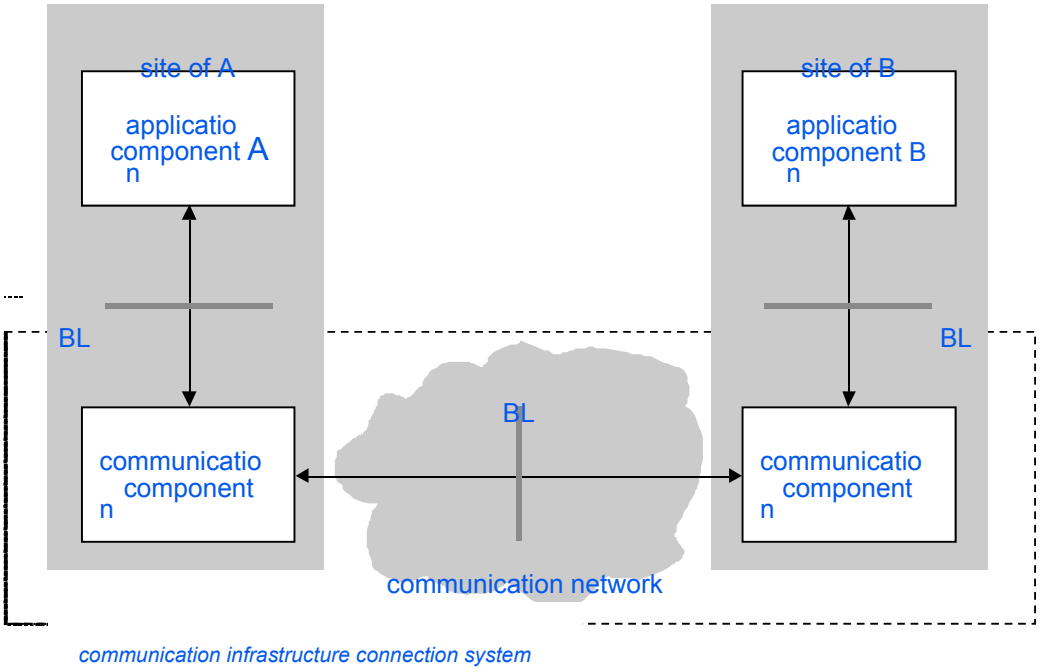

Figure 4 — Transport model between two application components on different sites

Comparing Figure 4 with Figure 1, page 33, 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).

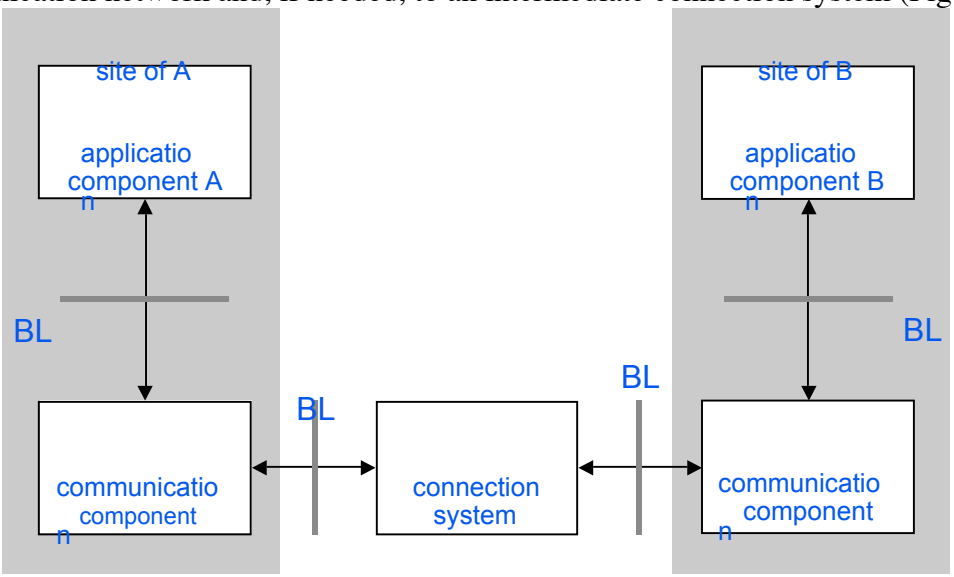

Figure 5 — Transport via an intermediate connection system

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 event *e1* in Figure 6 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.

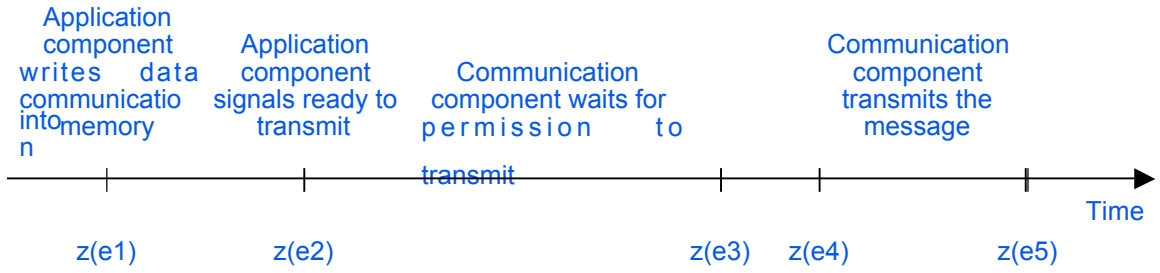

Figure 6 — Timing of a Message Send Operation

Figure 7 shows the timing of the receive operation. 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.

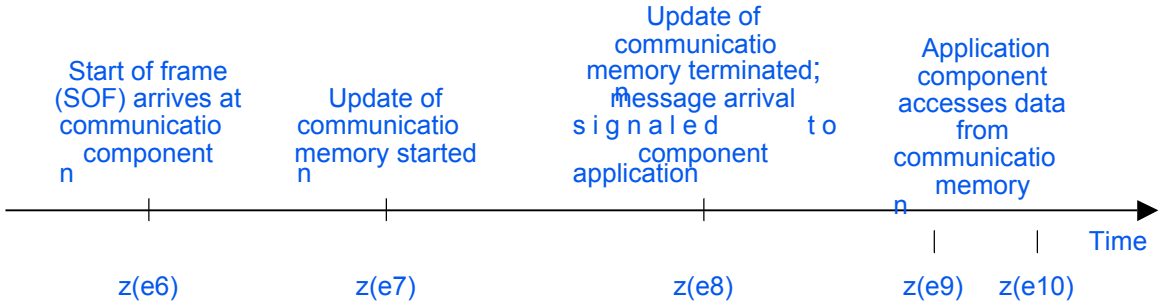

Figure 7 — Timing of a Message Receive Operation

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 an explicit acknowledgment message to the sender, after the successful arrival of a message, 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 (retransmission) 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 by, among other attributes, 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 54), 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 assures 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 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 sending a message only at the agreed instants, and the recipient commits itself to accepting all messages sent by the sender, as long as the sender fulfils 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 has failed 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 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 3) 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 3 — 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:

Enqueue/dequeue: If the transmitted information contains non-timestamped event observations, 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 eventually be 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 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-triggered Event message

An event-triggered event message (or for short, event message) combines a unidirectional data flow with a bi-directional control flow. Unless event messages are timestamped, they are not idempotent, so exactly-one semantics is required. A typical means of implementing these semantics is to use a message queue, with bi-directional control flow between the sender and recipient to ensure that the queue data structure does not overflow. 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). Event messages are used, for instance, in

client-server protocols.

4.3.3.2 Periodic state message

A periodic state message sequence 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 state 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. Accessing a state message at the interface is similar to the accessing a variable in memory (we assume that read and write operations are atomic).

The following table (Table 4) compares the characteristics of these two transport mechanisms.

Event messages are sent sporadically, triggered by the irregular occurrence of events.

	Event Message	**Periodic 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 4 — Characteristics of the transport mechanisms:
Event Message and Periodic 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. Indeed, this is consistent with the observation that asynchronously exchanged messages increase system modularity [Pnueli 1986]. State messages, like non-blocking invocations, avoid control propagation through component interfaces, thus improving error-confinement and decreasing interdependencies between a system's components.

4.3.4 Integration of event-triggered 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 5. 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 [Führer et al. 2000]. 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 [Kopetz and Bauer 2003] 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 [Kopetz and Bauer 2003], 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 are 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 5 — Alternatives for the Integration of ET and TT Services

For embedded real-time control systems, DSoS has selected the middle alternative of Table 5, 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 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: input and output interfaces for state messages (update-in-place on input, no consumption on output) and input and output interfaces 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.

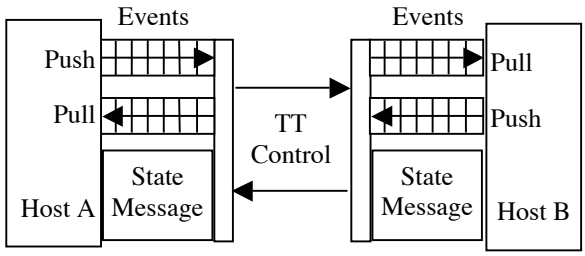

Figure 8 — Model TT-ET Interface

In most real-time architectures, the basic communication service is a broadcast service (e.g., in CAN and TTP) that connects the *n* nodes of a cluster. Every transmitted event message thus generates *(n-1)* event messages at (n-1) 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 messages from the receiver's queue based on the age of the message. A maximum *queue-storage duration* is assigned to each timestamped 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 [Obermaisser 2002].

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 are best 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.

4.4 Ideal characteristics of LIFs

Each of the following LIF characteristics facilitates development and/or verification of dependable SoSs. These characteristics are not obligatory, and indeed some – which are indicated – are targeted at safety-critical real-time systems. Section 5.3 provides examples of how some of these characteristics benefit SoS development and verification.

4.4.1 What does 'ideal' mean?

The ultimate objective is the development and verification of dependable systems of systems. LIFs are central to these activities, for two reasons:

1) LIFs are critical for the emergent services of a system of systems, and to their dependability, since these are realised by the interaction of component systems at their LIFs;

2) LIF specifications insulate SoS-level considerations from component-system-level considerations.

Good LIF specifications allow SoS design and verification to proceed under the assumption that suitably verified component systems will be provided (either developed afresh, or pre-existing and perhaps then wrapped); and they allow engineers to provide component systems, and verify them against their LIF specifications, without depending on any particular SoS design. Ideal LIF properties facilitate this decoupling of SoS and component system considerations.

The 'ideal' LIF characteristics are presented according to how they support composability of component systems into systems of systems.

Four 'Principles of Composability' are identified in [Kopetz 2002c] as ideal characteristics of composable architectures. In the terminology of the DSoS Conceptual Model, these principles are (1) Independent development of component systems, (2) Stability of prior services, (3) Performance of the connection systems, and (4) Replica determinism.

The temporal domain of SoSs is similar for different application domains. This similarity allows generic rules, such as the above principles of composability, to be formulated. In turn, these rules help explain some ideal LIF characteristics, identified below. However, we have not identified generic rules for the value domain – composability in this domain depends very much on the application. The ideal LIF characteristics described below focus mainly on temporal composability.

Even so, one "golden rule" can be formulated for the design of ideal LIFs (irrespective of the domain): to understand the operation of a component system at a LIF, it should only be necessary to understand its interfacing component systems and not the whole SoS. This reduces the complexity of cognitive perception, because to understand a SoS it is then only necessary to understand the individual interfaces and components. By this rule, a pre-defined communication schedule is preferable to a priority scheme.

Many of the characteristics described below can be justified on the grounds that they lead to simple interfaces. This is clearly helpful for human understanding, and so for design. It also simplifies formal verification, as explained in Section 5.3 below.

4.4.2 Independent development of components

For independent development of the component systems of a SoS to be possible, based on some set of component system specifications, these specifications must include all their behavioural information that is relevant to the SoS design. As well as allowing component systems to be developed independently of each other, such specifications allow verification of the design to proceed independently of the development of the component systems.

Of course, development can proceed top-down, bottom-up or as some mixture of these paradigms. Construction of suitable component system specifications is the responsibility of the SoS designer if development is top-down. In bottom-up development the component system specifications may be available before SoS design begins. SoSs are likely to be made from existing component systems, though the SoS designer will be free to design and implement new connection systems.

LIF specifications should together fully specify component system service insofar as is relevant to correct operation of the SoS – they will then insulate SoS design from component system development (where component systems are not already available), since *all issues related to the composition of a set of component systems can then be investigated with reference to only the LIF specifications of the component systems without any knowledge of their internal structure and operation* (Section 4.1).

Of course, LIF specifications should be clear and unambiguous.

Time is fundamental to the DSoS Conceptual Model. By definition LIFs must be specified in both the value domain and the temporal domain.

It is helpful if component systems (perhaps wrapped) provide distinct interfaces for separable services; this helps disentangle unrelated functions and so simplifies SoS design and verification. For this reason it may be helpful if separate service specifications are available for each stakeholder.

Explicitly available interface state allows interfacing systems to have easy access to the state of a component system, insofar as is relevant to its behaviour at the interface. In effect, the interface state is a ready-made relevant abstraction of the component system; it hides irrelevant details and thus simplifies dependable SoS design, since interfacing systems are saved the effort (and complexity) of deriving the interface state. It is strongly advisable to make at least some parts of the interface state explicitly available, if only to help interfacing systems recognise failures of component systems, and take appropriate action.

LIFs may contain redundancy in the presented interface state, for efficiency reasons and/or dependability reasons. It may be desirable to group state variables of the interface state, to allow fast self-restart or efficient re-initialisation of failed peer component systems.

A sparse time model simplifies interface state because time can be abstracted to a sequence of global times. This gives rise to an observation/action lattice [Kopetz and Bauer 2003] according to which all component systems act.

4.4.3 Stability of prior services

Importantly, LIFs should be incapable of propagating control errors – it should not be possible for flow control to be exerted by one system on another through LIFs. This prevents component system errors contaminating an interfacing component except by providing incorrect data. Restricting component system errors to be data errors – that is, not control errors – often simplifies error detection and error handling [Kopetz and Bauer 2003].

4.4.4 Performance of the communication system

Time-triggered, resource-adequate communication, with small transport delay and minimal jitter, allows temporally accurate RT images to be maintained, which are necessary for safety-critical real-time SoSs.

Stable delay and low jitter are required for composability (Section 4.1).

5 FORMALIZATION

This section discusses the formal specification and verification of dependable systems of systems. Emphasis is placed on the linking interfaces of component systems within a SoS – the LIFs of the component systems are central to the specification of each component, and of course it is via the LIFs that the connections are realised that allow the services of the SoS to emerge.

Section 5.1 classifies systems of systems as either non-time-critical or time-critical. This distinction is useful in the subsequent sections. Section 5.2 discusses formal specification and verification for SoSs in which no particular characteristics of the LIFs are assumed. This is followed by a discussion, in Section 5.3, of how the various 'ideal' characteristics of LIFs (introduced above in Section 4.4) benefit the formal specification and verification of SoSs.

Section 5.4 discusses formal specification of LIFs, and also of complete SoSs. A popular approach to specifying interfaces is the use of an interface definition language (IDL); CORBA's IDL is summarised and its suitability for defining LIFs is assessed. The UML-based architecture description language (ADL) that has been developed within the DSoS Project is also briefly described.

Other DSoS formalization activities are summarized in Section 5.5.

5.1 The Universe of Applications

The scope of what is viewed as a system in DSoS is very wide (see Introduction, Section 1.2). This in turn means that no single formal notation (nor technique) will cover all Systems of Systems. This section briefly sets out the distinguishing characteristics of non-time-critical systems and time-critical systems.

5.1.1 Non-time critical

In some senses, it is confusing to refer to any system as being non-time critical: if a square-root routine took a hundred hours to compute, it would be unacceptable. So the overall performance of even a simple operation (or function) is of concern and might be part of its specification. (Performance issues of this sort are often referred to as non-functional or meta-functional requirements – this terminology is unfortunate in that performance *is* connected in most users' minds with function and is clearly specifiable and measurable.) Beyond such simple (non-interfering) operations of systems, there are concurrent systems which are normally characterised as being non-time critical. As indicated below, characterising the interference of such systems can be a difficult issue but, unless the relative rates of progress have a part to play in the specification, a

specification can be given which does not require explicit reference to the progress of time. As indicated below, real-time questions normally become central when the progress of time in the environment of a system affects the behaviour required of the system. If the temperature in a nuclear reactor is increasing, it matters how often that temperature is sampled and when the reaction is damped.

Essentially then, the characterization of a system as "non real time" means that it can be specified by simpler approaches that are less-constraining in terms of time awareness, which allow more design freedom. However, time related properties cannot be guaranteed for the resulting system, so one should not pretend that a system is not time critical if it actually is. On the other hand, one should strive to identify sub-systems where time can be discussed as simply as possible

5.1.2 Time critical

It is most common for systems to become time-critical when they interact with the physical world: a train continues its forward progress (possibly into an occupied section of track) unless the brakes are applied. A system controlling the train must issue orders in a timely fashion. In order to do so, the system will also need to obtain information at sufficient frequency (there is no value in out of date speed or signal status information).

As explained below (Section 5.2.7), the formal specification and verification of time-critical systems is greatly eased if the system architecture supports a sparse time base (see Annex 1). A sparse time base allows significant simplification of the formalization process, as it allows time to be represented by integer values. These integer time values can be consistently ordered and used by an application. If the architecture includes a mechanism that provides fault-tolerant clock synchronization, these integer values have a guaranteed relationship to real time, and this relationship may be exploited for the purposes of verification.

5.2 Current Approaches to Formalization

There is an extensive literature of formal specification techniques of various kinds. Much of this literature pre-dates the identification of LIFs as a key concept for specifying real-time systems and, suitably generalised, for specifying systems of systems in general. This work is reviewed here and its suitability is considered for SoSs in which no special characteristics of the LIFs are assumed.

5.2.1 The role of specifications

In nearly all engineering disciplines, there is a need to understand key properties of an artefact without knowing all of its internal details. A purchaser of a car might be interested

in the energy from the engine or in the time to accelerate to various speeds; only those concerned with design and maintenance will be concerned with the precise dimensions of the pistons etc.

It is, then, commonplace to say that a specification concerns *what* a system should do rather than *how* this is achieved. Such a specification can serve as an insulation between those using a system and those building it. If we are ever to achieve a world of re-usable systems, these systems must be accompanied by specifications. It is also crucial that such specifications define the *semantics* (as well as the *syntax*) of the operations of a system. How this is to be achieved is of course a matter of debate – one which is tackled below.

The *what* and the *how* above can be considered as different levels of abstraction. Abstractions, and their importance to verification, are discussed in more detail in Section 5.2.3.

So far, the case has been made for specifications of extant systems so that one can determine whether they fit (possibly after wrapping) a required purpose in constructing a system of systems.

One particular role for specifications is in the creation of a new system (or the evolution of an old one). A specification should be used to provide a reference point for development; otherwise the developer has no way of knowing what is required. Notice that the qualification of "formal" (specification) has been dropped here: the observation is simply that there must be some way of specifying what function is required of a system. (It can be argued that formal specifications have advantages in contracts, but a different argument is used for formalization below). It is worth heading off two potential objections here. First, it is easier to talk about the order of specification creation and implementation as in a "waterfall" diagram although, in nearly all real systems, there will be iteration back to the specification (and thus to the customer) during the design process. This simplification makes description easier; it is not a limitation to the approach. Second, enthusiasts of "open source" development or "extreme" programming might maintain that their systems are developed without specifications. While there might be some truth in this claim where the developers of a system are also a subset of the intended user community, it is certainly not an appropriate model for many sorts of system. In order to reflect the position of such developers, the arguments for a specification as an insulator between the developers and an eventual (larger) user community can substantiate the case for understandable descriptions that avoid the need to understand the internals of a system.

It is widely accepted that the productivity of developers of large systems is not high. One clear reason for this is the imprecision of informal specifications. Furthermore, much of the lost productivity comes from undetected mistakes in the early stages of development, which in turn leads to the scrapping of much work based on flawed early design decisions.

Inspections of informal high-level designs all too often concern themselves with trivia such as grammatical and spelling errors because no formal material exists until code is available (code inspections –by contrast– are far more effective). One advantage of formal specifications is that they can support both precise arguments and objections early in the development process.

Another use of (formal) specifications is the generation of test cases.

The remainder of this section discusses some of the main approaches to formal specification.

5.2.2 Operations and state machines

So far, we have described systems in terms of the operations they provide (here called the *operation style* of description). A common alternative style is to describe systems in the *state machine (automaton) style* of description. Depending on the system, the automaton used may be finite or infinite state, and it may be deterministic or nondeterministic. A state machine description of a system consists of a set of system states, with transitions between them (which may be labelled). Such an automaton describes a system that moves between states along transitions. Transitions may occur in response to inputs (external stimuli), or when providing outputs, or by internal activity. The particular action (or event) for a transition may be represented by its label.

A system description in terms of its operations can be cast, instead, in terms of an underlying state machine: an operation can be thought of as a set of transitions from states satisfying the pre-condition to states that are related to these by the post-condition. The state machine is deterministic only when all post-condition relations are functional. Conversely, any state machine description can be cast in terms of operations: in the extreme, one might have one operation per transition (though there is a free choice of operation names, pre-conditions and post-conditions as long as the two descriptions are equivalent).

This duality between operation style and state machine style descriptions allows us to move between the two styles as we wish; some of the issues discussed below are most easily made in terms of one style or the other, but it should be clear that the same points, suitably translated, apply to both styles.

5.2.3 Abstractions

The idea that any system can be considered at different levels of abstraction is well known in all scientific and engineering disciplines.

A specification can be pitched at any one of many levels of abstraction. A user only interested in the basic features of a system does not care whether it has an 'expert' mode, but of course an expert user does. There is often a hierarchy of abstraction levels, at each of which the system can be described in a suitable way for particular stakeholders. At successively lower levels of abstraction, the stakeholders might, for example, include basic, intermediate and advanced users, system designers and then software and hardware engineers. The particular levels of abstraction that are appropriate will obviously depend on the system, its end-users, and its manner of implementation. Since simple descriptions are desirable, any system description should ideally omit as much irrelevant detail as possible: it should 'abstract away' from the lower level details without leaving out any information that is relevant to the stakeholders at this level of abstraction.

Verification very frequently exploits a hierarchy of abstractions: by arguing that each system description *refines* (in some sense) that at the level of abstraction above it, one can argue that the least abstract description of a system refines the most abstract one. A suitably transitive notion of refinement is needed, which (to be useful) must correspond to one system description satisfying the more abstract system description (in a well-defined sense).

Each description is at a particular level of abstraction. The intention with a hierarchy of abstractions is that each description can be viewed as a specification of the systems described at lower layers, and as an implementation of the systems described at higher layers. Thus 'specification' and 'implementation' are descriptions that have distinct roles.

When discussing hierarchies of abstraction layers, it is often convenient to focus on a so-called *concrete layer* and a (higher) *abstract layer*. Examples are the real world system and a model of it, respectively. Often, the layers themselves are not located in a particular hierarchy, in which case the discussion refers to any pair of layers, provided only that the concrete layer is indeed below the abstract one in some hierarchy.

The suitability of any abstraction from a concrete system to an abstract model is obviously determined by the uses to which the model is put. To be useful for verification, an abstraction must preserve enough information for properties of the model to be validly mapped to properties of the real world system.

A general approach to verifying properties of a concrete system is as follows:

1. map the concrete system (up) to an abstract model;
2. reason about the abstract model to discover some of its properties;
3. map these abstract properties (down) to properties of the concrete system.

An example of this approach is the seminal work of [Cousot and Cousot 1977] on *abstract interpretation*. They introduced *abstraction functions* to map from any concrete description or property up to an abstract one, and *concretization functions* to map from any abstract description or property down to a set of concrete ones. Essentially, abstract interpretation calculates properties of a concrete program by using abstractions and concretization functions for steps 1 and 3 above and interpreting the abstract program in step 2.

5.2.4 Model-based techniques

The centrality of the notion of a state in most systems has been discussed in the Introduction (Section 1) and in Section 3. A system interface then consists of a number of operations which, optionally[8]

1. take arguments;
2. depend on the state of the system when the operation begins;
3. transform the state by termination of the operation; and
4. deliver results.

In the case that there is no interference (there is only one operation executing at any one time and the state of this system is insulated from interference from any other system), it is a straightforward task to specify each operation. The terminology of VDM [Jones 1990] is used here but the same points could be made with other formalisms such as B [Abrial 1996] (or with some reservations Z [Hayes 1993]).

A pre-condition is used to record the applicability of an operation. A pre-condition is a predicate of a state (the initial one) and any inputs. A post-condition describes the valid outcomes of an operation. A post-condition is a predicate of two states (the initial and final ones) and all inputs and results. So an operation can be partial and non-deterministic. Each of these points is worthy of consideration.

One can argue that operations should be made as robust as possible and this would suggest that pre-conditions should evaluate to **true** for any inputs and thus require that the implementer arranges that the code always terminates. In fact, it is very difficult to write systems that are this general (even for the trivial example of a **Stack** one can ask what result should be returned when **Pop**ping from the empty stack). In general, a specification method must permit partial operations to be specified (in fact, VDM allows error clauses

[8] Everything that is written here about specifying operations specialises to pure functions simply by dropping the States.

to be specified that enlarge the domain of an operation without clouding the normal post-condition with strings of implications).

The implementation of an operation is required to terminate for all situations where its pre-condition evaluates to **true,** and to yield results (when compared to the starting situation) for which the post-condition evaluates to **true**. It is thus easy for a specification to be written that admits more than one result or final state. This is actually desirable even when the final implementation is to be a deterministic program. Permitting a range of results allows design decisions to be postponed. For example, one might say of a storage management system that it should **alloc** a free piece of store (of the requisite size) and postpone the design of free chains etc. (There is a more subtle point about non-determinacy resulting from levels of data abstraction in [Hayes et al. 1993].)

Post-conditions are written in the language of first-order predicate calculus (or the LPF variant – see [Barringer et al. 1984]). This makes it possible to document requirements in a compact and economical way: conjunction and negation are operators which are not (in full generality) available in programming languages (the simplest example that illustrates why this makes it possible to state perspicuously what is required is sorting, where one can say that an ascending sequence is required as a result and that this must be a permutation of the input).

Useful though the pre/post-condition idea has proved, the technique of specifying states of systems in terms of abstract objects is even more important. Much of what makes programming difficult is choosing objects which fit the linear address space of a computer. Specifying systems in terms of

- sets
- maps
- sequences; and
- composite objects (cf. records or structures)

makes specifications far shorter and more tractable than the eventual implementations. Another idea which proves very useful in practice is that of “data type invariants”. Further details of these ideas can be found in [Jones 1990] along with formal methods (operation decomposition and data reification) for proving that designs and/or implementations satisfy specifications; it is time here to move on to the specification issues which are more challenging.

One contribution of the TTA work [Kopetz and Bauer 2003] is the identification of multiple interfaces as a way to structure understanding of a single system from different points of view. It is not yet completely clear whether all such interfaces should be

described at the same level of abstraction. It might –for example– be necessary for a diagnostic interface to discuss representation details which are not required in the service level interface. If this turns out to be the case, one could deploy known ideas on relating representations (see again [Jones 1990]).

Another interesting issue is the existence of systems where the availability of operations depends on the state of the system (to resume an earlier trivial example, the **Pop** operation could be made unavailable when the stack is empty). In practice, such systems can be rather fragile and can result in deadlock. But –contrary to widely held opinion– there is little difficulty in specifying them. Many object-oriented languages (e.g. POOL [America 1989]) include an Ada-like routine for each method which defines which calls can be accepted. This point is resumed below under the discussion of controlling interference.

5.2.5 Extensions to deal with concurrency

The essence of concurrency is interference. If one is to have a specification method to deal with concurrent systems, this fact must be taken on board. In particular, devising a compositional development method for concurrent systems requires that specifications characterise the interference that operations can accept from –and inflict on– their environment. The "Owicki/Gries" method [Owicki 1975] [Owicki and Gries 1976] is non-compositional because the method is to:

1. specify the separate components;
2. develop and prove code which satisfies the separate (pre/post) specifications;
3. prove that no step of one process can interfere with a proof step of the other process.

This can result in having to discard one or more developments if the "interference freedom" test fails.

Rely and guarantee-conditions were proposed in [Jones 1981] as a way of specifying interference. A rely-condition for an operation is a predicate of two states and limits the interference that a developer must tolerate; a guarantee-condition is a predicate of two states that specifies the limits of interference which the developed component can generate. (It is useful to compare pre with rely-conditions: both give the developer permission to ignore certain situations; and guarantee with post-conditions: both are requirements on the created system.) Of course, it is only safe to deploy a system in an environment where it can be shown that the rely (or pre) condition holds. To this end, there are proof rules (which can be compared to the standard rule for **while** introduction) which permit the introduction of a parallel statement in the design process. This rule is compositional in the sense that the specification tells the developer all they need to know

about the requirements on an operation. The product of their labours will not be rejected by some post-facto check.

The basic idea of rely and guarantee specifications has been greatly extended (including covering forms of progress) in the theses of Ketil Stølen, Xu Qiwen, Pierre Collette and Juergen Dingel (for example).

But writing such specifications is delicate: [Collette and Jones 2000] shows how one has to balance placing conditions in the various predicates (and the "dynamic invariant"). It became obvious that one should localise the use of interference and one attractive way of doing this is to recognise that object-based languages permit:

1. complete isolation of their instance variables;
2. the segregation of **private** references that point to "islands" of objects in which no interference is possible; and
3. general references that the designer can use to control interference.

This observation led to the "POBL" approach – see [Jones 1996] (which paper also has a useful sketch of the rely/guarantee idea).

5.2.6 Process Algebras

One way of looking at process algebraic approaches is to ask – if (shared) state is a problem – why not eliminate states? (This echoes the point that functional languages are more tractable than (sequential) procedural languages.) It is also true that process algebras provide a natural way of fixing the varying availability of services in a process (again, taking the stack example, it is easy to describe a stack which declines to accept a **pop** message when there is no data available). This of course introduces a possibility of deadlock like situations. While the traces of processes are very intuitive, the divergences and refusals are harder to reason about. But there is a rich literature of equivalences (bi-simulations).

As soon as one realises that one can simulate a shared variable with a CCS process, it is obvious that the scourge of interference has not been eliminated. Colin Stirling [Stirling 1988] has actually published a development of the rely/guarantee idea which applies to process algebras. It does not therefore appear likely that viewing systems (of systems) solely in terms of process algebras will solve all of the problems of describing SoSs.

However, process algebras can still be useful for effective modelling and verification of concurrent systems. In DSoS, we have taken advantage of the compositionality of the CSP process algebra in order to describe the development of systems based on coordinated atomic actions (see Section 5.5.1).

The proposed development method allows two types of refinement: the refinement of processes (where a process is replaced by a network of sub-processes), and the refinement of communication interfaces (where the communication mechanism used to link processes is replaced by another mechanism described using activities at a lower level of abstraction). To relate processes with different interfaces a new implementation relation is used in addition to the standard CSP refinement relation ***[BKP01]***.

The process algebra CSP has been used within the DSoS Project to model CORBA's protocol GIOP (General Inter-Orb Protocol). This work has demonstrated the utility of data-independence techniques for proving properties that are relevant to the dependability of CORBA-based SoSs and that hold for arbitrarily large SoSs, in a well-defined sense (see Section 5.5.2).

In addition, CSP is being used to model an emulation of the CAN protocol on top of TTP/C. By model-checking, we are exploring the effectiveness of this CAN emulation. This work is described in Section 5.5.3.

5.2.7 Coping with real-time

5.2.7.1 The Challenge

Real-time systems are inherently parallel. The real-time constraints come from the fact that other "processes" are evolving in parallel. In many cases these external processes are actually physical systems linked by sensors and actuators. Thus, coping with real-time systems is bound to be at least as difficult as reasoning about concurrency. The extra difficulties relating to the actual passage of time are largely related to finding some abstraction level to avoid having to reason at the level of machines instructions and clock ticks.

It is not in principle difficult to develop notations which can record assumptions or requirements in terms of intervals on a time line. But those "real-time" logics which handle everything in terms of specific points in time and explicit intervals are of little use in a design process. Notations such as the "Duration Calculus" are an attempt to provide more tractable notations which admit relating levels of abstraction.

5.2.7.2 Approaches to real-time verification

Formal reasoning about real-time systems directly (i.e., without first abstracting away from real time) is possible using a theorem prover, but cannot be guaranteed to terminate if automated and is generally harder than for integer-time systems. If a model-checker is used then real-time systems must be abstracted to rational-time or integer-time systems (in order to guarantee a finite state space). So, real-time verification, whether by theorem

proving or model-checking, is either simplified by, or requires, abstraction away from real time.

Recall the three step approach to verification described in Section 5.2.3: (1) map the concrete system to an abstract model; (2) reason about the abstract model to discover some of its properties; (3) map these abstract properties to properties of the concrete system. Crucial to this approach, when verifying real-time systems, is the availability of a suitable abstraction function that maps a real-time concrete system to an integer-time abstract system. Then (importantly) step 2 only requires reasoning about an integer-time system. Identifying this abstraction is a key task; it can be found manually or, sometimes, automatically, as outlined below.

The abstraction from real-time system descriptions to integer-time descriptions is typically induced by an abstraction from real time to integer time. This *time abstraction* might be intrinsic to the system: all component systems might operate with respect to a common, discrete time. (This can be the case for real-time systems if the discrete time is guaranteed to correspond to real time in a precise and well-defined way.) Alternatively, the time abstraction might be a discretization based on 'significant' points in time, which are those times at which 'significant' events occur in the system or its environment. (The significant events are those that matter to the property being verified.)

Manual discovery of an abstraction function for system descriptions is often difficult. It can be labour-intensive and error-prone. Also, manual discovery is not 'formal'.

Automatic discovery of an abstraction function is possible for some real-time (or rational-time) modelling languages. Particular examples are the timed formalisms UPPAAL [UPPAAL] and Kronos [Bozga et al. 1998]. Automatic discovery can be achieved formally.

Alternatively, the system architecture might support a common sparse time base and so allow all nodes of a distributed system to operate with respect to a common, discrete, notion of time. Separate, one-off formal verification of the architecture might prove that it provides the claimed common time base. This is the intention with the Time-Triggered Architecture (TTA). Efforts are continuing to verify that the TTP/C protocol provides a common sparse time base to all participating nodes [Rushby 2002].

5.3 Benefits of ideal LIF characteristics

This section describes some particular ways in which the 'ideal' LIF characteristics of Section 4.4 are beneficial for SoS development and verification.

5.3.1 Formal Reasoning for non-time sensitive SoSs

Where the interface requirements of a system (its LIF) do not have explicit timing constraints, familiar notations such as VDM, Z or B can be used to specify the behaviour. Notice that nearly all systems actually have some performance constraints: what is really being done in non time sensitive descriptions is to separate the performance issue from the "functional" specification. DSoS recognises that there are many systems where such a separation is impossible.

5.3.2 Formal Reasoning for time sensitive SoSs

5.3.2.1 Formal reasoning for temporal firewalls

Temporal firewalls are ideal LIFs for state messages in safety-critical real-time systems. Here we describe how they facilitate formal reasoning.

The temporal firewall has been designed as the fundamental interface of a time-triggered architecture. A temporal firewall is an operationally fully specified digital interface for the unidirectional exchange of state information between a sender and a receiver over a time-triggered communication system. The basic data and control transfer of a temporal firewall interface is depicted in Figure 9, showing the data and control flow between a sender and a receiver [Kopetz 2002b].

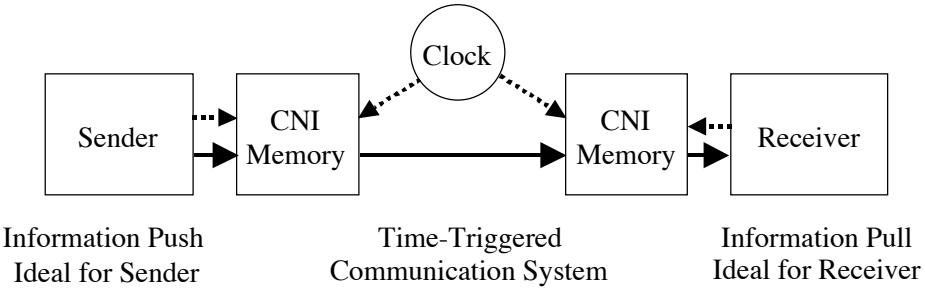

Figure 9 — Data flow (full line) and control flow (dashed line) across a temporal firewall interface

The CNI (Communication Network Interface) memory at the sender forms the output firewall of the sender, and the CNI memory of the receiver forms the input firewall of the receiver. The sender deposits its output information into its temporal firewall (update in place) according to the information *push* paradigm, while the receiver must *pull* the input information out of its CNI (non-consumable read). The transport of the information is realized by a time-triggered communication system that derives its control signals autonomously from the progression of time. It is *common knowledge* to the sender and the receiver at what instants (on a sparse time base) the typed data structure is fetched from

the sending CNI and at what instants this data structure is delivered to the receiving CNI by the communication system.

Importantly, pre- and post-conditions suitable for formal verification can be obtained directly from the temporal firewall interface specifications. The precise operational interface specifications (in the temporal and value domain) of the inputs give the *pre-conditions* for the correct operation of the application software. The precise interface specifications of the outputs give the *post-conditions* that must be satisfied by the application software, provided the preconditions have been satisfied by the environment.

Since no control signals cross a temporal firewall interface, control-error propagation across this interface is eliminated by design. In addition to clear dependability benefit, we believe this has the potential to simplify formal verification.

A temporal firewall eliminates low-level concurrency from the interface. The sparseness of the global time establishes a system-wide *action lattice*, the lattice points of which are precisely synchronized with the global time. The behavior of a system with temporal firewall interfaces can be explained by sequential stepwise progression through this action lattice. The elimination of concurrency from interfaces simplifies understanding, since the human mind is ill-equipped to handle concurrent processes [Reason 1990] Formal verification, too, is simplified by the avoidance of concurrency.

At the CNI within a node of a time-triggered architecture, the records of the RT entities are periodically updated by the real-time communication system to establish temporally accurate RT-*images* of the RT-entities. The computational tasks within the host of a node take these temporally accurate RT-images as inputs to calculate the outputs, which are stored in the CNI and transported by the time-triggered communication system to the CNIs of other nodes at *a priori* determined instants. This known timing of inputs and outputs contributes temporal information to the pre- and post-conditions for formal verification: the outputs are required within an *a priori* known time interval, and that this deadline will always be met can be verified, for any given piece of source code, by worst-case execution time (WCET) analysis.

The *a priori* knowledge of the *receive instant* of a message in a time-triggered system can be used to implement prompt error detection at the receiver of the message. As soon as the instant of arrival of a time-triggered message has passed without the message arriving, an error handling process can be initiated at the receiver. This form of *time-based error detection* at the receiver can also be deployed in systems that provide unidirectional information and unidirectional control flow only. In contrast, in an event-triggered system – where it is not known *a priori* when a message should arrive at a receiver – a bi-directional control flow is required for error detection. This bi-directional control flow complicates the interface, especially when a multi-cast communication topology must be

supported. Formal verification benefits from the simpler interface that is possible for time-triggered SoSs with temporal firewalls as LIFs.

We have seen that temporal firewalls simplify formal verification. The next section discusses the formal verification of TTP/C protocol itself.

5.3.2.2 Proving properties of TTP/C

The TTP/C protocol is intended for safety-critical real-time applications. It provides fault-tolerant scheduled communication between the CNI (communication network interface) memories of TTP/C nodes. Systems of systems can make use of TTP/C's fault-tolerant communication service to connect component systems together, but of course such SoSs can only be fully verified if the TTP/C properties they exploit are verified. Therefore, although the verification of TTP/C was not intended as part of the DSoS Project, it is worth discussing the TTP/C verification challenges that remain.

TTP/C comprises a set of distributed algorithms that provide functions such as clock synchronization and group membership. These algorithms are highly mutually dependent, and ingenious reasoning is required to prove correctness formally; indeed, it is challenging to express the properties of TTP/C in a fully formal manner, let alone to verify them.

In the face of these challenges, much verification work for TTP/C has been performed ([Rushby 2002] contains a good summary). The published verification efforts to date have concentrated on proving the correctness of simple forms of the individual algorithms within TTP/C, and their predecessor algorithms. (For example, group membership has typically been verified under the assumption that clocks remain synchronized.) Furthermore, these verifications sometimes make strong assumptions about the patterns of faults that can arise, thus restricting the verification claims. The algorithms that have been verified, along with the fault assumptions in each case, are summarized in [Rushby 2002].

[Rushby 2002] also discusses a major remaining challenge: reasoning about the various services provided by TTP/C together, rather than in isolation. The constituent algorithms of TTP/C are tightly integrated, for reasons of efficiency, but this in turn makes their formal verification particularly challenging. It is unlikely that the full range of services provided by TTP/C can be proved, without overly-restrictive assumptions, by combining isolated results about the correctness of its individual algorithms. Rushby proposes an interesting style of proof: a form of assume-guarantee reasoning where decomposition is by function, rather than by structure. Circular reasoning is broken by exploiting the progression of time. Essentially, the group membership algorithm could **assume** correct operation of the clock synchronization algorithm in (or perhaps up to) round n, and in return **guarantee** to maintain the group membership correctly in round k+1. It is argued

that this should allow the mutually dependent algorithms of TTP/C to be verified together instead of in isolation.

5.4 Formalizing LIFs and Compositions

Within the DSoS project, the architecture of a dependable system of systems consists of component systems composed together at connections, which are often connection systems. The formalization approach that we have adopted follows this decomposition, formalizing component systems by providing specifications of their LIFs, and formalizing the notion of composition through techniques derived from the field of Architectural Description Languages. Work on the formalization of LIFs has used OMG IDL as a basis for syntactic specifications, and an ADL that is based on UML.

5.4.1 IDLs as syntactic specifications

This section summarizes OMG IDL and discusses its suitability for describing the interfaces of component systems of a system of systems.

5.4.1.1 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 bit value for characters
- *octet*: 8 bit unsigned value (is not subject to conversions)
- *short* and *unsigned short*: 16 bit integer value
- *long* and *unsigned long*: 32 bit integer value
- *long long* and *unsigned long long*: 64 bit 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 *struct* or *union* types, or use several instances by using the *sequence* or *array* type constructors.

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 be declared in OMG IDL but not defined, i.e., only parameters (input- and output-parameters) and results of methods (regular results or exceptions) 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. If a receiver is not under the control of the SoS integrator, a connection system can be implemented to convert the data provided by the sender to the format expected by the

receiver. This allows the integration of different computer architectures to be transparent to the user. A syntactic property mismatch cannot 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 it may happen that no result is 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 method 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.

5.4.1.2 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. Some CORBA systems use a leasing mechanism, 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.

5.4.2 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.

5.4.2.1 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.

5.4.2.2 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 subsequently validated).

The proposed DSoS 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.

5.4.2.3 Formal Verification of ADL Designs

It is widely recognised that the development of any system can benefit greatly from an ability to 'check the design' in order to catch design errors that might otherwise lead to wasted implementation effort. An ADL description of a design provides an obvious opportunity to check the design itself, without the need to wait for an implementation to

become available. Indeed, much research on ADLs for software has concentrated on easing the behavioural analysis of software systems at the architectural level. The ADL developed within the DSoS Project [Nguyen and Issarny 2002] extends the approach to the behavioural analysis of SoS designs. The prototype framework for verification of such SoS designs is summarised here.

ADLs describe systems by describing their architectural elements and the relationships between them. The DSoS ADL, which is based on UML, represents component systems and their interfaces, connections between interfacing systems, etc.

The properties to be checked of an architectural design can be represented in PROMELA using the ADL and incorporated into the design itself. This is natural because properties can often be decomposed according to system structure – though this is sometimes difficult. It is helpful if the ADL allows system properties to be associated with the system description, and to be decomposed into properties of constituent sub-systems if the designer is able to perform the decomposition.

The ADL developed in the DSoS project has been specialized for enabling the behavioral analysis of system architectures using model checking, as follows:

- ADL components are characterized by a property, called "*Body Behavior*", whose value can be assigned to a textual specification, given in any behavioural modelling formalism, describing the components' behaviour.
- UML interfaces provided/required by ADL components are characterized by a property, called "*Port Behavior*", whose value describes in some textual specification, the particular protocol used at that point of interaction.
- ADL connectors are characterized by:
- A property, named "*Body Protocol*", whose value specifies the role-independent part of the interaction protocol.
- A set of properties, named "*Role Protocol*". Each one of these corresponds to an association end, i.e., a role. The value of each property specifies the role-dependant part of the interaction protocol represented by the connector.

From the standpoint of associated tool support, we chose SPIN[Holzmann 1997]because: (i) it is based on a C-like language for modelling system behaviour, which is more familiar to system developers compared to other modelling languages, and (ii) it has built-in channels, i.e., constructs used for modelling message-passing, with which we can easily model parts of the ADL connectors. A model in the SPIN modelling language, i.e., PROMELA, consists of a number of independent processes (each one having its own thread of execution) which communicate either through global variables or through special

communication channels by message-passing, as is done in CSP. Therefore, our basic architectural elements can be mapped to the constructs of PROMELA in a way analogous to the mapping used by the Wright ADL [Allen and Garlan 1994] for CSP. In particular, for each component, connector, port/interface and role, a corresponding process is generated in [Allen and Garlan 1994]. Each generated process communicates with the rest through channels generated as prescribed by the system configuration. However, such a mapping results in the generation of a large number of processes and requires substantial resources for model checking.

To alleviate this problem, we chose to generate independent processes for each component and connector specified in a system architectural description, while for each port and role we generate PROMELA inline procedures. This inline procedure construct of PROMELA allows us to define new functions that can be used by processes, but that do not introduce their own threads of execution. In this manner, we minimise the number of different processes the model-checker will be asked to verify, thus enabling the verification of larger architectures. Then, for each port of an ADL component we declare a communication channel in the PROMELA description of the component, named after that port. This channel will be used by the process related to the ADL component for communicating through that specific port. Since ports of ADL components are bound to specific roles of ADL connectors, their channels are passed as arguments to the processes created for these connectors at the time of their initiation.

Thus, messages sent from a process of an ADL component at a channel corresponding to a port of it, will be received by a process of an ADL connector. Similarly, messages sent from a process of an ADL connector to a channel it has received as argument at initiation time, will be received by a process of an ADL component, whose port was mapped to that channel. The proposed mapping may seem to deprive the architect of the ability to describe complex cases, such as multi-threaded components, but this is not so. Indeed, it is always possible to describe a component as a composite one, i.e., one that consists of a number of simpler components and connectors, which will subsequently be modelled as independent processes. The steps that are followed for generating a complete PROMELA model from an architectural description are given in Table 6.

A prototype implementation of the DSoS development environment, including generation of PROMELA models from architecture descriptions, is presented in DSoS Deliverable CSDA2 [Nguyen and Issarny 2002]. Future work to formalize DSoS architecture will aim to make the behavioural specification of architectural elements more tractable for developers, using a library of architectural elements that characterize common architectural styles.

Component	For each component *c*: • Create a PROMELA process type, "*proctype*", named after the component, whose behaviour is given by the value of "*Body Behaviour*" • For each port *p* of *c*, create an "*inline*" procedure whose name is the catenation of the component's and the port's names, i.e., *c_p*. This procedure contains the *Port Behaviour* of the respective port *p*. For interacting with its environment, *c_p* uses a channel named after the port, i.e., *p*.
Connector	For each connector *c*: • Create a "*proctype*" named after the connector, whose behaviour is given by the value of "*Body Protocol*". Unlike the processes corresponding to ADL components that take no arguments, these processes receive as arguments at initiation time the channels they will be using for their respective roles. These channels are named after the roles themselves.
Configuration	Create a special process called "*init*" in PROMELA, which will be responsible for instantiating the rest of the architecture. More specifically: • The "*init*" process creates as many instances of the processes corresponding to particular ADL components as there are instances of these components in the configuration. • Afterwards, it does the same for each instance of an ADL connector but it uses the attachments of component ports to connector roles to deduce the specific channels that should be passed as arguments to the processes corresponding to the connector.

Table 6 — Generating complete PROMELA models from a DSoS ADL description

5.5 Other DSoS Formalization Activities

5.5.1 Compositional development based on 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.

5.5.1.1 Historical approaches to formalizing CA actions

Several models have been proposed for formalizing 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 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 provide a better understanding of the CA action concept and can be used in developing general models incorporating mechanisms that support 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 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.

5.5.1.2 Compositional development of systems designed using CA actions

In our research within the DSoS project we returned to the modeling of the CA action concept [Randell et al. 1997] to investigate compositional methods for verifying the correctness of dependable systems of systems. To model CA actions we used the most recent version of the ERT model [Burton et al. 2001a]. We have proposed a development method for systems designed using CA actions. The method is iterative and starts with an

initial abstract system design given in the form of a network[9] of concurrently operating processes modelled in the CSP process algebra [Hoare 1985], [Roscoe 1998], [Schneider 2000].

For a high-level description, one can relatively easily verify the relevant correctness requirements using, for example, the FDR model-checking tool [*Failures-Divergence Refinement: FDR2 User Manual* 1992-99]. Requirements such as deadlock freeness, or a particular order of the execution of actions, can be expressed as refinement of suitably chosen CSP 'specification' processes.

Two refinement steps have been studied for compositional development of the initial specification towards a correct implementation. These are *process refinement* and *intercommunication interface refinement*.

Process refinement can be applied to any of the sub-processes of the network – any sub-process can be implemented as a separate sub-network. By doing so, the designer is able to describe the chosen sub-process in a more concrete and detailed manner. Because the substituted sub-process and the network that replaces it have **the same interfaces**, we can relate them using the standard refinement order (_) of CSP, which distributes over the network composition operator (see [Roscoe 1998]).

The second kind of refinement – intercommunication interface refinement – can be applied to the communication channels linking the sub-processes of the network. The motivation for allowing this type of refinement is as follows: when deriving an implementation from a specification, we often wish to implement abstract, high-level interface actions at a lower level of detail, or in a more concrete manner. In the following example scenarios, interface refinement is desirable:

1. The channel connecting one component process (P_i) of the network to another component process (P_j) may be unreliable and so may need to be replaced by two channels: a data channel and an acknowledgement channel;
2. P_i itself may be liable to fail and so its behaviour may need to be replicated, with each new component having its own communication channels to avoid a single channel becoming a bottleneck [Burton et al. 2001a; Burton et al. 2001b; Burton et al. 2002];
3. It may simply be that a high-level action of P_i has been rendered in a more concrete, and so more directly implementable, form.

[9] We define the *network* $P_1 \_ \ldots \_ P_n$ to be the process obtained by composing the processes P_i in parallel and then hiding all interprocess communication, i.e., the process $(P_1 \parallel \ldots \parallel P_n) \setminus B$, where B is the set of channels shared by two different processes in the network.

As a result, the interface of an implementation process may end up being expressed at a lower (and so different) level of abstraction to that of the corresponding specification process. The refinement of communication interfaces in our development method means that we can replace two of the component processes of our network with respective implementations that have **different interfaces** than the specifying components have.

The resulting network still has the same interface as the original one, but since we want to verify our system compositionally (first verify that each component of the implementation network implements an appropriate component of the specification network, and then legitimately conclude that the overall new network satisfies the original specification) we need to relate processes with different interfaces. This can be done using the *implementation relation* (_) introduced in [Burton et al. 2001a] (and further investigated in [Burton et al. 2002]instead of the standard CSP refinement order (which cannot be used in this refinement step, since it relates processes that have the same interfaces).

Our development method relies on the compositionality of the relations _ and _.

The results of our work on compositional development based ofn CA actions are presented in [Burton et al. 2002] where we compare the _ and _ refinement relations and show how to employ them in our scheme. We illustrate our development steps on a practical example of the production cell [Randell et al. 1997].

5.5.2 CSP modelling of GIOP

5.5.2.1 Modelling Context

CORBA is used to build a wide variety systems of systems, so it is of interest to find ways to reason effectively about such systems. This is especially true for large systems, which usually require scalable verification techniques. The work presented here has two aims: 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 rigorously extrapolate the results of the modelling – which is necessarily finite-state – to arbitrarily large systems.

We did not expect our modelling 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, we make assumptions about the underlying transport-level protocol – for example, addressing information is necessarily transport-

specific. Our modelling assumes that the underlying protocols are TCP/IP – so, in effect, we are modelling 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.

The CSP GIOP modelling presented owes much to the research reported in [Kamel and Leue 1998]. In that research, the GIOP protocol was modelled in PROMELA, and basic properties were verified of the system using the Spin model checker [Holzmann 1997]. The main novelty of the results obtained using CSP is the extrapolation to systems of arbitrary size (using data-independence techniques); this is not easily achievable with PROMELA and Spin.

5.5.2.2 Overview of the Model

We have modelled basic ORB processing of GIOP messages.

The OMG allows vendors considerable leeway in the way 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 may *relocate* at will.

By introducing the concept of object relocation early on, we were able to resort to the data-independence theory of [Lazic and Roscoe 1998] in order to extrapolate our results for an arbitrary number of objects (five is the calculated *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 *data-independent induction* [Creese and Roscoe 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 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 for this issue (i.e., through CSP/FDR).

Finally, we have described how the results of certain 2-ORB CSP models (such as those presented here) 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 do this for them automatically. 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.

Further details of this modelling can be found in DSoS Deliverable DSC2 [Fabre et al. 2003], which describes on-going work to enable formal application of data-independent reasoning when model-checking models expressed using CSP_M, the machine-readable version of CSP.

5.5.3 CSP Modelling of a CAN Emulator

Section 4.3.4 discusses the integration of event-triggered (ET) and time-triggered (TT) communication, opting for integration, in the DSoS Conceptual Model, by implementing ET communication on top of TT communication. This approach is exemplified by the CAN on TTP/C emulator being developed at TU Vienna [Obermaisser 2002]. Here, ET messages that conform to the CAN (Controller Area Network) standard are accepted at a sending node by a local CAN emulator, which packs them into TT messages; these are broadcast by the underlying TTP/C bus to remote nodes. Remote CAN emulators then reconstruct the ET data *as if it had been communicated by a valid implementation of CAN* (but see the next two paragraphs).

There are two points to make. First, the final sentence of the previous paragraph needs some explanation. The faithfulness of this CAN emulation, compared with the chosen CAN implementation, can be set in accordance with the demands made of the emulation (by the application).

The second point is that the emulation actually provides extra information, not provided by CAN implementations, by virtue of the underlying use of the TTP/C protocol. In particular, the clock synchronisation and group membership information can be made available to the event-triggered application, as can other parts of the TTP/C interface state. This extra information can potentially be used to improve the service provided by the ET application in the absence of faults, or its dependability in the presence of faults. An example of the former is the removal of the need for expensive clock synchronisation algorithms performed by the ET application on top of the ET protocol; an example of the latter is the provision of group membership information when the ET application doesn't provide it.

On-going work among DSoS Project partners is modelling the CAN emulator outlined above. This work uses CSP to model the emulator at local and remote nodes, and an

abstraction of the TTP/C protocol between these emulators. Its aim is to verify some faithfulness properties of the emulator.

A key feature of this work is the identification and modelling of the essential properties of the underlying TT communications, rather than the full TTP/C protocol; this has two benefits: (1) it simplifies the modelling, and (2) it assures the emulator for use on top of any conforming TT protocol, not just TTP/C. (Indeed, the emulator has been designed to work over a generic TT protocol.)

The property we have aimed to verify by our modelling is a consistency property: that the communications service provided by the emulator is consistent with that advertised for the particular CAN controller emulated. We have split the verification task into separate pieces by structurally and functionally decomposing the target property in a way that closely corresponds to the design of the emulator. In particular, this approach has allowed us to localise the modelling of time, greatly improving tractability.

We expect that this modelling work will help justify the use of the CAN emulator for CAN applications. In particular, the TTP/C architecture – which has been developed to high standards of assurance – will then be available for assured communication between CAN applications.

A very important aspect of this modelling, in the context of DSoS, is its potential to enable formal verification of systems of systems that include both TT and ET component systems, integrated using an emulator such as the one modelled.

6 SUMMARY AND FUTURE WORK

In this deliverable, we have presented the DSoS conceptual model, and illustrated some of the concepts using a series of examples. The case studies (as outlined in Section 1.4) have had a major influence on this conceptual model: it would have been straightforward to provide a set of concepts for one domain; the case studies have ensured that DSoS takes a wider view.

The taxonomy presented in Section 2 attempts to summarize the large number of different classificatory dimensions that can be of use for characterizing and comparing different systems of systems. Equipped with this taxonomy, it should be quite straightforward to position any given system of systems in the taxonomy. This positioning can help one to understand which features of the given SoS are likely to impact dependability.

Many general DSoS concepts have been introduced and motivated in Sections 3 and 4. Extensive work has been carried out to make the concepts, and the corresponding definitions, consistent and yet embrace a wide scope of SoSs. 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. Much effort has been invested in defining the concepts in a way that allows formal treatment, without sacrificing relevance to the real-world; this has entailed a great deal of discussion which, we believe, has greatly benefited the conceptual model.

Finally, Section 5 gives our views on formalization. It provides a comprehensive summary of formal and semi-formal specification and verification techniques that are particularly relevant to systems of systems, and describes a number of formal tasks performed within the DSoS Project that extend and illustrate what can be achieved.

This conceptual model has been validated against the case studies of the DSoS Project, yet it would be useful to test and expand them further by considering more systems of systems. We believe it would be beneficial to structure the concepts into a hierarchy of core concepts and progressively specialised concepts, each suited to a particular application domain or sub-domain. Such a structure would benefit understanding, and allow techniques to be developed and refined either for all SoSs or for particular (sub)domains – in each case with reference to just those concepts that are relevant.

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 modelled 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 □ with an interval of silence of duration □ between any two active intervals, then, we call the time base *□/□-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 □ and □ 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.

A.1 Multi-cluster global time

The number of system components that can be directly connected is limited due to physical constraints, where 'directly' means using one physical connection system without a gateway component (that is, an intermediate system component between component systems). Yet, in order to achieve good synchronization, the jitter of the communication delay must be small, which can only be achieved using direct connections. As a consequence, it is advisable to group component systems into so-called clusters that consist of several directly connected component systems. A system comprising several connected clusters is called a multi-cluster system, where clusters are connected via gateway components. Component systems of one cluster should build their own global time using internal clock synchronization. A common system-wide notion of time can be achieved by synchronizing the global times of different clusters.

Component systems belonging to the same cluster have high functional and temporal coupling, while coupling between the component systems of different clusters is much looser.

Synchronization of the global times of clusters has the effect of synchronizing all component systems, of all clusters, to within a maximum drift rate and/or accuracy with respect to the time of an omniscient external observer (see Section 3.7). Multi-cluster synchronization makes a single global time base available system-wide, which is a prerequisite of a sparse time base. A sparse time base enables the establishment of a consistent order of significant events in a system without the execution of an agreement protocol. Furthermore, a system-wide notion of time simplifies the design and evaluation of algorithms, since they can take advantage of the common notion of time. Furthermore, the quality of multi-cluster synchronization – measured in terms of system-wide precision

– determines the quality of a system-wide image of the environment and, consequently, the quality of control exerted by the whole system.

Clock synchronization for multi-cluster systems must fulfill several properties in order to be deployable in dependable systems of systems; these properties are described in detail in [Paulitsch 2002]. One of these properties is non-interference, which is defined as composability with respect to the precision of clusters. The synchronization of the different global times of clusters must not increase the precision required of the internal clock synchronization of the synchronizing clusters. An increased required precision may require a corresponding increase in the granularity of the global time of a cluster and thus invalidate timing properties of applications that depend on the global cluster time. It can be shown that non-interfering clock synchronization of the global times of multi-cluster systems can be achieved if the clock synchronization algorithms take advantage of the global time at cluster level and of reliable broadcast [Paulitsch 2002].

ANNEX 2. GLOSSARY

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

(Abstract) Interface State: The (abstract) state of a component system as viewed from a particular interface. It is a notional attribute of the interface that is sufficient to explain future behaviour of the component system across this interface.

(Abstract) State of a System: At a given instant, a notional attribute of the system that is sufficient to determine its potential behaviour.

Accuracy: An image is an *accurate* representation of a state variable (it is *valid*) at a given instant if it is value accurate and temporally accurate.

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 connections and interactions between the components of a system.

Behaviour: A sequence of (perhaps timestamped) send and receive operations of a system.

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

Composite Interface: An interface across which composite interactions can occur. A *composite interaction* is one where at least one message is transmitted according to the information pull model (i.e., the consumer of some message exerts control – back pressure – on its transmission).

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.

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 Interface State: At a given instant, the value assigned to a declared data structure that can be accessed via an interface and that records all the stored state that is relevant to (i.e., that can influence) the future essential behaviour of the system at the given interface.

Declared State: At a given instant, the value assigned to a declared data structure that can be accessed via an interface and that records all the stored state that is relevant to (i.e., that can influence) the future essential behaviour of the system.

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.

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.

Duration: A section of the timeline.

Elementary Interface: An interface across which only elementary interactions can occur. An *elementary interaction* is one where all messages are transmitted according to the information push model (i.e., the consumer of each message exerts no control – no back pressure – on its transmission).

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

Error Containment Region: A well-defined subsystem of a computer system that contains error-detection mechanisms 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.

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 the state observation immediately *before* the happening and the state observation immediately *after* the happening. An event observation can be represented by the tuple

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

where the *time of the event* field may be NULL, in which case the observation has no timestamp.

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

Fault: A fault is the cause of an error.

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.

Idempotency: An observation is *idempotent* if the effect of processing it more than once can be made the same as the effect of processing it once.

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

Input Interface: An interface of a system 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.

Interface Model: the model of the concepts a user has in mind when he/she relates the meaning of the chunks of information in a message (which are the results of the syntactic specification) to his/her conceptual world.

Interface State: See *(Abstract) Interface State* and *(Stored) Interface State*.

Jitter: The difference between minimum and maximum latencies.

Latency: The time interval that elapses between a stimulus and a response. For example, the latency of an observation is the interval between the instant of observation of a real-time entity and the instant of use of the observation.

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 resolve property mismatches and thus incorporate these systems into a system of systems with new emergent services.

Linking Interface (LIF): 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: A data structure that is formed for the purpose of communication among computer 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.

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: A 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 Failure: A failure at a service interface of the system.

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 record of the value of a state variable. It may be represented as 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 was observed. The recorded time t_{obs} may be NULL, in which case the observation has no timestamp.

State of a System: See *(Abstract) State of a System* and *(Stored) State of a System.*

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

(Stored) Interface State: The (stored) state of a component system that is relevant to future behaviour at a particular interface. Together with a definition of the system, it is sufficient to explain the behaviour of the component system across this interface.

(Stored) State of a System : At a given instant, the total information explicitly stored by the system (in state variables) up to the given instant.

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

System of Systems (SoS): A system constructed from autonomous component systems, where *autonomous* means independence with respect to existence, operation and/or evolution.

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.

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.

Value Accuracy: An image is a *value accurate* representation of a state variable if the interpretation of the image value by the user is in agreement with the semantic content of the state variable at the instant of observation.

References

[Abrial 1996] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings,* Cambridge University Press, 1996.

[Ahuja et al. 1990] M. Ahuja, A.D. Kshemkalyani and T. Carlson. "A Basic Unit of Computation in a Distributed System," in *10th IEEE Distributed Computer Systems Conference,* pp. 12-19, IEEE Press, 1990.

[Allen and Garlan 1994] R. Allen and D. Garlan. "Formalizing Architectural Connection," in *Proc. 16th ACM-SIGSOFT-IEEE International Conference on Software Engineering (ICSE'94),* pp. 71-80, 1994.

[Allen and Garlan 1997] R.J. Allen and D. Garlan, "A Formal Basis for Architectural Connection," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 3, pp.213-249, 1997.

[America 1989] P. America, "Issues in the Design of a Parallel Object-Oriented Language," *Formal Aspects of Computing*, vol. 1, no. 4,1989.

[Arlat et al. 2000] J. Arlat, J.-C. Fabre, V. Issarny, M. Kaâniche, K. Kanoun, C. Kloukinas, B. Marre, E. Marsden, D. Powell, A. Romanovsky, P. Thévenod-Fosse, H. Waeselynck, I. Welch, I. Zakkiudin and A. Zarras. *State of the Art Survey (DSoS Project deliverable BC2),* University of Newcastle upon Tyne, 2000.

[Arlat et al. 2001] J. Arlat, J.-C. Fabre, V. Issarny, C. Kloukinas, V.K. Nguyen, M. Rodriguez, A. Romanovsky and A. Zarras. *Architecture and Design: Initial Results on Architectures and Dependable Mechanisms for Dependable SoSs (DSoS Project deliverable IC2),* University of Newcastle upon Tyne, 2001.

[Avizienis 1982] A. Avizienis. "The Four-Universe Information System Model for the Study of Fault Tolerance," in *12th FTCS Symposium,* Los Angeles, IEEE Press, 1982.

[Barringer et al. 1984] H. Barringer, J.H. Cheng and C.B. Jones, "A Logic Covering Undefinedness in Proram Proofs," *acta*, vol. 21, pp.251-269, 1984.

[Bozga et al. 1998] M. Bozga, C. Daws, O. Maler, A.O. and, S. Tripakis and S. Yovine. "Kronos: A Model-Checking Tool for Real-Time Systems," in *Proc. 10th International Conference on Computer Aided Verification,* pp. 546-550, Vancouver, Canada, Springer-Verlag, 1998.

[Brownbridge et al. 1982] D.R. Brownbridge, L.F. Marshall and B. Randell, "The Newcastle Connection, or - UNIXes of the World Unite!," *Software Practice and Experience*, vol. 12, no. 12, pp.1147-1162, 1982.

[Burton et al. 2001a] J. Burton, M. Koutny and G. Pappalardo. "Implementing Communicating Processes in the Event of Interface Difference," in *Proc. ICACSD'01,* pp. 87-96, Newcastle upon Tyne, U.K., 2001a.

[Burton et al. 2001b] J. Burton, M. Koutny and G. Pappalardo. "Verifying Implementation Relations in the Event of Interface Difference," in *Proc. of FME 2001, Lecture Notes in Computer Science 2021,* pp. 364-383, Springer-Verlag, 2001b.

[Burton et al. 2002] J. Burton, M. Koutny, G. Pappalardo and M. Pietkiewicz-Koutny. "Compositional Development in the Event of Interface Difference," in *Concurrency in Dependable Computing, ed. P. Ezhilchelvan and A. Romanovsky*, pp. 3-22, Kluwer Academic Publishers, 2002.

[Campbell and Randell 1986] R.H. Campbell and B. Randell, "Error Recovery in Asynchronous Systems," *IEEE Trans. Software Engineering*, vol. SE-12, no. 8, pp.811-826, 1986.

[CAN 1990] CAN. "Controller Area Network CAN, an In-Vehicle Serial Communication Protocol," in *SAE Handbook 1992, SAE J1583,* pp. 20.341-20.355, SAE Press, 1990.

[Caprile and Tonella 1999] C. Caprile and P. Tonella. "Nomen est omen: Analyzing the Language of Function Identifiers," in *Sixth Working Conference on Reverse Engineering,* pp. 112-122, IEEE Press, 1999.

[Clark 2001] D. Clark, "Face-to-Face with Peer-to-Peer Networking," *Computer*, vol. 34, no. 1, pp.18-21, 2001.

[Clarke et al. 2000] I. Clarke, O. Sandberg, B. Wiley and T. Hong. "Freenet: A Distributed Anonymous Information Storage and Retrieval System," in *ICSI Workshop on Design Issues in Anonymity and Unobservability,* Berkeley, CA, International Computer Science Institute, 2000.

[Collette and Jones 2000] P. Collette and C.B. Jones. "Enhancing the Tractability of Rely/Guarantee Specifications in the Development of Interfering Operations," in *Proof, Language and Interaction, ed. G. Plotkin, C. Stirling and M. Tofte*, pp. 277-307, MIT Press, 2000.

[Cousot and Cousot 1977] P. Cousot and R. Cousot. “Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in *Proc. 4th ACM Symp. Principles Programming Languages,* pp. 238-252, 1977.

[Creese and Roscoe 2000] S.J. Creese and A.W. Roscoe. “Data Independent Induction over Structured Networks,” in *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '00),* Las Vegas, USA, 2000.

[Deline 1999] R. Deline. *Resolving Packaging Mismatch (PhD Thesis),* 178, Computer Science Department, Carnegie Mellon University, Pittsburgh, 1999.

[Dobson and Periorellis 2002] J.E. Dobson and P. Periorellis. *Organisational Aspects of Failure (DSoS Project deliverable PCE4),* University of Newcastle, 2002.

[Fabre et al. 2003] J.-C. Fabre, M. Goldsmith, T. Losert, E. Marsden, N. Moffat, M. Paulitsch, D. Powell, W. Simmonds and P. Whittaker. *Validation of Fault Tolerance and Timing Properties (DSoS Project deliverable DSC2),* Report No. 03174, LAAS-CNRS, 2003.

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

[Forman et al. 1985] I.R. Forman, M.H. Conner, S.H. Danforth and L.K. Raper, “Release-to-Release Binary Compatibility in SOM,” *ACM Sigplan Notices, Proc. OOPSLA 1995*, vol. 30, no. 10, pp.426-438, 1985.

[Führer et al. 2000] T. Führer, B. Muller and W. Dieterle. “Time-Triggered CAN-TTCAN:Time-Triggered Communication on CAN,” in *Proc. 6th International CAN Conference (ICC6),* Torino, Italy, 2000.

[Garlan et al. 1995] D. Garlan, R. Allen and J. Ockerbloom. “Architectural Mismatch or Why It's Hard to Build Systems out of Existing Parts,” in *Proc. ICSE 17,* pp. 179-185, Seattle, 1995.

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

[Hauzeur 1986] B.M. Hauzeur, “A Model for Naming, Addressing, and Routing,” *ACM Transactions of Office Information Systems*, vol. 4, no. 4, pp.293-311, 1986.

[Hayakawa 1990] S.I. Hayakawa. *Language in Thought and Action,* Harvest Original, San Diego, 1990.

[Hayes 1993] I. Hayes. *Specification Case Studies,* Prentice Hall International, 1993.

[Hayes et al. 1993] I. Hayes, C.B. Jones and J.E. Nicholls, "Understanding the Differences between VDM and Z," *FACS Europe*, vol. 1, no. 1, pp.7-30, 1993.

[Hoare 1985] C.A.R. Hoare. *Communicating Sequential Processes,* Prentice Hall, 1985.

[Holzmann 1997] G. Holzmann, "The SPIN Model Checker," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp.279-295, 1997.

[Jones 1981] C.B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. Programming Research Group Technical Monograph 25. 1981.

[Jones 1990] C.B. Jones. *Systematic Software Development using VDM,* Prentice Hall International, 1990, 333 p.

[Jones 1996] C.B. Jones, "Accommodating Interference in the Formal Design of Concurrent Object-Based Programs," *Formal Methods in System Design*, vol. 8, no. 2, pp.105-222, 1996.

[Jones 2002] C.B. Jones. "A Formal Basis for some Dependability Notions," in *Proc. UNU/IIST Anniversary Colloquium (To appear),* Lisbon, 2002.

[Jones et al. 2001] C.B. Jones, M.-O. Killijian, H. Kopetz, E. Marsden, N. Moffat, M. Paulitsch, M. Powell, B. Randell, Romanovsky, A. and R.J. Stroud. *Revised Version of DSoS Conceptual Model (DSoS Project deliverable IC1),* CS-TR-746, University of Newcastle upon Tyne, 2001.

[Jones et al. 2002] C.B. Jones, A. Romanovsky and I. Welch. "A Structured Approach to Handling On-Line Interface Upgrades," in *COMPSAC 2002,* pp. 1000-1005, Oxford, UK, IEEE CS Press, 2002.

[Kamel and Leue 1998] M. Kamel and S. Leue. *Validation of Remote Object Invocation and Object Migration in CORBA GIOP using Promela/Spin,* Ecole Nationale Supérieure de la Télécommunication, 1998.

[Kopetz 1992] H. Kopetz. "Sparse Time versus Dense Time in Distributed Real-Time Systems," in *Proc. 14th Int. Conf. on Distributed Computing Systems,* pp. 460-467, Yokohama, Japan, IEEE Press, 1992.

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

[Kopetz 1997] H. Kopetz. *Real Time Systems: Design Principles for Distributed Embedded Applications,* Boston, Kluwer Academic Publishers, 1997.

[Kopetz 1999] H. Kopetz. “Elementary versus Composite Interfaces in Distributed Real-time Systems,” in *ISADS 99,* Tokyo, Japan, IEEE Press, 1999.

[Kopetz 2000a] H. Kopetz. *Preliminary Version of Conceptual Model (DSoS Project deliverable BC1),* University of Newcastle upon Tyne, 2000a.

[Kopetz 2000b] H. Kopetz. “Software Engineering for Real-Time: A Roadmap,” in *Software Engineering Conference 2000,* Limmerick, Ireland, IEEE Press, 2000b.

[Kopetz 2002a] H. Kopetz. *Fault Containment and Error Detection inthe Time-Triggered Architecture,* 39/2002, Technische Universitat Wien, Institut fur Technische Informatik, 2002a.

[Kopetz 2002b] H. Kopetz. “Time-Triggered Real-Time Computing,” in *IFAC World Congress,* Barcelona, Spain, IFAC Press, 2002b.

[Kopetz and Bauer 2003] H. Kopetz and G. Bauer, “The Time-Triggered Architecture,” *Proc. IEEE, Special Issue on Modeling and Design of Embedded Software*, vol. 91, no. 1,2003.

[Kopetz et al. 1999] H. Kopetz, T. Galla, H. Angelow, E. Fuchs, T. Führer and R. Hexel. *Specification of the TTP/C Protocol,* http://www.ttpforum.org, 1999.

[Kopetz and Nossal 1997] H. Kopetz and R. Nossal. “Temporal Firewalls in Large Distributed Real-Time Systems,” in *Proc. IEEE Workshop on Future Trends in Distributed Computing,* pp. 310-315, Tunis, Tunesia, IEEE Press, 1997.

[Kopetz and Ochsenreiter 1987] H. Kopetz and W. Ochsenreiter, “Clock Synchronisation in Distributed Real-Time Systems,” *IEEE Trans. Computers*, vol. 36, no. 8, pp.933-940, 1987.

[Kopetz 2002c] H.S. Kopetz, N. *Compositional Design of RT Systems: A Conceptual Basis for Specification of Llinking Interfaces,* 37/2002, Technische Universitat Wien, Institut fur Technische Informatik, 2002c.

[Koutny and Pappalardo 1998] M. Koutny and G. Pappalardo. *The ERT Model of Fault-Tolerant Computing and Its Application to a Formalisation of Coordinated Atomic Actions,* TR 636, Department of Computing Science, University of Newcastle upon Tyne, 1998.

[Laprie 1992] J.C. Laprie, (Ed.). *Dependability: Basic concepts and terminology — in English, French, German, Italian and Japanese,* Dependable Computing and Fault Tolerance. Vienna, Austria, Springer-Verlag, 1992, 265 p.

[Lazic and Roscoe 1998] R. Lazic and A.W. Roscoe. *Verifying Determinism of Concurrent Systems Which Use Unbounded Arrays,* TR-2-98., Oxford University Computing Laboratory, 1998.

[Lee 1999] E.A. Lee. *Embedded Software - An Agenda for Research,* UCB/ERL No. M99/93, University of California, Berkeley, 1999.

[Marsden 2001] E.F. Marsden, J-C. *Failure Analysis of an ORB in the Presence of Faults,* LAAS-CNRS, 2001.

[Medvidovic and Taylor 2000] N. Medvidovic and R.N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Transactions on Software Engineering*, vol. 26(1), no. Jan. 2000, pp.70-93, 2000.

[Meyer 1988] B. Meyer. *Object-Oriented Software Construction,* Prentice Hall, 1988.

[Nguyen and Issarny 2002] V.K. Nguyen and V. Issarny. *Demonstration of Support for Architectural Design for Dependable SoSs (Deliverable CSDA2),* INRIA, 2002.

[Obermaisser 2002] R. Obermaisser. "CAN Emulation in a Time-Triggered Enviornment," in *Proc. 2002 IEEE International Symposium on Industrial Economics,* 2002.

[OFTA 2000] OFTA. *Software Architecture and Component Re-use,* Arago. Paris, Masson, 2000.

[OMG 1999] OMG. *CORBA Persistent State Service V2.0, Joint Revised Submission,* orbos/99-07-07, Object Management Group, 1999.

[OMG 2000a] OMG. *CORBA Externalization Service V1.0,* formal/00-06-16, Object Management Group, 2000a.

[OMG 2000b] OMG. *CORBAServices: Common Object Service Specification: Event Service Specification.* 2000b.

[OMG 2000c] OMG. *CORBAServices: Common Object Service Specification: Notification Service Specification*. 2000c.

[OMG 2000d] OMG. *Fault Tolerant CORBA Specification V1.0,* ptc/2000-04-04, Object Management Group, 2000d.

[OSF 1992] OSF. *Introduction to OSF DCE, Open System Fondation,* Englewood Cliffs, N.J, Prentice Hall, 1992.

[Owicki 1975] S. Owicki. *Axiomatic Proof Techniques for Parallel Programs.* Department of Computer Science. 1975.

[Owicki and Gries 1976] S. Owicki and D. Gries, "An Axiomatic Proof Technique for Parallel Programs," *Acta Informatica*, vol. 6, pp.319-340, 1976.

[Paulitsch 2002] M. Paulitsch. *Fault-Tolerant Clock Synchronization for Embedded Distributed Multi-Cluster Systems*. Institut fur Technische Informatik. 2002.

[Peti 2002] Peti. *The Concepts behind Time, State, Component and Interface - a Literature Survey.,* 2002/52, TU Vienna, 2002.

[Pnueli 1986] A. Pnueli. "Specification and Development of Reactive Systems (Invited Paper)," in *Proc. IFIP World Computer Congress,* pp. 845-858, Dublin, Ireland, 1986.

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

[Powell 2002] D. Powell. "Time/Event Triggering is Orthogonal to State/Event Observation," in *Workshop on the Integration of Event-Triggered and Time-Triggered Services,* Grenoble, France, 2002.

[Powell et al. 1988] D. Powell, G. Bonn, D. Seaton, P. Veríssimo and F. Waeselynck. "The Delta-4 Approach to Dependability in Open Distributed Computing Systems," in *18th IEEE Int. Symp. on Fault-Tolerant Computing Systems (FTCS-18),* pp. 246-251, Tokyo, Japan, IEEE Computer Society Press, 1988.

[Radia and Pachl 1993] S. Radia and J. Pachl. "Coherence in Naming in Distributed Computing Environments," in *13th Int. Conference on Distr. Computing Systems,* pp. 83-92, IEEE Press, 1993.

[Randell et al. 1997] B. Randell, A. Romanovsky, R.J. Stroud, J. Xu, A.F. Zorzo, D. Schwier and F. von Henke. *Coordinated Atomic Actions: Formal Model, Case Study and System Implementation,* Manuscript, 1997.

[Reason 1990] J. Reason. *Human Error,* Cambridge, UK, Cambridge University Press, 1990.

[Roscoe 1998] A.W. Roscoe. *The Theory and Practice of Concurrency,* Prentice Hall, 1998.

[Saltzer et al. 1984] J. Saltzer, D.P. Reed and D.D. Clark, "End-to-End Arguments in System Design," *ACM Transactions on Computer Systems*, vol. 2, no. 4, pp.277-288, 1984.

[Saltzer 1978] J.H. Saltzer. "Naming and Binding of Objects," in *Operating Systems: An Advanced Course,* pp. 1-105, New York, Springer Verlag, 1978.

[Schneider 2000] S. Schneider. *Concurrent and Real-time Systems: The CSP Approach,* Wiley, 2000.

[Schwier et al. 1997] D. Schwier, F.v. Henke, R.J. Stroud, J. Xu, A. Romanovsky and B. Randell. "Formalisation of the CA Action Concept Based on Temporal Logic," in *DeVa - Design for Validation, 2nd year,* pp. 3-15, ESPRIT LTR 20072, 1997.

[Siegel 2000] J. Siegel. *CORBA 3 - Fundamentals and Programming,* OMG Press, John Wiley, 2000, 899 p.

[Stirling 1988] C. Stirling, "A Generalisation of Owicki-Gries Hoare Logic for a Concurrent While Language," *TCS*, vol. 58, pp.347-359, 1988.

[Szyperski 1998] C. Szyperski. *Component Software,* Addison Wesley, 1998.

[Tisato and DePaoli 1995] F. Tisato and F. DePaoli. "On the Duality between Event-Driven and Time Driven Models," in *Proc. of 13th. IFAC DCCS 1995,* pp. 31-36, Toulouse France, 1995.

[UPPAAL] UPPAAL. *An Integrated Tool Environment for Modeling, Simulation and Verification of Real-Time Systems,* Aalborg University. [http://www.uppaal.com]

[Vachon 2000] J. Vachon. *COALA: A Design Language for Reliable Distributed Systems*. 2000.

[Veloudis and Nissanke 2000] S. Veloudis and N. Nissanke. "Modelling Coordinated Atomic Actions in Timed CSP," in *Formal Techniques in Real-Time and Fault-Tolerant Systems, ed. M. Joseph*, pp. 228-239, Springer, 2000.

[Veríssimo 2000] P. Veríssimo. *Topological Model*. Malicious and Accidental-Fault Tolerance for Internet Applications: Reference Model and Use Cases. 67-74, 2000.

[Vigotsky 1962] L.S. Vigotsky. *Thought and Language,* Boston, Mass., MIT Press, 1962.

[Wayner 1997] P. Wayner. *Human error cripples the Internet.* New York Times. 1997.

[Whitrow 1990] G.J. Whitrow. *The Natural Philosophy of Time,* Oxford Science Publications. Oxford, Clarendon Press, 1990.

[Xu et al. 1995] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R.J. Stroud and Z. Wu. "Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery," in *25th Int. Symp. on Fault-Tolerant Computing (FTCS-25),* pp. 499-508, Pasadena, CA, USA, IEEE Computer Society Press, 1995.

[Xu et al. 1999] J. Xu, B. Randell, A. Romanovsky, R.J. Stroud, A.F. Zorzo, E. Canver and F. von Henke. "Rigorous Development of a Safety-Critical System Based on Coordinated Atomic Actions," in *29th Int. Symp. on Fault-Tolerant Computing (FTCS-29),* pp. 68-75, Madison, WI, USA, IEEE Computer Society Press, 1999.

[Xu et al. 1998] J. Xu, A. Romanovsky and B. Randell. "Co-ordinated Exception Handling in Distributed Object Systems: from Model to System Implementation," in *Proc. 18th IEEE International Conference on Distributed Computing Systems,* pp. 12-21, Amsterdam, Netherlands, 1998.

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