TUB-TCI

A Generic Architecture for Distributed Test Execution

Markus Lepper, 6th September 2002

ISSN 1436-9915

Published as Technical Report 2002/08 by
Technische Universität Berlin,
Fakultät IV – Elektrotechnik und Informatik
Franklinstr. 28/29 • D-10587 Berlin
ISSN 1436-9915
URL=`http://www.cs.tu-berlin.de/cs/ifb/Ahmed/RoteReihe/2002/TR2002_8.ps`

Contents

List of Figures

List of Tables

1 Introduction

1.1 Subject and Genesis of this Paper

This paper presents TUB-TCI, an architecture for automated control of distributed execution of test code.

The topic came across the author as a member of the TTCN-3 compiler project group, headed by Peter Pepper and affiliated at Technical University of Berlin/Fac. IV/ISTI/ÜBB. In this research group Jacob Wieland did the main job in designing and implementing the TTCN-3 compiler, and Baltasar Trancón-y-Widemann is the main author of the generic meta-toolkit, serving as compiler infrastructure.

This research group took part in two discussion fora initiated by ETSI, which dealt with the TTCN-3 related run-time environment and its possible standardization. The final outcome of these ETSI working groups are the specifications called „TRI“ (= „TTCN-3 Runtime Interface“, cf. [tri02]) and „TCI“ (= „Test Control Interface“, cf. [tci03]).

Other participants came from industry (Nokia, Erikson, Telelogic, Testing Technology, *et.al.*) and academia (FhG Fokus), and the interesting and fruitful discussions showed, that in industrial practice there is some need for a generic, versatile, open, simple and powerful architecture for the management of distributed test execution, which additionally takes into account the special needs of test execution in the field of telecommunication devices. Such an infrastructure is called TCI (= „Test Control Infrastructure“) in the following.

Out of the internal discussions of the Berlin research group the construction of TUB-TCI (= „TU Berlin TCI“) arose. This architecture does cover substantially more functionalities than required by *ETSI-TCI* : While this concentrates on the requirements of some TTCN-3 run-time library for use in the field of communication technology, TUB-TCI is totally generic w.r.t. the concrete functionality of code and hardware, and does not give preference to any distinct programming language or execution model[1].

1.2 Intended Purposes and Status of this Paper

The central issue of this paper is to give a *model-based specification* of TUB-TCI: the set of permitted behaviors of any TUB-TCI implementation is specified by constructing a mathematical model which actually performs these behaviors.

There are several different purposes this paper is intended to serve, and there are several others it is *not* :

+ The central part of this paper is the text in section 3, which presents one consistent mathematical model (called the *main model*) describing (some specific possible variant of) the functionality of TUB-TCI.
  The main intention with this collection of mathematical formulæ (together with the natural-language text) is to serve as a precise, *human-readable* basis for further inter-human discussions, lifting these to a more exact and concrete level. While natural language is sufficient and adequate when talking about general concepts, based on well-defined notions related to common experiences, its usage turns out to be rather tedious and error-prone when talking on the detail level of behavior specification.
  Compare e.g. the preciseness and communicability of the „Request For Change“ with the wording „There should be some order-preserving w.r.t. messages“, to the

[1]Indeed the author does intend the application of TUB-TCI in quite different fields of real-time signal-processing, as indicated in figure 13 on page 94 by the symbols chosen for sources and drains of signals.

wording „in clause 112 change $f : \mathsf{set}[\alpha]$ to $f : \mathsf{seq}[\alpha]$“. In the latter version you „can put the finger“ precisely on the very detail you want to refer to.

- But, in spite of its appearance as a kind of standardization proposal, TUB-TCI does *not* intend to propagate all the individual design decisions taken in the main model! Contrarily, — for the purpose to serve as a basis for a precise discussions of just these decisions, the main model had to be designed as *one* single, *consistent* model. Therefore many decisions had to be taken almost arbitrarily, just for constructing the model, not implying any preference to the selected alternative[2].
  This paper is written with the *intention* to be subject to major changes and modifications.
- The specification is not fully formalized w.r.t. *extension*, but covers just one *aspect* of possible TUB-TCI implementations, which can be called „functionality“.
  Any total formalization would probably miss the goal of human-readability. So many important and necessarily covered aspects of a TCI system are not covered by the formal part, but left to human-language description and „common sense“. This will be further discussed in section 1.3[3].

+ So this paper can be read as a practical case study concerning the consequences of the necessity to combine — with distinct aspects and areas — different levels of formalization, and concerning the appropriate mixture of formal, semi-formal and informal means for constructing understandable and precise specifications.

- The specification is not fully formalized w.r.t. *intension*. Again the goal of readability required far-going extensions, up to modifications of the strict and formal mathematical calculus selected as the basic meta-model (here: Z, [Spi92]), see 3.4.4. A translation to pure Z, eliminating all genericity, reflection etc. is of course possible, but tedious. The benefit would be the direct applicability of formal methods and tools, as has been shown, e.g. for type-checking and execution *(sic!)* in [Gri99], for model checking in [Büs03], and for theorem proving e.g. in [San98].

+ Since the paper contains one complete, consistent and practical specification of one central aspect of a real medium-scale engineering system, it can be read as a case study on the requirements for further *language design*, which shall support this kind of „mathematical programming“.

- This is not a scientific paper „on“ distributed systems, and hardly any theoretical questions are concerned.

+ Contrarily, it is a piece of *concrete* engineering, a kind of „hacking with mathematics“: The main model presents a complete and working „machine“ defined by pure mathematical means[4] [5].
  This machine could be directly implemented in a straight-forward way. Because of the abstractness of its interface definitions, this machine and each of its implementations are equipped with „sockets“ to „plug-in“ the application of theoretic analysis methods and results, as well as run-time bridging software to off-the-shelf formal CASE-tools.

Since with increasing processor power and transmission bandwidth, the discussion of

[2]Section 5.1 will deliver in addition some possible alternatives.
Furthermore, some variants in non-critical detail design decisions will possibly be decidable not before experiences with concrete implementations will have been made.

[3]For constructing a fully generic implementation, some neighbor areas of semantics also need to be specified (e.g. value encoding mechanisms, hardware node capabilities, bus topology etc.). These areas of semantics will be described in more detail in section 5.2.

[4]A way of „programming“ which resembles the approach taken by the micro-soft AsmL group, cf. [mic].

[5]A way of „programming“ which really is fun `;-)`

encodings could be lifted to a far more abstract level than only few years ago (e.g. by using CORBA IDL, XML,SDL). So standardization panels will more and more have spare time to deal with more advanced topics, like language definition and behavior specification.

These advanced matters require the application of precise meta-models, the usage of which must strongly be propagated in standardization processes. Psychologically level, an model-based approach like herein should be suitable to bring some Sex&Cryme to standardization discussion, — just because the medium of presentation is a „machine that really works“. So this paper is intended to be useful on this „political“ level, by fiercely campaigning for exactness as a means of creativity.

Since the design principles of TUB-TCI are intended to be as simple and canonical as possible, we hope to present an architecture which is also somehow useful on the „object“ level, — for the current discussions, but even more for future developments.

1.3 Technical Tasks and Operations Covered by TCI

The central technical task of any TCI implementation (as defined above), is to provide a communication framework for test execution, which allows easy and versatile inter-operation of soft- and hardware distributed over several physical nodes. The nodes may be provided by different vendors, and cooperate in changing constellations.

Consider e.g. a setting in which a specialized high-speed testing device and a general purpose workstation cooperate in testing an SUT[6]. For both the workstation and the testing hardware there may exist a TTCN-3 compiler. Some given TTCN-3 source code contains the statements to make the high-speed device do some protocoling of filtered bus traffic, and to make the workstation generate some low frequency test stimuli.

Assume the source has been compiled for all target nodes successfully. Then, to achieve the intended behavior of the hardware nodes[7], the following pure technical operations have to be performed:

(1) The code has to be loaded onto both hardware nodes, (2) some hardware resources (internal to the nodes) must be allocated and configured, (3) communication channels must be established and (4) finally the execution of code has to be started on all nodes almost synchronously. For all these steps a uniform and reliable infrastructure is desirable.

Additionally there may be more specialized hardware test devices, which do not offer the up-loading of user-defined code. Instead they present interfaces for the external programming of certain (internal) *resources*. These resources have to be inquired, addressed, reserved, configured, examined, reset, etc. Also these basic operations should be performable through a TCI implementation in a standardized way. The unlimited multitude of possible types and classes of such internal resources must be dealt with by some *generic* element in any TCI architecture.

In case of a TTCN-3 based test process it can simply be defined that TCI covers all operations still necessary after the compilation of the sources.

1.4 Related Work

There are some proprietary approaches to the TCI problems, i.e. architectures developed by certain companies for internal use. These are neither published nor adoptable by the whole community.

[6]SUT = „System Under Test“. — For a list of abbreviations see section 2.10 on page 25.

[7]This „intended behavior“ of the hardware indeed makes up the „semantics“ the author has had in his/her mind when writing the ATS!

As an official standard there is the „Test Synchronization Protocol“ TSP1 [tsp97], which specifies signatures in a formal way, but behavior or semantics only informally.

Furthermore there are some basic and important theoretical works (e.g. [BCPR99], [UK99], [Tör99]) and also some first practical experiences with implementing TSP1 in a TINA context ([VGSB$^+$99]), but — as far as we know — this paper presents the first attempt of a single, consistent, over-all behavioral specification of a TCI architecture.

1.5 Structure of the Paper

The following section 2 presents a top-down look on TUB-TCI, starting with the requirements and design principles. Then the concepts and ways of operation are described informally. This description is focused on and starts with the functionality at run-time, which of course is the real aim of the efforts and of main interest to the reader. All auxiliary functionality (like boot and error behavior) are just sketched at the end of this section.

Section 3 constructs TUB-TCI systematically by giving a model-based specification of its functional aspect. Because the necessary infrastructure must be constructed bottom up, the more descriptive parts for a first reading will be found (beside in the introduction) in the last subsections.

Section 4 gives guidelines to generate the concrete encodings, by which the service requests are realized on the data-link layer.

Section 5 offers possible variants of the specification and lists open issues and related specification areas.

2 Concepts of TUB-TCI

2.1 Design Principles and General Design Decisions

The design principles of TUB-TCI are induced by some central goals and requirements, namely . . .

- Simplicity.
- Openness (= Extendibility and Portability).
- Reliability.

Simplicity is an important underlying design principle for all basic mechanisms in TUB-TCI. It is firstly required by the need to implement (a subset of) TCI even on primitive nodes with very limited resources. Besides, simplicity is an important prerequisite for ensuring reliability, and it is required for æsthetic reasons[8].

Extendibility means that the architecture must be extendable and parameterizable by third-party definitions of type, data and behavior. Then it will probably also be open to most results of forthcoming development. In our approach the chosen means are mainly *modularity* and *genericity*.

Portability implies the applicability of TCI to communication infrastructure of widely varying kinds. Low-cost portability is a central goal, because it is a prerequisite for the intended inter-operation between hardware nodes from different vendors.

[8]Applying certain categories from the field of æsthetics to technological problems can indeed have substantial positive economic impacts. Consider e.g. the differences in training costs, maintainability of sources, efforts for documentation etc., when comparing some „ugly“ with some „pretty“ language.

As a consequence, (1) the set of all assumptions made about the style and behavior of the underlying system levels (hardware, OS, execution model, communication architecture, etc.) must be as small as possible. Therefore,(2) the formalism and kind of specification technique used for TCI must necessarily be rather *loose*.

Reliability: Because of this looseness of specification[9], much labor has been invested to guarantee *reliability*.

Firstly, the general style in the design of all interactions and transformations is *strictly defensive*. Secondly, there have been some central design decisions taken in the interest of reliability and robustness:

- **Single and Fixed Master**.
  In each TCI test setting there is *one single* central entity providing all necessary static and dynamic configuration information[10] [11].
  This permits e.g. to represent each subsystem by a scalar value used as identifier, which is *unique* w.r.t. the *whole* setting and a known interval of real-time, because only the single master entity is allowed to assign identifiers. This defensive approach is the only way to guarantee referential integrity in a loose specification without complicated (three-semaphore) communication at run-time[12].
- **Hard Reset**.
  It must be considered that the communication lines used to drive TCI commands can be part of the „system under test“ themselves. Therefore a *hard reset* is forseen, which could be performed after each test suite (or even after every test case) to bring each node (and each bus!) into a defined startup state again.
- **Redundant and Simple Encoding**.
  At some crucial points there are fields of redundant data required by the specification of TCI messages, — e.g. the source node of each transmission is encoded explicitly in the message, instead of being derived from the internal protocol of the receiving hardware. This additionally facilitates diagnosis, e.g. by external bus sniffing.
- **Formal Specification**.
  Much effort in scientific Software Engineering has been spent on making critical systems more reliable. The evolved methods and concepts should also be used when implementing test systems, which are not life critical by itself, but shall be applicable for testing life critical systems.
  Therefore TUB-TCI applies the concept of „Formal Specification“: the kernel behavior of TUB-TCI is specified in a precise way by mathematical formulae. The peripheral areas (e.g. language binding, error recovery) are specified in a semi-formal, symbolic way, but nevertheless without ambiguity.

[9]. . . . but indeed because of the heterogeneity of the practically existing hardware pools, which is only reflected by that looseness

[10]Indeed this entity is implemented in *two* subsystems, CAS and TM, realizing two levels of „strategic knowledge“, –see below in 2.6.

[11]Future times could require a distributed mastership, when e.g. making „planetary distance“ remote tests ($\odot \longrightarrow \Psi$) with high delay time in communication. Such a design could perhaps be derived from the mono-master design presented herein by finding „sub-objects“ and „sub-categories“ in the network of flow of data and control.

[12]This design may indeed cause significant execution time overhead when creating objects in run-time. At the moment this seems acceptable: `create()` commands are assumed to happen in TPrep, and in TPrep execution time is not considered critical in the current version of TUB-TCI.

Figure 1 Hardware Nodes and Testing Sand-boxes

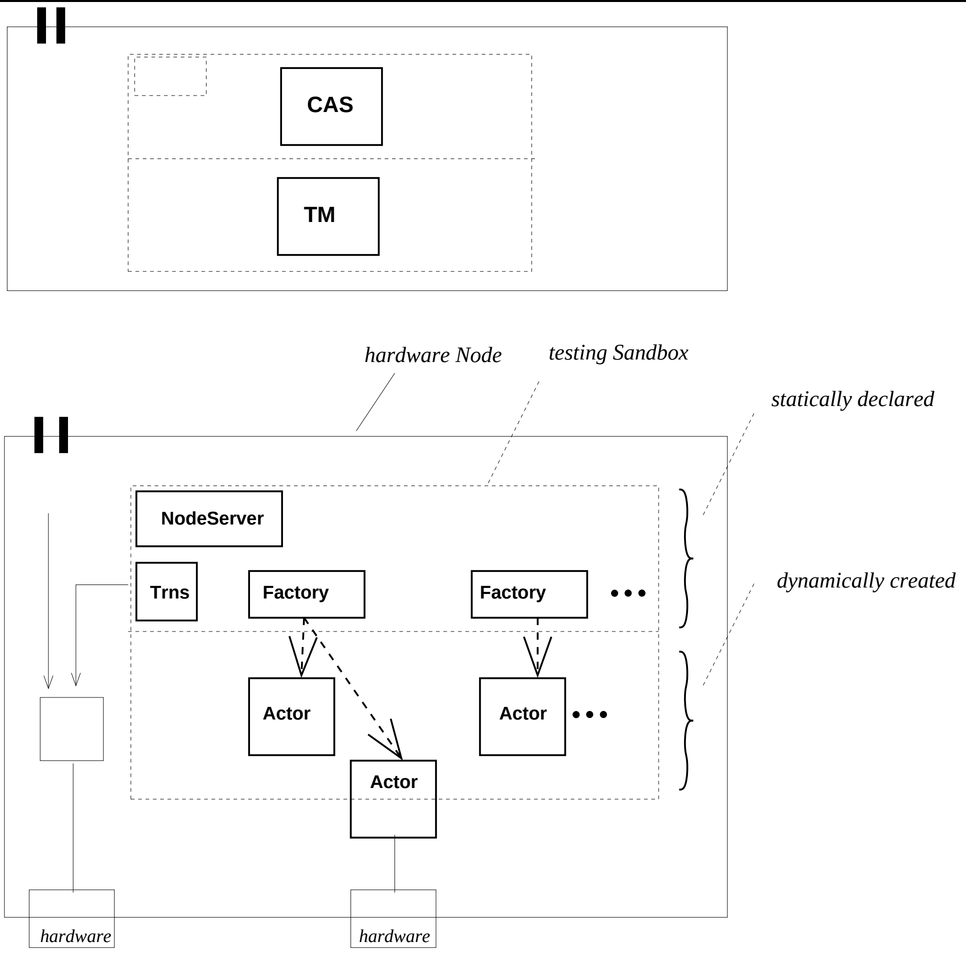
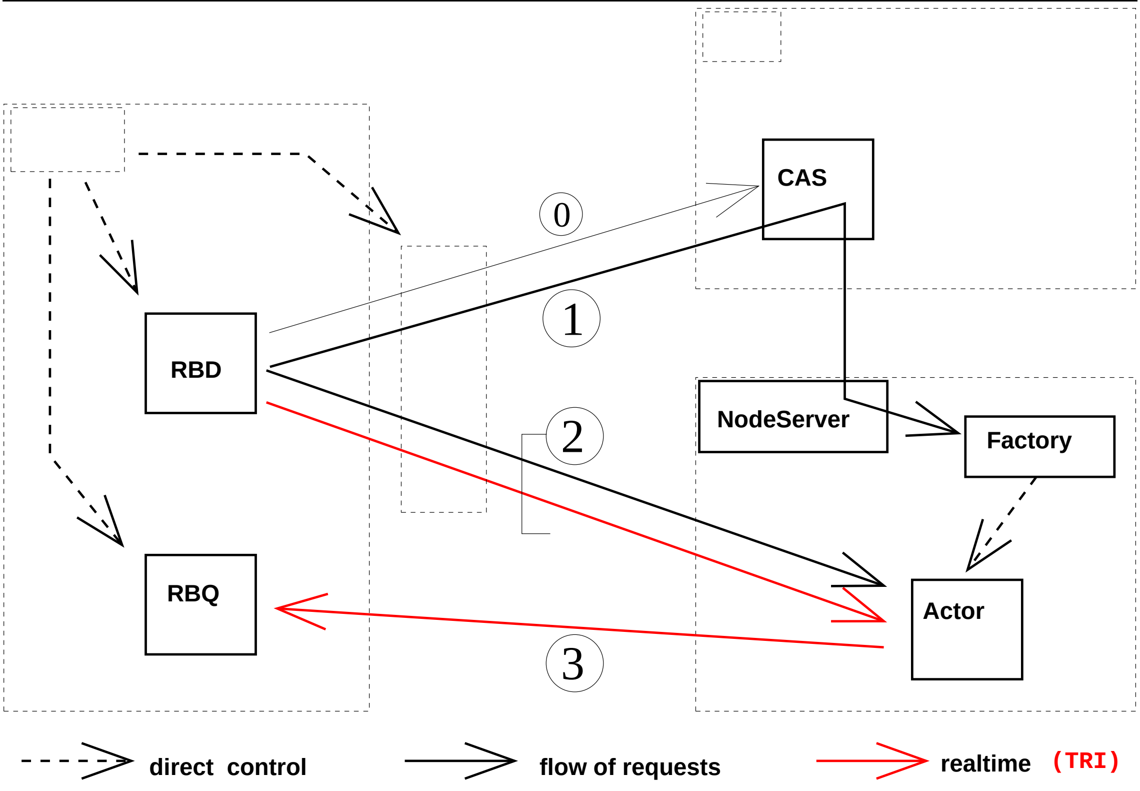


Figure 2 Most Frequent Paths of Service Requests



2.2 Imperative Test Specification : Execution Model as Semantic Model

Historically and in today's practice there are most different ways of making a psycho-internal image of „testing“. The spectrum extends from pure mathematically given specifications (or models) of semantics, via executable programs in a functional or (constraint-)logical high-level language, up to a traditional imperative style of modeling.

The last is the viewpoint taken by languages like TTCN-2 and TTCN-3: The test procedure is specified as „really a procedure“, and is given as a traditional imperative program[13].

The semantics in the head of the author while authoring such a test program, are determined by the imagination of the further technical processing: The program will be compiled, and then loaded onto some hardware. and then, as long as the compiled code is „being executed“ (i.e. as long as the behavior of the hardware ensemble is determined by the imperatively given algorithm) this hardware will continue …

- to send stimuli to the SUT[14],
- to monitor and analyze the reactions of the SUT,
- and to compare this input with the set of expected reactions, i.e. with those reactions which are allowed for the SUT to pass the test.
- In case of violation or fulfillment of this expectation, a `verdict` can be raised, — an event which normally stops the test execution loop, e.g. for starting the next

[13] Of course tests defined in some higher-level declarative language will finally, e.g. after being compiled, end up as imperative code as well.

[14] SUT = „System Under Test“. — For a list of abbreviations see section 2.10 on page 25.

test-case automatically[15].

The totality of all behavioral definitions given by a certain test case or test suite on the source level (e.g. as a TTCN-3 source text) is called *Abstract Test Suite* (ATS).

The totality of all fragments of executable code resulting from the compilation of a given ATS is called *Executable Test Suite* (ETS).

2.3 Physical Architecture, Node Topology

TCI now comes in play after compilation, in case that the hardware intended to execute the ETS consists of several independent nodes.

The physical picture of a typical TCI setting w.r.t. **space** is an ensemble of computing devices from most different families: general-purpose PCs, specialized real-time filters and routers, bus analyzers, together with the monitoring firmware in all these devices, etc. Each of these „physical" computing devices is called *hardware node* (or just *node*) in the following, if and only if it behaves independently from all other hardware nodes, — e.g. can be reset, initialized and configured on its own.

These physical devices of course must be connected by physical communication channels. These, too, may be of most widely differing types and technologies. These communication devices (or some virtual systems built atop of them) are called *busses* in the following[16].

The set of all nodes and busses involved in performing a given test case is called *test ensemble* (TEns) in the following.

> **Please notice** that each hardware node which is able to execute ETS code or which offers hardware devices, does do this is a dedicated and restricted area, called (*testing*) *sandbox*. Figure 1 schematically shows two hardware nodes, their sandbox areas and the position of the different TCI „subsystems", as defined in the following sections. The figure also shows symbolically how the physical reality of the hardware node (outside the sandbox) can be mapped into the reign of TCI using trns and actor subsystems.
>
> It is exclusively the contents of this sandbox which is controllable by TUB-TCI. Therefore, for sake of simplicity, (1) the following figures will only show these sand-boxes, — cf. figure 1 with figure 4, (2) the following text simply says „(hardware) node" when referring to this sandbox, and (3) the formal specification of TCI-behavior in section 3 refers to the behavior of these sand-boxes.

W.r.t. **time** we refer to some distinct, continuous interval of real-time as a *test session* (TSess), if during this interval testing processes are prepared, performed, protocoled etc. The internal organization of test sessions is out of scope of TUB-TCI (and of any TCI in general), but related to some (mostly informally given) test strategies, in-house process regulations etc. From TUB-TCI's viewpoint a test session is seen as a sequence of sub-intervals, which alternate between *test preparation* and *test performance*.

Those intervals of real-time, in which the user intervenes for configuring and preparing the next test are called *test preparation phases* (TPrep). During TPrep the *code objects* a given node has to execute in a given test case has to be up-loaded and then the logical communication channels have to be established.

The intervals of real-time, in which an executable test case or an executable test suite is performed (i.e. the interval in which the addressed devices are acting totally

[15] In case of TTCN-3 most parts of the three activities (ie. sending of stimuli, analyzing the reactions, calculating the verdict) must be calculated „by hand" in an imperative style, — beside some support by pattern matching and genericity.

[16] Even if they are just one-to-one connections.

determined by the semantics of the ETS) is called *test execution time* or *run-time* (TRun) in the following. This phase is entered when the „test execution“ is started by some controlling instance. During TRun the necessary real-time communication between nodes must take place.

> **Please notice** that different nodes can be in different phases, because there must be some system command switching each node from TPrep to TRun. Looking at it very precisely, this command does never reach any two nodes synchronously.
>
> This phænomenon of „relativity“ resembles the fact that on each distinct node a *different* „local time“ must be used to represent the abstract scalar „time“.

2.4 Features and Functionalities offered by TUB-TCI

From these scenarios one can deduce the necessary features and functionalities. Any implementation of TCI provide means for ...

- resetting and initializing hardware nodes,
- loading code fragments for execution onto nodes,
- allocating hardware resources on remote nodes,
- accessing (=inspecting and manipulating) hardware resources on remote nodes,
- preparing and configuring communication channels,
- starting and aborting test execution (i.e. switching a node between TPrep and TRun),
- passing real-time control and data information between nodes,
- controlling and monitoring the transport of real-time data.
- distributing the global test parameters,
- collecting the verdicts raised by an ETS.

These functionalities will be addressed by clients from two very different groups. Those listed first will be used by a super-ordinated test control software (TM) for preparing a TRun. This will happen mostly during TPrep. The latter will be used by the executed ETS itself during TRun to perform the real-time interactions corresponding to the statements contained in the ETS.

For sake of versatility there are two further central requirements on TUB-TCI:

Generally we state that w.r.t code deployment, hardware allocation and run-time communication control, that ...

- the *static* and the *dynamic* way of decision taking (i.e. the compiler-based and the run-time-based scheduling of resources, happening in TPrep or during TRun) must equally well be supported,
- and both methods should even be arbitrarily *combinable* for controlling the same TRun.

On the technical level of communication there is also a combination of two paradigms:

- The aim of inter-operability requires the architecture to be based on some standardized message format (here: XML-based),
- but issues of performance speed may require to integrate *additionally* some specialized and/or proprietary binary encoded high-speed communication channels.

2.5 Behavior Specification by Parallel Execution, — Multiple Sources of Activity

The overall behavioral specification as given by ATS and ETS is mostly realized as a collection of co-operating co-routines or (virtually or physically) parallel executing threads.

So the total semantics of ETS have to be split into those of separate processes. The common execution of all these processes (according to a certain schedule) is required to realize the semantics. Therefore each of the processes needs a kind of „active object“, which performs the execution of the process code as its behavior.

Concerning the identification of active objects and of sources of activity there are widely differing pictures, depending on the viewpoint:

- Seen from the viewpoint of the TTCN-3 author, i.e. considering execution w.r.t. the TTCN-3 semantics, there are exactly the `component` objects as only „active objects“, i.e. the only source of activity in the whole system. Each call of the `component`-constructor call in execution of the source text can be considered to correspond to the instantiation of exactly one thread (on a given node). This thread is going to perform the compiled code of TTCN-3 behavior definition, which has been assigned to this `component` object in the sources.
- In a „finer“ view – i.e. looking at lower system aspects – there are the producer-style subsystems, forming a further group of active objects (e.g. hardware timers, input ports, demon threads polling passive sensors). They create stimuli unexpectedly and independently from the flow of control in the TTCN-3 code. This model of activity is also depicted in figure 3 by boxes with thick borders.
- In the topmost perspective we have just the opposite interpretation : Here some „Test Manager Application“ is meant to *control* all activities in all hardware nodes (including possibly the SUT). From this point of view, this user interface – it may be interactively controlled or batch oriented – is the only source of activities[17].

2.6 Logical Architecture, Categories of Subsystems

TUB-TCI does take just another, different view to execution:

- We call the „*total behavior* of an ATS(ETS)“ *all* active code fragments and processes which are executed by the hardware nodes of a TEns when performing this ATS(ETS). So the „total behavior“ comprises the user defined code itself (= the ETS), and *additionally* all system processes, drivers, infrastructural demons, interrupt handlers etc.

During any TRun there is exactly one total behavior ruling the behavior of the TEns, depending on the selected ATS.

In TUB-TCI, this total behavior is now realized by the parallelization of different fragments of (active) code, which are represented as so called *subsystems*[18].

It is of central importance that this definition is *complete* w.r.t. extension: all code running physically during TRun is uniquely assigned to a single subsystem. The identity

[17]By the way – concerning the question for the sources of *data flow* the latter dichotomy indeed causes problems : Many times the ATS/ETS source text and the concrete Test Manager Application *compete* in defining test values (e.g. „Module Parameters“ or „IXITs“).

[18]The subsystems correspond to the concrete running tasks on a node only as a concept, because on implementation level two things will frequently happen : Conceptually different subsystems will be implemented in *one* piece of code, – and one single subsystem can be realized by more than one running threads or processes.

of this subsystem is always known, there is no „hidden“ code outside this formalism, even if all other aspects of the code are totally undefined.

The TUB-TCI-specification is *not* complete w.r.t. intention: there are large semantic areas, which are application dependent and not specified (or only generically defined) in TCI.

Any subsystem is characterized by the following facts:

- It is always known when it is created and when it is deleted.
- Throughout its lifetime it is hosted on one and the same distinct hardware node[19], called the *„hosting node“* of the subsystem.
- It is identified by one single constant address (e.g. by a *NodeIdent* or a *LUID*).
- It belongs to exactly one of the *categories of subsystems*[20] defined in TUB-TCI.

All subsystems belonging to the same category are called „instances“ of the category. **Please notice** that we frequently use an abbreviating terminology like „. . . any actor subsystem . . . “ or even „. . . any actor . . . “ instead of „. . . any subsystem of category actor. . . “, whenever confusion is impossible.

2.7 Service Request Interaction as Evaluation Mechanism

All evaluation in the TUB-TCI-specification takes place by „Service Request“-style interactions between two subsystems. In such an interaction one subsystem takes the *client* rôle, the other the *server* rôle.

In the following we will say that the client *„generates a service request“* (=*„SRQ“*) (and *„sends it to the server“*), — or that the client *„requests a service (from some subsystem)“*.

The subsystem acting as a server *„executes the SRQ“* or *„processes the SRQ“*, and may send back a *„reply“*[21]. So every subsystem *must „offer“* some services, ie. define some services callable by other subsystems, to be of any use at all.

Additionally every subsystem *may* request services, either out of the execution of another service, or out of some „active“ behavior.

The definitions of which services are offered and requested is totally defined by the category the subsystem is an instance of. For each category of subsystems it is defined . . .

- How many instances are allowed/required per TEns/node during a valid TSess, as indicated in table 2.
- Which global system phases („epoques“) limit the life-cycles of its instances.
- Which services (minimally and maximally!) are offered to other subsystems, — i.e. how the instances of this category behave in the server rôle, as listed in table 4.
- Which services (maximally!) may be called out of the execution of the subsystem's code, — i.e. how the instances of this category behave in the client rôle.

So each category defines the set of services which any instance must implement and offer to the other subsystems (cf. table 4), and the set of services maximally *requested* from within its active behavior. This active behavior of its instances itself is also, more or less, determined by the category.

[19]Except the „virtual“ subsystems QAS and trns, which are realized on each hardware node once, but exist on the conceptual level, ubiquitously, only once.

[20]The wording „category“ has been chosen instead of „class“, to avoid confusion with *actor classes*, which reside on the object level of the language.

[21]All these interactions are realized as message-exchanges of `deliver()` message in the trns subsystem. These again are realized as „electric interactions“ between the corresponding BusAdpt subsystems.

Indeed, the version of TUB-TCI presented herein consists substantially of a collection of nine (or more, as you count it) cooperating, but independent specifications, — one per category of subsystems, following the principle of modularity.

This allows to realize a complex behavior (on different nodes or even on one single node) by combining implementations of subsystems by different vendors or programmers, written in different languages.

2.8 Subsystem Categories defined in TUB-TCI

In a first approach TUB-TCI identifies and defines nine to twelve categories of subsystems (cf. table 2 and table 1), which can be characterized as follows:

- All instances of all subsystems offer some services.
- The sets of identifiers („names“) for the offered services are pair-wise disjoint for all pairs of categories. So there is a function from service names to server category, and a clean graph from category to category, spanning the flow of *SRQS*, as shown in figure 3 on page 17.
- All categories (except actor and RBQ) can (and probably will) also act as clients.
- For some categories (trns, CAS, QAS, NodeServer) TUB-TCI contains a *total* specification of the behavior of all their instances.
- The other categories are only partially defined, and their behavior depends widely on the application level semantics. Nevertheless, the limiting sets of minimally offered/maximally requested services, as described above, are always known.
- A few of these categories (trns, HsLink, ...) realize some general, IP-like mechanism of data exchange between physical nodes. They form the low-level layer of the architecture, combining some kind of „data-link layer“ and „transport layer“ functionality.
- The subsystems of the other categories are situated on top of this layer, using it for the realization of their basic activity, which is the *SRQ*-interaction. These eight (TM, CAS, NodeServer, Factory, Producer, (Actor,) RBD, RBQ, QAS) form the upper layer, a kind of application layer (cf. figure 6). They can be shortly characterized as in table 1, and will be discussed in detail in the next sections.

Figures 2 to 5 show some of these subsystems deployed onto different nodes, together with the most important flows of service requests in different grades of abstraction.

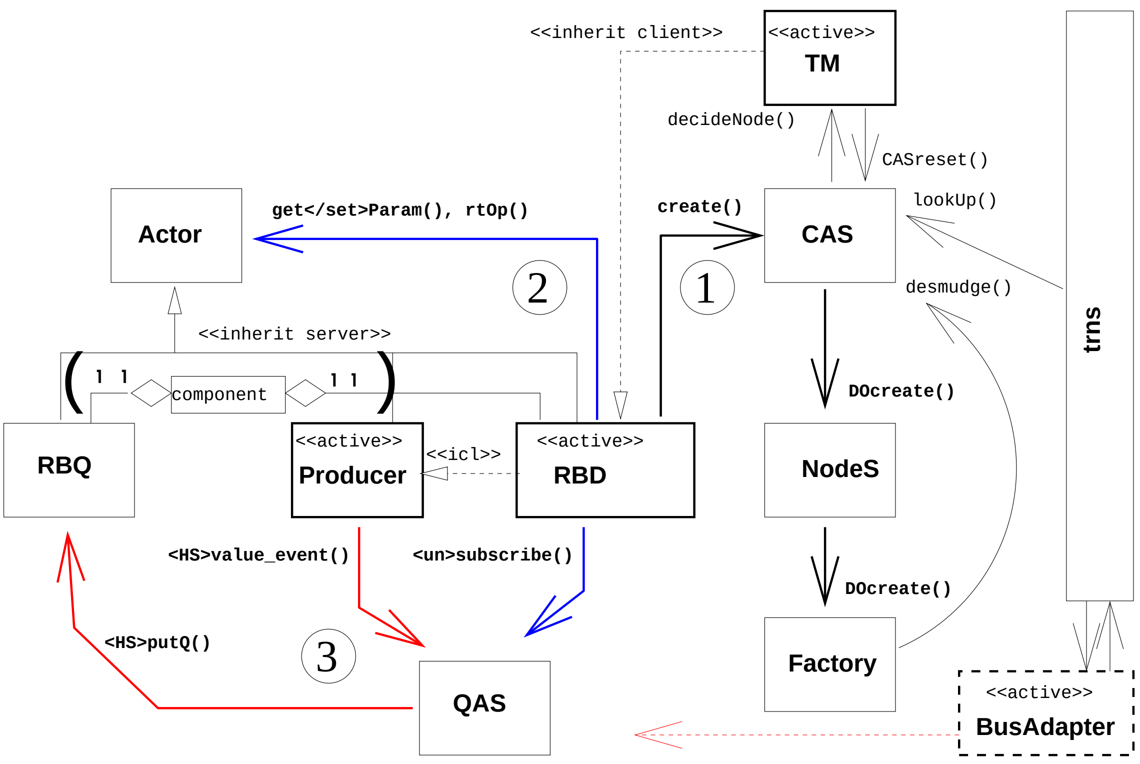
2.8.1 RBD and RBQ = Run-Time Behavior Dynamics and Queues

The first two categories of subsystems (RBD and RBQ) represent the active and the passive aspect of the compiled user code, i.e. of the ETS, and the underlying principle has been taken over from some historic approaches:

The viewpoint of the TTCN-*x* source semantics as mentioned above had already been reflected in the design of GCI [GCI96], the run-time interface definition originally developed for TTCN-*two* support: In the GCI/TTCN-2 context the ETS was given by one single „user level“ test procedure, the compiled code of which includes calls to the functions of the run-time library. This library itself offers only *passive* interfaces, — the only *source of activity* when running the ETS is that single user-defined procedure. The realizing code for this active aspect of the ATS was called RBD (= „Run-Time Behavior/Dynamic“).

In „classical GCI“ the active behavior was completed by a second, separately given and inverse interface, called RBQ (= „Run-Time Behavior/Queue“). This interface realized the *passive* aspect of the ATS , i.e. the rôle as „receiver“ of incoming messages. The

Figure 3 The Flow of Service Requests w.r.t. Categories of Involved Subsystems



functions of this interface had to be called by active objects on system level, ie. interrupt handlers or polling demons, which realize some producer-style, active device drivers.

Since this design is really canonical w.r.t. industrial coding practice, it is preserved in our new approach, and so are the names RBD and RBQ.

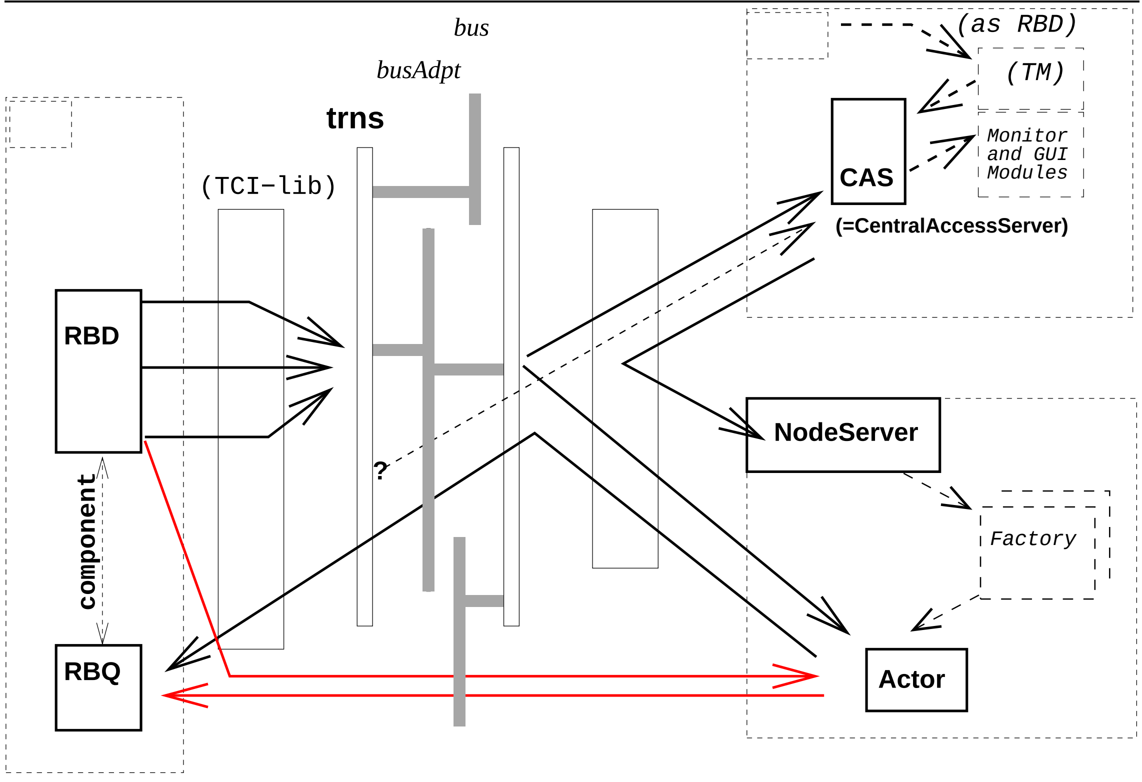
Furthermore TTCN-3 does overtake from the old TTCN-2 concept the treatment of the temporal semantics of externally generated, received events: On the RBQ side, all incoming messages are just entered into a queue as soon as they are received „physically“. But the time when to perform the *analysis* of the incoming message, and the time when some *reaction* is performed, is totally determined by the execution of the corresponding *active* part (RBD): In case of TTCN-2 and TTCN-3, messages are not considered until the execution of the ETS reaches the execution of an `alt` statement: Whenever the active part performs an `alt` statement, the first message is dequeued and analyzed. Then the execution of the active part can branch accord to the results of this analysis and perform the corresponding reactions.

But now, in case of TRI/TTCN-3, we have to deal with parallelism on the level of authoring: The two single interfaces RBQ and RBD are replaced by a multitude of subsystems, and every `component` object in TTCN-3 source semantics has to be realized by *two* TUB-TCI subsystems, one of category RDB and one of RBQ[22].

Compared to GCI, here abstraction and separation enlarges the applicability of the

[22]On the implementation level the correspondence between these both subsystems of course has to be reflected, e.g. by one parameter value field of type *LUID* on each side, pointing to each other. This is depicted in figure 3. But indeed the mapping of the `component` source level construct to subsystems is just a practical matter of the *usage* of TCI, and outside the scope of this specification.

Figure 4 Servers as seen from „Run-Time Behavior“



concept „RBD&RBQ“ significantly: E.g. a counter or a protocol writer can be declared of just RBQ-style, without having a corresponding active behavior, — a filter object doing some recoding of a signal stream can be declared as Producer (i.e. as signal source, see next section) *and* as RBQ, thereby being insert-able between an arbitrary Producer and each RBQ, — etc.

Please notice that RBD and RBQ „include“ or „inherit“ the services of actor category. These services are required to create, initialize, control, delete etc. the subsystems. This fact is described in detail in sections 2.8.9.

2.8.2 Actors, Actor Classes and their Generic Definition

To perform testing of physically existing hardware systems, their input and output devices must be „lifted“ (1) into the sematic sphere of the applied programming language (e.g. TTCN-3) and (2) into the TCI run-time collection of subsystems. The latter is done by mapping them to dynamically created subsystems of the actor category[23].

Each single actor subsystem represents one distinct physical device (or one separately controllable „logical channel“ of a physical device). Actors can be used to model timers, ports, display elements, input switches etc.

Only the RBDs operates on these actors by issuing control commands and sending output data to them. Actors (if they are not producers, see next section) are „passive objects“ from the code's point of view.

[23]The wording „actor“ is somehow misleading, but all alternative proposals („device“, „tidget“, „object“) also carry wrong connotations. Misleading is, that actor subsystems (1) need not have any *act*ive behavior at all, and (2) need not be realized by an *active* object.

Each actor is related to exactly one(1) *actor class*[24]. An actor class describes the possible control operations and data types which can be issued by an RBD to each actor of this class.

The definition of the different actor classes is (naturally) *not* part of TUB-TCI. Instead TUB-TCI defines a *generic interface* for plugging-in declarations of actor classes. These declarations have to describe the *configuration parameters* for installing and modifying actor subsystems, and have to define the *run-time operations* they can be subjected.

Beside this generic declaration interface — which at the moment only lives on concept level — TCI comes with a library of general-purpose *actor classes* (cf. section 3.14), and requires from every actor a few basic services for receiving configuration control commands and run-time operations. In general the *information contents* of these interactions are treated *opaquely* by TCI.

Please notice that, as mentioned above, the subsystems of categories RBD and RBQ also implement the (few) services defined with the actor category.

This can be considered a case of „specialization“ or „inheritance“. But from the point of view of the generic actor (meta-)category, RBD and RBQ are *predefined instances* of actors, not sub-classes! This complicated but fundamental relation is described with more details in sections 2.8.9.

2.8.3 Producers = Event Stream Producing Actors

One sub-category of actors is used to model those external hardware devices, which can produce stimuli to the system „on their own“ and „spontaneuosly“. To these devices belong timers producing the events indicating their expiration, input ports producing an event on each arrival of a datum, switches producing an event when their position is changed etc. The representing actors are called „Event Stream Producing Actors“, or producers for short[25].

Whenever the hardware device represented by a producer subsystem detects an external event, this is signaled to TCI by a `value_event()` *SRQ*, generated by the active behavior of the corresponding producer.

2.8.4 QAS = Queue Access Server

These `value_event()` *SRQS* generated by producers are serviced by the QAS subsystem.

To receive `value_event()` messages from a distinct producer, an RBQ has to *subscribe* for it. This subscription is done by another service offered by QAS.

All `value_event()` messages are distributed by QAS to all RBQs which are currently subscribing for the source of the signal, and to all remote nodes which also host currently subscribing RBQs.

The QAS subsystem is *realized* by one task running on each node, and will normally be implemented in the same piece of code as the NodeServer. In this specification it appears separately, because on the *conceptual* level there is only *one single* QAS server in the whole TEns (while the NodeServers are distinct individuals with each different capabilities, corresponding to the different types and configurations of hardware of nodes they have to manage).

[24]Which on the other hand can combine the features of more than one different actor classes, see below section 3.9.

[25]Technically, actor classes for producers are not modeled as a category of actors on its own, but are realized using the genericity of actor, — they simply are actor classes which „include“ the features of the predefined abstract actor class *Event_Stream_Producer*.

2.8.5 CAS = Central Access Server

Following the design principles of reliability and ease of implementation, there is – as mentioned above – conceptually one single central entity managing and holding all knowledge which is global w.r.t. the whole TEns. This entity is realized by *two* categories of subsystems, each of them having *exactly one* representative in every possible TEns.

One of these categories is called „Central Access Server" (CAS), and its specification is totally in the scope of this specification. The other one is called „Test Manager" (TM), and only a small subset of its functionality is defined within TCI, — see section 2.8.6.

The CAS gathers and distributes all relevant information concerning deployment, which is deducible from the operations performed internally by TCI.

Consider e.g. the problem of referential integrity, i.e. how an RBD can reliably reach the interface of an actor in a distributed test environment, without knowing the exact times of its creation and deletion. In TUB-TCI this is accomplished by the rule, that all `create()` commands performed by an RBD are directed to the single one CAS. All operations on remote actors, from which the hosting node of the physical implementation is not yet known, are feasible by asking the CAS *once* for the „address" of the hosting node.

This solution, caused by the requirement of reliability, has the second advantage that a TM (and any other diagnosis tool) can retrieve information on the state of the whole TEns, all existing actors and all running RBDs just by inquiring one single source of information, the CAS.

Like any other subsystem the CAS is hosted on a single node. This node is called CASnode. It is required for the topology of the network made of nodes and busses, that

- each node is reachable by the CASnode,
- and each node can reach the CASnode.

Please notice that this does *not* imply that all busses in a TEns have to be bi-directional. Indeed it is possible that the CASnode reaches a given node N via a node N_1, while N reaches CASnode via some other nodes, not crossing N_1. The architecture of TUB-TCI but *does* imply that each node has an output channel, i.e. not only can receive service requests, but can also send back some confirmation.

2.8.6 TM = Test Manager Application

The super-ordinated test control software is called Test Manager (TM). Its behavior is out of the scope of TCI, – from its point of view a TM is just a special kind of RBD (see below), with the main difference that it is the only RBD which is initialized „from outside", i.e. which is active before any TCI activity takes place.

The overall deployment information (which code has to be loaded onto which node, on which node a given timer, port or component has to be installed, how the nodes are connected physically and how the network addresses are assigned, etc.) are *not* represented in the scope of TCI, but have to be realized in the TM.

There are only two defined services offered by TM to CAS, by which the TM offers some required information passively (cf. table 4): By the first the CAS lets the TM decide, on which node a new actor subsystem demanded by an RBD shall be created, – the other allows to ask for the current values of the global *test parameters*.

So the TM has to pass all other some necessary portions of the overall configuration information to each node in the TEns *actively*, i.e. has to configure the whole TEns by issuing the correct sequence of TCI service requests (in fact to the NodeServers, see below).

How this sequence is scheduled in a sensible way is not specified in TCI, but can easily be concluded from the specified subsystems' behavior.

2.8.7 NodeServers, Factories and „Drivers“

The creation of the physical actors can only be done locally on the hosting node. This is the central job of the the NodeServer category of subsystems. Each node hosts exactly one NodeServer.

The real work of creating new actors is done by a factory. The NodeServer just passes the create request and its parameters to such factory and passes back the results.

In the sphere of a concrete **implementation** these factories can be realized as „classes“, and the creation methods can be seen as „static“ methods of such a class, while the functionalities offered by created actor subsystems can be seen as „member functions“ or „non-static methods“. Then the class realizing the factory also realizes a kind of „driver“ for the functionality of its dynamically created instances.

This specification uses a different point of view : The creation methods are offered by the nodeServer and factory subsystems, while the operations of (or on) a dynamically created subsystem are specified with the actor subsystem.

2.8.8 Putting it together

The cooperation of the subsystems introduced up to now is depicted in figure 2 on page 11 in a most abstract view which shows four(4) fundamental kinds of interactions :

(0) An RBD speaking directly to the CAS, for requesting e.g. a `setVerdict()` or a `getGlobalValue()` service.
(1) An RBD requesting a `create()` service, which is passed from CAS via NodeServer to a Factory.
(2) An RBD doing some control operations on an actor.
(3) A producer sending a `value_event()` via QAS to an RBQ.

Figure 4 shows in detail, how the same communication situations are realized by the underlying trns subsystem.

Figure 5 shows a realistic situation, where a TM prepares a communication channel ($\longrightarrow$) from some producer to some protocoling unit ($\in$ RBQ) (for later use at TRun) by issuing `create()` requests and configuration messages ($\longrightarrow$).

2.8.9 Inheritance Relations between Actor, RBQ, producer, RBD and TM

So far we used a clean model of pairwise-disjoint sets of offered services, and limited sets of requested services for each single category of subsystems, as introduced with the concept of subsystems in section 2.8.

This clean model is somewhat disturbed by the fact that some of the most important flows of signals and control are implemented by *violating* this principle of disjointness. Instead there is some „inheritance-like“ inclusion relation, which is depicted in figure 3 on page 17 (together with those signal-flows).

The small server interface of actor(= `setParam()`, `getParam()` and `rtOperation()`) and its empty client interface are inherited and extended ...

- by RBQ, adding the *service* for receiving signal event values asynchronously.

- by producer, adding the primitive *client* behavior of generating `value_event()`s asynchronously.
- by RBD, adding all of the full and rich *client* behavior related to the application semantics, — mainly creating new subsystems, operating on and configuring of actors, and controlling the run-time signal flow. But also requests for arbitrary specialized control and meta-control services may be included[26] offered e.g. by trns, CAS, NodeServers or Factories.

Figure 3 on page 17 illustrates how the three „fundamental kinds of interaction“, mentioned in the mentioned in the pre-going section[27] are implemented using this inheritance-like mechanism:

Since RBQ, RBD and Producer all implement the actor services, these can be used by any RBD to create, configure and control all these different subsystems by the same interface, cf. figure 3, paths (1) and (2). This is accomplished by means of *genericity*.

A Producer subsystem may send a `value_event()` message to QAS; then QAS sends a `rtOperation(putQ())` message to all subscribed RBQs, see cf. figure 3, path (3).

Please notice that the whish to *react* on incoming signals in general, and the execution of a TTCN-3 ETS especially, require to close a „central processing cycle“ by connecting the incoming signals arriving at an RBQ to some active behavior in an RBD. This cycle is *not* closed in TUB-TCI, but is left to the *usage* of TCI. In case of TTCN-3 this cycle is closed by a `component` construct, which ties one subsystem of RBD category to on of RBQ. While this construct is of course also outside the scope of TCI, it is nevertheless graphically indicated in some of the figures 2 to 5.

- Last not least: the TM category must inherit most of the *client* behavior of RBD. This is necessary simply because this is the only way at all foreseen in TUB-TCI to become somehow „active“ for doing anything senseful, e.g. send `create()` requests to Factories and configuration changes to actors.[28].

2.8.10 trns = Transportation Layer

Two subsystems make up the lower layer of the architecture:

First there is the trns subsystem, which realizes a minimalistic version of a kind of „combined transportation and network layer“. For all *SRQS* and *replies* all subsystems must use the trns layer for information exchange.

Figure 6 shows the information flow between all subsystems described so far, and how the communication is piped through trns. The thick blue lines indicate the „logical“ interfaces defined in this paper.

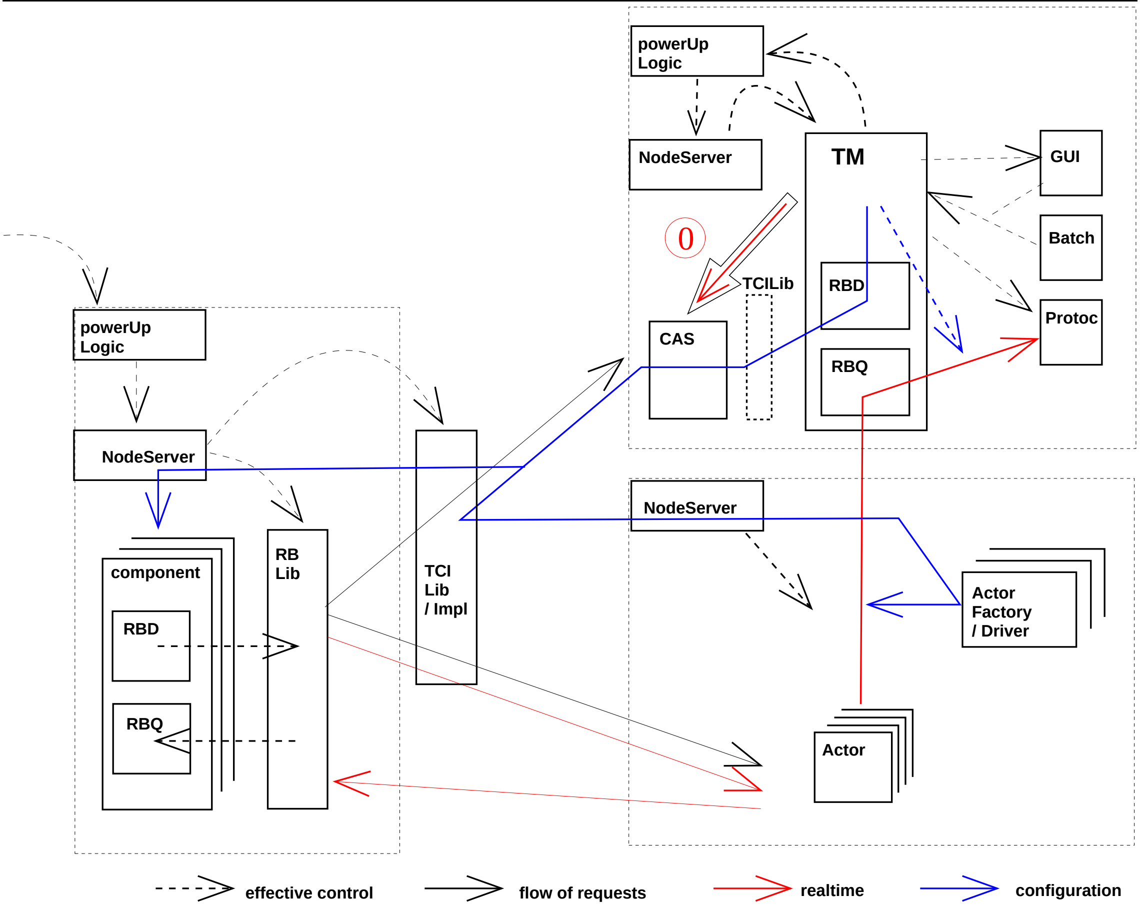
The thick green line indicates the „physical“ interface, i.e. the definition of the data format actually exchanged on the wire. This is ruled by a central (and only :-) specification :

[26] If the request of a `value_event()` service is permitted for an RBD (i.e. a kind of software-generated events), then RBD seems to inherit from Producer, and only indirectly from actor. This possibility is indicated graphically in figure 3.

[27] (1)=creation, (2)=control and operation of actors, (3)=run-time signal flow, depicted in figure 2 on page 11.

[28] On the server side there needs not to be a relation from TM to RBD. The services offered by TM are defined on their own. So the actor services for configuration control, status inquiry, deletion, etc. need not to be supported by an instance of TM, — mostly the TM-process itself is *not* running under the control of TCI.

Figure 5 A realistic Scenario, including a „Test Manager“ TM



All TUB-TCI information exchange is realized by the exchange of valid, ASCII encoded XML fragments.

Section 4 describes a canonical encoding rule, which translates all „Free Types“ appearing in the specification into XML `Element` definitions. All systems which claim to be compliant with TUB-TCI must support (at least one variant of) this XML encoding.

Since (1) complex *SRQS* containing complex configuration information should seldomly happen in TRun, but in the less time-critical TPrep, (2) since XML is an easy to decode, „almost binary“ data format, and (3) because processing speed will further increase in future, the overhead of using a „textual“ encoding for configuration purpose is more than balanced by the un-ambiguousness and easy traceability which will yield high profits, especially in complex hardware situations.

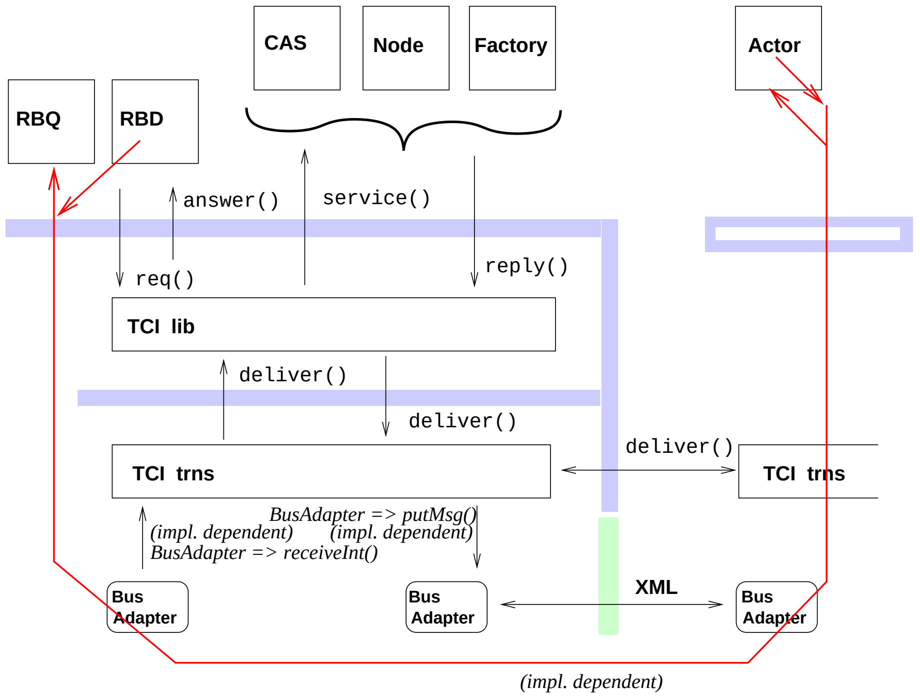
2.8.11 HsLink = High Speed Channels

For *real-time data flow* the situation can be different, – coding and decoding real-time value events into XML could turn out to be infeasible for performance reasons.

Therefore TUB-TCI provides a second transportation layer protocol, the HsLink (= High Speed Link). This subsystem allows to integrate **native, binary encoded** data channels into the setup.

Their encoding and semantics are outside the scope of TCI, but the run-time flow

Figure 6 Overview on message flows between Subsystems



control and configuration is completely defined and totally controlled by TUB-TCI.

In figure 6 these channels are indicated by the red arrows ($\longrightarrow$). TCI does *not* make any assumptions neither on the electrical, nor on the encoding techniques in which a HsLink is implemented. The TM is responsible for issuing the configuration requests correctly, i.e. only the TM needs all required information concerning encoding compatibility, supported bandwidths etc. All this is (still) out of scope of TUB-TCI, but section 5.2.2 will discuss some possible extensions.

2.8.12 BusAdpts = Bus Adapters

TUB-TCI does try to make least assumptions on the way of operation of the underlying „physical“ data transmission layer, — neither in case of the normal XML exchange via trns, nor in case of the high-speed channels controlled by HsLink.

The structures and behaviors of these lower communication layers are hidden behind the *façade* defined by the BusAdpt interface. Their behavior is only partly in scope of TUB-TCI, namely w.r.t this interface[29].

2.9 The Problem of „Architectural Information Leaks“

During the work on the design of TUB-TCI, we frequently came across one and the same architectural problem in rather different appearances:

This problem is given by the fact that a piece of information, hosted in one particular layer of the architecture, could be of great use if it was known in a function of a quite different layer. This normally happens for sake of *performance*, seldomly for sake of clarity.

The claim for a clean system design prohibits the use of such an information from a different layer. But if we do give the allowance of such „peeping“, e.g. because the clean

[29] Therefore this category of subsystems does not appear e.g. in table 2.

solution would be too expensive, we call this license an „architectural information leak“, – information „leaks“ from one layer into another, where it normally should never appear.

An example: the node topology should (naturally) be totally encapsulated in the trns subsystem and not visible to the higher subsystems, except (at the top!) the TM. But for to subscribe an RBQ for an Producer, we have to consider the topology explicitely, and it would be nice if we were allowed to „reuse“ the trns layer information.

Now as we have a formal specification of all layers, we can indeed make this kind of *licenca*, because the formulas can show *precisely*, where such maceration does happen, – in our text the locations of architectural information leaks are marked by $\boxed{\text{red boxes}}$.

2.10 Glossary of Central Notions

2.10.1 Cited Notions

test case	One single programmed definition of sequences of stimuli to, and their expected reactions from an SUT, cf. [ttc01]. .
test suite	A sequence of test cases.
(testing) sandbox	An execution area offered by a hardware node, where up-loaded code can be executed in a secure way with well-defined limited execution permissions.
ATS	Abstract Test Suite, the source text of a test procedure, given e.g. in TTCN-3. The notion „ATS“ is also used for the *semantics* of this program text, cf. [ttc01].
ETS	Executable Test Suite, the result of compiling an ATS, i.e. a collection of segments of executable code, cf. [ttc01].
SUT	System Under Test, cf. [ttc01].
PDU	Protocol Data Unit. One unit of information processed or transported atomically.
TTCN-3	A lately defined new programming language for imperative specification of test procedures, see [ttc01].
GCI	Generic Compiler/Interpreter Interface. The single-threaded run-time environment for (old) TTCN-2 etc. [GCI96].
IXIT	Implementation eXtra Information for Testing. Some additional parametrization of a test suite, not related to the ATS, but only to some special compilation, cf. [ttc01].
Module Parameter	Single global environmental constant for parametrizing a test case[30].
TM	Test Manager. An active software which controls test execution, either interactively or batch driven. It may include visualization, report generation etc., cf. [tci03].
SRQ, *SRQS*	Service Request, Service Requests.
`component`	A construction in the TTCN-3 language, partly comparable „active object“ or „thread“ or „co-routine“.

2.10.2 Definitions of Notions

TEns	„Test Ensemble“ A pool of computers, test devices, systems under test and auxiliary message processing devices, together with the busses connecting them. All these devices must be under the control of TCI.
Subsystem	is the name for „(active) objects“ in the TUB-TCI-architecture. The whole behavior of a TEns during TRun is *totally* defined by the parallel execution of

[30] This is apparently *not* what the reader might is used to!

all known subsystems.
Each subsystem is an instance of one category. Each category defines the set of services offered and requested by its instances.
In TUB-TCI there are nine categories of subsystems populating an the upper application level, namely : RBD, RBQ, Node, Factory, actor, CAS, TM and QAS.
Below, there are the trns and HsLink subsystems, realizing the underlying data-link and communication layer.

Client/Server	in context of this paper: the names of two rôles taken by two subsystems. One subsystem acting as „client“ starts the C/S relationship by generating a `request()` message, which reaches a second subsystem (acting as server) in form of a `service()` message. Any `reply()` message sent back by the server reaches the client (in case that it is expecting such an answer and waiting) encoded as an `answer()` message.
CAS	Central Access Server. One central subsystem which keeps all configuration information. It is addressable by every other subsystem using a single, fixed and predefined *NodeIdent*.
Actor	All dynamically created (logic) entities. They normally represent some internal hardware device, like a `Port`, a `Component`, a `Timer`, etc. While normally being treated as separately categories because of their special capabilities, all RBD and RBQ subsystems also offer all services defined with the actor category , as the only means for of being created, inquired, deleted and controlled.
actor class	Dynamically loaded description of a set of possible actors. Each living actor is assigned to exactly one *actor class*. The actor class defines the names, types and properties of the configuration parameters of its instances, and the set of run-time operations understood by its instances.
Producer	A sub-category of actor representing all devices which can *produce* events spontaneuosly and asynchronously, e.g. incoming ports, timers, switches, sensors etc.
RBD	„Run-Time Behavior Dynamic“ An active subsystem realizing the active aspects of an ETS. RBDs have a rich client-behavior, esp. w.r.t creating and controlling actors and communication channels.
RBQ	„Run-Time Behavior Queues“ A passive subsystem, serving as a drain for run-time signals. The only service it offers (beyond being an actor) is `rtOperation(putQ())`.
Node	Physical computing or testing device, operating independently from other nodes in the TEns.
NodeServer	Exactly one subsystem of the NodeServer category runs during a whole TSess on each physical node. It realizes the basic control of the physical device as well as the creation and deletion of actor subsystems.
NodeIdent	Node Identification. An arbitrarily chosen datum (e.g. text string), which uniquely identifies each distinct node in the TEns.
QAS	The central subsystem which manages the flow of run-time signals (push-channels and `value_events()`), which mostly are generated by producer and end in RBQ subsystems. Virtually there is one single ubiquitous QAS, – on implementation level there

	is one implementation on each node.
trns	The underlying transport layer. trns uses a uniform, XML-based message encoding which is part of this specification.
HsLink	High Speed communication channels. These use *hardware specific* encoding on a binary level. They must be allocated and configured in TPrep, and may be switched on and off in TRun. A HsLink can only be used between sources and drains which use the same encoding. The management of the different encodings is out of scope of TCI,see below section 5.2.2.
BusAdpt	A bus driver subsystem hosted on a distinct node and connected to one distinct bus. Onto this bus it can write out messages. This functionality is used by trns to write out a message to the trns subsystem hosted on a remote node. A busdriver also receives incoming messages asynchronously, and passes them to the trns subsystem hosted on its hosting node.
BusId	Arbitrary text string which uniquely identifies a BusAdpt w.r.t. its hosting node.
BusAdr	Arbitrarily structured value type. A BusAdr b identifies w.r.t. a given BusAdpt a uniquely one foreign node n, which must connected to the bus connected to a. Each `deliver()` message sent to a with b given as the target value, will be delivered to the trns subsystem of this foreign node n.
`CASNode`	The node hosting the `CAS`.
LUID	Limited Unique ID. A (numeric, e.g. 32 bit) Identifier which uniquely identifies one actor, – uniquely w.r.t. a certain TEns and a certain *Epoque*.
TAID	Transaction ID. A (numeric, e.g. 32 bit) Identifier which uniquely identifies all outgoing transactions/messages w.r.t. a given node and a certain *Epoque*.
TSess	An interval of real-time during which „test activities“ happen, e.g. test procedures are prepared, performed, analyzed etc. TUB-TCI makes only one assumptions on the structure of TSess, namely that they consist of a sequence of intervals, alternating between TPrep-type and TRun-type.
TPrep	An interval of real-time in which no test process is being executed, but the next execution of a test is being *prepared*.
TRun	An interval of real-time in which a test (test-case, test-suite) is being executed, i.e. code segments (or subsystems) determined by the ETS are currently being executed on the respective hosting nodes.
epoque	An interval of real-time between two system `reset`s. Some subsystems are only living (and their corresponding identifiers are only valid) during a certain epoque. One could distinguish *TAID*-epoques, *LUID*-epoques, *HsLink*-epoques, etc., but in the current TUB-TCI-design all these epoques have been unified.
`rtOperation()`	Run-Time Operation, a service which can be requested by an actor subsystem to perform some activity or change of configuration, even in TRun. The set of possible `rtOperation`s for a given actor is predefined and fixed by the definitions given by its actor class.

Table 1 Overview on Categories of Subsystems

- trns = Transportation Layer.
  Basic layer for each communication between all other subsystems and hardware nodes.
- HsLink = High Speed Link.
  Basic layer for fast, binary encoded and maybe proprietary communication between producer and RBQ or RBD and actor subsystems.
- TM = Test Manager.
  Only one single instance per TEns, doing the interactive or batch driven control of the overall testing process.
- CAS = Central Access Server.
  Single central instance for registration, look-up and lifetime control for all dynamically created subsystems.
- QAS = Queue Access Server.
  Conceptually one single (ubiquitous) instance for the management of run-time flow of signals and values.
- NodeServer.
  One subsystem per node, — allows to control the node's hardware resources and to create new actors on the node.
- Factory.
  One or more on each node, — each Factory is responsible for creating new actors of one single type (= „*actor class*“).
- Actor.
  Dynamically created subsystem, the behavior of which depends on its definition and usage, both outside the scope of TCI.
  The actor category can be called „virtual“, since instances can only be built from its subcategories, which are . . .
- Producer = signal producing active external device
  Active subsystem representing some hardware device, which spontaneuosly generates events.
- RBD = Run-Time Behavior / Dynamic.
  Active subsystem, compiled code from an ETS, e.g. active part of a TTCN-3 `component`-object.
- RBQ = Run-Time Behavior / Queue.
  Passive subsystem as drain for a signal stream, e.g. the passive part of a TTCN-3 `component`-object.

Table 2 Categories of Subsystems specified in TUB-TCI

Abbrev.	Name	Instances per TEns	specified herein?
TM	*T*est *M*aster	1	partially
CAS	*C*entral *A*ccess *S*erver	1	YES
QAS	*Q*ueue *A*ccess *S*erver	$1(n)$	YES
trns	TCI *Trans* portation Layer	$1(n)$	YES
NodeServer	*Node Server*	n (one per Node)	YES
Factory	*Factory* of Actors	$\Sigma_{x=N_1}^{N_n}\ C_x$	partially
RBD	*R*untime *B*ehavior – *D*ynamic	d	partially
RBQ	*R*untime *B*ehavior – *Q*ueues	d	partially
Actor	*Actor*	d	partially

n	=	Number of *N*odes in Test System.
d	=	*D*ynamically defined by Code and Compiler.
C_x	=	Number of supported Actor *C*lasses on node x.
$\Sigma_{x=N_1}^{N_n}\ C_x$	=	Sum of preceding numbers over all nodes in the TEns.
$1(n)$	=	Conceptually one single instance, but one physical implementation per node.

Table 3 Inheritance between Categories of Subsystems

	as Server =	as Client =
Actor	`setParam()` `getParam()` `rtOperation()`	{}
RBQ	= Actor ∪ {`putQ()`,...}	{}
RBD	= Actor	all kinds of creation and control of Actors
Producer	= Actor	{`value_event()`,...}
TM	{`decideNode()`}	⊂ RBD

Table 4 Offered Services

TM	1	called by CAS :	`getGlobalValue()`
			`decideNode()`
CAS	2	called by TM :	`CASreset()`
	3	called by RBD :	`getGlobalParameter()`
		Intercepted by trns $\longrightarrow$	`create()`
		Intercepted by trns $\longrightarrow$	`delete()`
			`setverdict()`
	4	called by trns :	`lookup()`
	5	called by Factory :	`registersubactors()`
			`deleted()`
			`desmudge()`
NodeServer	6	called by CAS :	`reset()`
			`loadRouting()`
			`sendHdwStateInfo()`
			`startsession()`
			`stopsession()`
			`DOcreate()`
			`DOdelete()`
			`HScreateinlink()`
			`HSregisteroutlink()`
Factory	7	called by NodeServer :	`DOcreate()`
			`DOdelete()`
Actor	8	called by RBD :	`setconfigparams()`
			`getconfigparams()`
			`rtOperation()`
RBQ	9	called by QAS :	`rtOperation ( putQ() )`
			`HSputQ()`
QAS	10	called by RBD :	`subscribe()`
			`unsubscribe()`
			`HSsubscribe()`
			`HSunsubscribe()`
	11	called by Producer	`value_event()`
			`HSvalue_event()`
trns		called by trns (= interface from/to *Bus Adapters*) :	
			`deliver()`
		called by *Client* (=all but RBQ) :	`req()`
		delivered to *Server* :	`service()`
		called by *Server* :	`reply()`
		delivered to *Client* :	`answer()`
back door interface (≈ *TCP*)		called by *client* :	`BDopen()`
		called by hosting node of *server* :	`BDregisterServer()`
		called by *client* and *server* :	`BDread()`
			`BDwrite()`
			`BDclose()`

3 Operational Semantics of TCI Subsystems — Messages and Behaviors

3.1 Mapping of the Meta-Models

This section describes the semantics of TUB-TCI as one large, monolithic transition relation.

The concepts of *subsystems* and of their client/server interactions, as introduced in the last section, (called „architectural model“ in this section) will *not* appear in this formal parts of the TUB-TCI specification explicitely.

Instead, the *execution model* is that of a finite set of parallel processes. One or more of these processes corresponds to a certain subsystem in the architectural model.

Each process (at any time) evaluates some (current) expression. These expressions are either terms of the „message“ data types defined in this section, or some terms of some „hosting“ programming language. Respectively, each evaluation of such an expression is either the application of one of the transition rules defined in the following sections, or an evaluation step of the hosting language, outside the scope of this specification.

The execution of a *SRQ*-style interaction between two subsystems in the architectural model, is here modeled as the evaluation of the corresponding expressions in the contexts of the two processes corresponding to the involved subsystems.

Each „service“ offered by a category of subsystems A and called from instances of some other category B in the architectural model (as described in the pre-going section) is represented by a corresponding data type with name $\mathit{SRQS\_from\_}\langle B\rangle\mathit{\_to\_}\langle A\rangle$, which will be defined in the subsections called „**Messages**“ of the following sections.

But these different type definitions are pure conceptual, — effectively there is only one single definition

TCI_Message == *(Disjoint Union of all „SRQS_from_to_“ data type definitions)*

and only instances of this data type are exchanged in the implementation.

So the involved subsystems are given only implicitly, but uniquely: The names of the meant services are uniquely represented by the constructors of *TCI_Message*. Furthermore, given by table 4, there is an immediate and unique identification of the category of subsystems offering the corresponding service. In case of more than one instance of this category (w.r.t. the executing hardware node), the identities of the subsystems are given explicitely, encoded in the *user data* of the *SRQS*. So the means, mechanisms and data formats for addressing individual subsystems can be rather different for different categories (e.g. actors by *LUID*s, NodeServers by *NodeIdent*s).

The names of the categories involved subsystems *do* appear in the *titles* of the sections, mostly in the form $A \overset{n}{\Longrightarrow} \boxed{B}$. Here n is the number of the group of services as given in table 4, a subsystem of category A is requesting from some B, and the subsystem in the $\boxed{\text{Box}}$ is the one the behavior of which described in the section.

As notation format we use a small subset of the specification language Z (cf. [Spi92]), which is enhanced by some „syntactic sugar“ for ease of reading.

3.2 Aspects of a TCI-System Covered and Not Covered by TUB-TCI

The TUB-TCI-model is only intended to model *one aspect* of the architecture, which could be called „functional“.

Since, — as already mentioned in the introduction above, — its main purpose is to serve as a common basis for implementation, formal techniques and *human discussion*, the formalization has only been driven far enough to present an „upper limes“ of permitted behaviors.

Many other aspects are left to the „common sense“ of the reader and to further discussion. Additional formalization of other well-defined areas and aspects may and shall happen as soon as a significant profit in productivity can be expected. This strategy could be called „formalization on demand“.

Some of these aspects are given by those which are *not* covered by (the current status of) TUB-TCI, for instance:

- Astonishingly there is no positive specification of a minimal behavior for any TUB-TCI-implementation! E.g. an implementation of a NodeServer is only requested *informally* to create an implementing object when receiving a corresponding `DOcreate()` request, if this creation would be possible w.r.t. the node's resources.
  It is *not* specified formally, that eventually some positive answer must come at all: an implementation always refusing to create any actor subsystems would be a valid TUB-TCI-implementation w.r.t. this specification.
- All questions of timing, scheduling, process algebra etc. (1) must at the moment be resolved by common sense and discussed by semi-formal, verbal specification, (2) may be subject of further research (or subject to the application of well-known results from other areas of research), and (3) their answers will probably widely vary with the concrete limitations of the concrete hardware, so they partially cannot be answered prior to practical experiments.
  TUB-TCI is safe and free of dead-locks just because of the simplicity and cycle-freeness of its atomic operations and because of the simple time-out mechanism on the lowest trns layer. But please note that there are no further „temporal“ rules or results, e.g. concerning *fairness* of execution or timing requirements.

3.3 Semantics of the Transformation Relation

The following sections specify single transition steps. Most of them involve only one process (process context), others involve two of those.

The overall transformation relation is (theoretically) given by the disjunction of all possible transitions given by these transition steps.

This transformation relation serves as a direct denotation for the set of all *maximally permitted* behaviors of all valid TCI implementations by reading it as follows :

1. For all expressions the constructor of which is *not* defined in this paper, there must be some matching transformation rule(s) in the actual language binding.
2. If no such transformation can be found, this is a „language performance error“.
3. If there are such transformations, one of them must be chosen (externally !) and applied to the current evaluation state to generate the new evaluation state.
4. For all expressions the constructor of which *is* defined in this paper, only the transformations defined in this paper are applicable.
5. If there are such transformations, one of them is chosen arbitrarily and applied to the current evaluation state to generate the new evaluation state.
6. If no such transformation exist, a „TCI system error“ is given, i.e. the transformation rule[31]

$$\mathcal{E}[[\mathcal{K}, K, \zeta_1\ ]] \ \rightsquigarrow\ \mathcal{E}[[\mathcal{K}, K, \texttt{TCIERROR}\ (\ \zeta_1\ )\ ]]$$

[31] The notation is explained in the next section.

it the *general fallback rule* for all *TCI_Message*s for which no appropriate transformation can be found.

7. Such an error may be silently ignored, by further reduction according

$$\mathcal{E}[[\mathcal{K}, K, \texttt{TCIERROR}\ (\ \zeta_1\ )\ ]] \ \leadsto\ \mathcal{E}[[\mathcal{K}, K, \varepsilon\ ]]$$

8. Otherwise it has to be signaled to some „responsible“ subsystem, e.g. by

$$\begin{aligned} & \mathcal{E}[[\mathcal{K}, K, \texttt{TCIERROR}\ (\ \zeta_1\ )\ ]] \\ \leadsto\ & \mathcal{E}[[\mathcal{K}, K, \texttt{deliver}(\texttt{\$CAS}, K.\mathit{ownIdent}, -9999, \texttt{ERR}, \texttt{true}, \\ & \qquad\qquad \texttt{TCIerror}(K.\mathit{ownIdent}, K.\mathit{localTime}, \pi, \zeta_1)\ )]] \end{aligned}$$

where π is some (opaque) information about the identity and inner state of the thread which was trying to execute the offending expression ζ_1.

3.4 Basic Model of Execution and Notation

3.4.1 Defining Transitions of Execution States

The model of execution is written giving one *expression* ready for execution, which is replaced by another expression. The notation is

$$\mathcal{E}[[\mathcal{K}, K, \zeta_1\ ]] \ \leadsto\ \mathcal{E}[[\mathcal{K}, K, \zeta_2\ ]]$$

where ζ_1 stands for some expression which is reduced to another expression ζ_2. K stands for the *state* of the node, on which the process evaluating expression ζ_1 takes place. $\mathcal{E}$ stands for „evaluation“, $\mathcal{K}$ stands for the sum of the states of all other nodes. The latter is mentioned just symbolically to indicate that the state of all other nodes not mentioned in the transition formula is *unaffected* by this transition.

The rare cases where more than one node is involved in a transition, naturally happens only between two trns subsystems. In this case the nodes and transitions are separated by comma, like

$$\mathcal{E}[[\mathcal{K}, K, \zeta_1, L, \zeta_1'\ ]] \ \leadsto\ \mathcal{E}[[\mathcal{K}, K, \zeta_2, L, \zeta_2'\ ]]$$

The notation

$$\mathcal{E}[[\mathcal{K}, K, \zeta_1 \mid \kappa\ ]] \ \leadsto\ \mathcal{E}[[\mathcal{K}, K, \zeta_2\ ]]$$

imposes some *conditions* on the state K, which have to be fulfilled to enable the transition to take place. The construct κ is a conjunction of boolean predicates using the variables of K and variables contained in ζ_1 and ζ_2.

All notations listed so far imply that the state K is *not* altered by the transition.

The notation

$$\mathcal{E}[[\mathcal{K}, \Delta K, \zeta_1 \mid \kappa\ ]] \ \leadsto\ \mathcal{E}[[\mathcal{K}, K, \zeta_2\ ]]$$

describes a state transition of the node state K. Here κ is a conjunction of boolean predicates like above, additionally using the variables of K in a *primed* form. These primed variables stand for the values of the variable after the transition took place. It is part of the semantics, that *all* variables of K *not* appearing in a primed form are *left unchanged*[32].

[32] For sake of brevity the same syntactic sugar is defined with the mapping construct: An assignment like $f'(3) = 4$ leaves all other positions of f unchanged, and thus is an abbreviation for $f' = f \oplus \{3 \mapsto 4\}$.

3.4.2 Language Binding

We assume that for each node or subsystem there is a set of arbitrary expressions and corresponding reducing transitions which make up the „programming language“, in which the behavior of a subsystem can be described. This set of „language expressions“ has to be totally *disjoint* from all expressions defined in this specification. The corresponding transitions are *not* described in this specification. But it is part of the specification that all expressions defined herein can *only* be reduced by transitions from this specification.

To model those library calls of the language binding which return results, we assume that there are some evaluation rules in the hosting language which support „nesting“ of expressions. This can informally be described as

$$\frac{\mathcal{E}[[\mathcal{K}, \Delta K, \zeta_I \mid \kappa\ ]] \ \rightsquigarrow \mathcal{E}[[\mathcal{K}, K_2, x\ ]]}{\mathcal{E}[[\mathcal{K}, K, \zeta_O\text{"("}\zeta_I\text{")"}\ ]] \ \rightsquigarrow\ \mathcal{E}[[\mathcal{K}, K_2, \zeta_O\text{"("}x\text{")"}]]}$$

where ζ_I represents some expression defined in TUB-TCI, ζ_O some expression from the language into which TCI calls are embedded, and x the result of evaluating the TCI call.

3.4.3 Thread and Execution Model

All application level transformations of TCI are specified assuming a multi-thread ensemble with *independent* performance of all threads. So the transformation steps specifying the evaluation of *TCI_Message*s need only be given for the (one and only) executing thread locally.

The *NodeState* „K“ is shared among all threads running on a distinct node, and each transformation step defined herein must be performed as a whole *atomically* w.r.t. all reads and writes of K and K'.

On the lower level of trns we need parallelism and sequentialization in the specification *explicitely*, – due to the fact that we have to treat the *rendez-vous* of two nodes communicating by a bus and two BusAdapters. For this purpose we introduce the notation

$$\mathcal{E}[[\mathcal{K}, K, \zeta_1\ ]] \ \rightsquigarrow\ \mathcal{E}[[\mathcal{K}, K, \zeta_2 \mathbin{\|\!\!+} \zeta_3\ ]]$$

to denote the „spawning“ of a new thread on the same node, i.e. one(1) thread performing ζ_1 is transformed into two(2) threads. Both threads inherit the same „language context“ implicitly. (The *NodeState* called K is global and unique anyway !)

The notation

$$\mathcal{E}[[\mathcal{K}, K, \zeta_1 \parallel \zeta_2]] \ \rightsquigarrow\ \mathcal{E}[[\mathcal{K}, K, \zeta_3\ ]]$$

denotates the dual case, when two(2) threads evaluation one expression each „at the same time“ are joined into and continued by only one(1) thread. This operator is *not* commutative, because the surviving thread inherits the language context of the first of the two incoming threads[33].

If both K and L are of type *NodeState*, then the notation

$$\mathcal{E}[[\mathcal{K}, K, \zeta_1 \parallel L, \zeta_2]] \ \rightsquigarrow\ \mathcal{E}[[\mathcal{K}, K, \zeta_3 \parallel L, \zeta_4]]\ ]]$$

denotates a transformation step which happens on two(2) nodes synchronously.

[33] **Please notice** that this „invisible language context“ can easily be eliminated by integrating its data into the expression ζ_1, i.e. the expression which is „waiting“ with the process for the „join“ $\parallel \zeta_2$ to happen. This would be a much cleaner design of the meta-model, but there had been no time left for doing this correction `:-(`

This construct will only needed when specifying the „electrical“ communication interaction of two nodes, and appears only in two places: Once in the `trns` interaction with its *BusAdapter*s (c.f. section 3.6.7), and in the High Speed Channel communication (cf. 3.13.8).

3.4.4 Extensions to the Z Notation

Incremental Declaration of Schemata : A notation like

```
── Example ─────────────────────────────
│ ——— some definitions ———
│ . . .
└───────────────────────────────────────
```

indicates, that the declaration of the schema type will be continued somewhere below in the text.

A notation like

```
── Example ─────────────────────────────
│ . . .
│ ——— some definitions ———
└───────────────────────────────────────
```

indicates, that this is a continuation of a schema definition which had been started somewhere above in the text.

Common Declaration of two FDTs : Whenever we define an FDT which shall be used as *request* message item, we can declare the FDT realizing the corresponding *reply* message at the same place, using the notation

$$
\begin{array}{lll}
\mathit{RequestType} ::= & \texttt{request} \langle\!\langle \ldots \rangle\!\rangle & \\
 & \quad \Rrightarrow \texttt{reply} \langle\!\langle \ldots \rangle\!\rangle & \\
 & \quad \Rrightarrow \texttt{anotherreply} \langle\!\langle \ldots \rangle\!\rangle & \\
 & | \ \texttt{anotherrequest} \langle\!\langle \ldots \rangle\!\rangle & \\
 & \quad \Rrightarrow \texttt{futherreply} \langle\!\langle \ldots \rangle\!\rangle &
\end{array}
$$

Since „at last“ all definitions of FDTs which can serve as messages are „thrown together“ into one lager message type definition, and therefore the constructors are chosen pairwise-disjoint anyway, this „anonymous“ declaration of FDT-constructors does not impose any problems.

Denotations of FDTs using schemas : PDUs and auxiliary state variables are written as *free data types*. The original Z notation is enhanced by the „syntactic sugar“ rule that the definition of a constructor using a schema type as argument, like

$$\mathit{data} ::= \texttt{variant} \langle\!\langle [n_1 : t_1, \ldots n_k : t_k] \rangle\!\rangle \ | \ \ldots$$

can be treated like

$$\mathit{data} ::= \texttt{variant} \langle\!\langle t_1 \times \ldots \times t_k \rangle\!\rangle \ | \ \ldots$$

when *denotating a single value.*

So we can write

$$\texttt{variant}\ (v_1, \ldots, v_n)$$

to denotate an element of the free type *data*. The construction of the necessary binding value, i.e. the value of the embedded schema type, is done automatically.

Dependent types : Dependent typing is written using a „meta assignment“ construction like e.g.

$$\mathit{SRQS\_from\_RBD\_to\_CAS}\ [\ \boxed{a =}\mathit{ActorClass}, \mathit{ActorParams}\,[a]\ ]\ ::=\ \ldots$$

which indicates informally, that the second generic parameter is parameterized with the value of the first parameter.

Pattern Matching : We want to write decomposition of a schema simply as an equation between a value of this schema type and a *pattern*.

The special symbol „_“ means that this field in the schema is irrelevant and may have any value and will not be bound to a variable.

Any notation like $B_0\Box\ldots\Box B_n$, where $\Box$ is some operator, is meant as abbreviation for repeated application of this operator to $n+1$ arguments.

Calling of „Drivers“ : A notation using the symbol $\boxed{\texttt{->}}$ (either instead of a colon „:“ or just instead of whitespace!) indicates the declaration or the application of a function which will be implemented by a kind of „driver method“.

These methods (1) are „physical“ interfaces which have to be implemented by all drivers which implement the schema, and (2) may have side-effects in the internal state of the realizing driver, which are out of scope of the TCI specification.

There a two variants of declarations led in by $\boxed{\texttt{->}}$: Either an identifier is related to a function signature, or it is declared to be a constructor. In the first case the function can be used by some transition rule for calculating some value. On the level of specification these calls are treated as pure function calls, – in contrast the *implementation* of the call is out of scope of TCI, as it (1) can read and write some internal state variables in the „driver object“, and (2) can calculate its return values by „mystic means“ using its internal state.

In the latter case expressions can be built. The reduction rules of these expressions *are* part of the TCIspecification.

Reflection : Since we use (or „abuse“) Z schema definitions as declarations for the isomorphic data types on object level, we need some reflective devices, especially (1) to store schema types in variables, (2) to lift schema types to functions from Ident to schema, (3) etc.

3.5 Basic and Auxiliary Types

(0)

$$\begin{array}{|l} \mathit{Ident} \\ \mathit{BOOL}\ ::=\ \texttt{true}\ |\ \texttt{false} \\ \mathit{TextString} \end{array}$$

(1)

| *Time* |
Duration
continuous *Time*
continuous *Duration*

3.6 Transportation Layer

3.6.1 Principles of Operation (Non-Normative)

Guided by the goals of reliability and ease of implementation, the transportation layer trns is designed as primitive as possible.

Its internal and horizontal PDU is `deliver()`. The interface messages from/to application layer are `req()`/`answer()` on client side and `service()`/`reply()` on server side (see figure 6).

The lowest level transportation layer works „IP-like“, i.e. unconfirmed. The application layer above can make use of the message transmission functionality offered by trns either „UDP-like“, i.e. unconfirmed, or in a „single message and single response“ discipline.

As mentioned above, an interaction on the (conceptual) application level occurs between two subsystems, one taking the client rôle and the other the server rôle. The client sends one(1) single request message to the server, — the server may send one(1) single reply message back.

Additionally there is the mechanism of **broadcasting** a message to more than one node, e.g. sending an *SRQ* to the corresponding servers, hosted on a defined set of nodes in the TEns. For this sake the routing information loaded on each node contains the definition of *broadcast groups*, which can be used as a target address instead of *NodeIdent*.

The corresponding reply mechanism allows the client of the broadcast service to wait for a whole *set* of nodes to reply, – either all nodes reply in the given time interval, or a timeout occurs.

This broadcasting mechanism for *SRQS* is primely intended to be used by CAS and TM to emit service requests which shall reach different NodeServers with least timing screw.

On the *concept level* of the application layer the „servers“ in TUB-TCI are the active subsystems of the categories RBD, RBQ, CAS, QAS, NodeServer and Producer, offering the services described in the pre-going section, and listed e.g. in table 4.

On this (lower) transport level a „server“ is identified by attributing each `request()` expression with (1) the *NodeIdent* of the node on which the server is hosted, – and (2) a request message according to the free data type *TCI_Message*.

TCI_Message is the sum of all single TCI application level request message definitions spread over the next sections of this document.

So here servers are identified by the combination of the node, as given explicitely, and the *kind of constructor* of the request message, i.e. the „subtype“ of *TCI_Message*, to which the request message belongs. There is a unique mapping from these types to one category of subsystems which offers the appropriate service. If there are more than one instances of this category on the executing node, then the target instance is identified by *user-level* data contained in the *SRQ*, the coding and meaning of which differs with the messages.

3.6.2 Selection of Confirmation Mode, Timeout and Broadcasting

Each `request()` sent by a client must be attributed with a further value, namely (3) the timeout indication.

Either the value special `$noresp` is given. In this case the request is unconfirmed, the client continues its thread without waiting, and any reply generated by the server's execution is discarded by the `trns` subsystem on the server's node. In this case, on the client side there is no information at all about the further processing of the message.

Alternatively a `wait`(d) can be given, with d being any valid duration value. In this case the thread of the client is suspended and will be reactivated to evaluate the result of the call. This is either an `answer()` message received from the server, or the time out signal `answer($TIMEOUT)`, generated by the local `trns` subsystem.

The fact whether the client requests a confirmation or not, influences the thread scheduling on the client's side immediately, since the client thread will be suspended, or not.

Additionally this fact is coded into the message sent to the server. If a server provides a reply, it does so on every call. It is the job of the `trns` subsystem on the server's node to discard this reply message in case that no confirmation is requested by the client, thereby avoiding unnecessary traffic.

Again for sake of reliability, there is *only one single* source of timeout, namely on the `trns` subsystem on the client side (i.e. the side originally generating the transaction id of the *SRQ*).

3.6.3 *TAID*s

Central concept is the *TransactionId* *TAID*. *TAID*s are assigned to transactions autarkicly by the node which hosts the client of the *SRQ*. Given a certain *taid-epoque* and the identity of the node, which originally generated the *TAID*, each *TAID* uniquely identifies a certain *SRQ* .

*TAID*s are used to link each `reply()` message to the corresponding `request()`.

3.6.4 Messages

(2)

$$MsgType ::= \texttt{REQ} \mid \texttt{REQNW} \mid \texttt{REPLY} \mid \texttt{BC} \mid \texttt{ERR}$$

(3)

$$\mid TAID$$

(4)

$$\mid NodeIdent$$

(5)

$$\mid NodeIdent\_X \;==\; NodeIdent \cup \{\texttt{\$nodeNotInitialized}\}$$

(6)

$$Msg\_G\,[\,\_\_Message\_\_\,] ::= \mathtt{deliver}\langle\!\langle\,[\; \begin{array}{l} target : NodeIdent \cup BCgroup \cup \{\mathtt{\$CAS}\} \\ origin : NodeIdent \\ taid : TAID \\ msgtype : MsgType \\ noreply : BOOL \\ msg : \_\_Message\_\_ \end{array} \;]\rangle\!\rangle$$

// This is an example for the principle of „redundant encoding“ (cf. section 2.1), since the message component „noreply“ could be concluded from the value of *msgtype*.

(7)

$$Msg == Msg\_G\,[\,TCI\_Message\,]$$

(8)

$$\begin{array}{rll} RqTiming ::= & \mathtt{\$noresp} \\ | & \mathtt{wait}\langle\!\langle\, Duration \,\rangle\!\rangle \\ | & \mathtt{waitset}\langle\!\langle\, Duration, \mathbb{P}\, NodeIdent \,\rangle\!\rangle \end{array}$$

(9)

$$\begin{array}{rll} ReplState ::= & \mathtt{timeout}\langle\!\langle\, Time \times \mathbb{P}\, NodeIdent \,\rangle\!\rangle \\ | & \mathtt{inf} \\ | & \mathtt{timeoutsignaled} \\ | & \mathtt{replydone} \end{array}$$

(10)

$$|\; BCgroup == \mathbb{N}$$

(11)

$$
\begin{array}{lll}
\mathit{UserRq\_G}\ [\ \_\_\mathit{Message}\_\_\ ] ::= & \mathtt{req}\ \langle\!\langle\ [& n : \mathit{NodeIdent} \\
& & c : \mathit{RqTiming} \\
& & m : \_\_\mathit{Message}\_\_ \\
& &]\rangle\!\rangle \\
& \mid\ \mathtt{broadcast}\ \langle\!\langle [& b : \mathit{BCgroup} \\
& & i : \mathit{TAID} \\
& & m : \_\_\mathit{Message}\_\_ \\
& &]\rangle\!\rangle \\
& \mid\ \mathtt{service}\ \langle\!\langle [& o :: i : \mathit{NodeIdent} \times \mathit{TAID} \\
& & m : \_\_\mathit{Message}\_\_ \\
& &]\rangle\!\rangle \\
& \mid\ \mathtt{reply}\ \langle\!\langle [& o :: i : \mathit{NodeIdent} \times \mathit{TAID} \\
& & m : \_\_\mathit{Message}\_\_ \\
& &]\rangle\!\rangle \\
& \mid\ \mathtt{answer}\ \langle\!\langle [\ m : \_\_\mathit{Message}\_\_ \cup \{\mathtt{\$TIMEOUT}\}\ ]\rangle\!\rangle & \\
&]\rangle\!\rangle &
\end{array}
$$

(12)

$$
\begin{array}{lll}
\mathit{BusMsg\_G}[m] ::= & \mathtt{busmsg}\ \langle\!\langle [\ \mathit{next} & : \mathit{NodeIdent} \\
& \mathit{last} & : \mathit{NodeIdent} \\
& \mathit{data} & : \mathit{Msg\_G}[m]\ ]\rangle\!\rangle
\end{array}
$$

// This *redundant encoding* (cf. above 2.1) is for reliability and diagnosis only !

(13)

$$\mathit{UserRq}\ ==\ \mathit{UserRq\_G}\ [\ \mathit{TCI\_Message}\ ]$$

3.6.5 State Space

(14)

$[\ \mathit{Bus}$

$[\ \mathit{BusAdr}\ [\mathit{Bus}]$

(15)

BusAdapterIn

// These values are *hard wired* into the implementation of the Driver

$\mathit{bus} : \mathit{Bus}$

$\mathit{busAdr} : \mathit{BusAdr}$

$\boxed{\texttt{->}}$ watchBus

$\boxed{\texttt{->}}$ incomingMessage (BusMsg)

(16)

BusAdapterOut

// These values are *hard wired* into the implementation of the Driver
$bus : Bus$
$busAdr : BusAdr$

$\boxed{\texttt{->}}$ putMsgToBus $(BusAdr, BusMsg)$

(17)

$BusAdapter \;==\; BusAdapterIn \;\cup\; BusAdapterOut$

(18)

NodeState

// These values are *hard wired* into the implementation of TCI
// running on a given node :
$adapters : \mathbb{P}\; BusAdapter$
$localtime : Time$
$bootListeners : \mathbb{P}\, BusAdapter$

$bootListeners \;\subset\; BusAdapterIn \cap adapters$
...

(19)

$MapEntry ::= \texttt{drain}\;\langle\!\langle Adapter \;\times BusAdr \;\times\; BOOL\;\rangle\!\rangle$
$\quad\quad\quad\quad\;\; |\;\; \texttt{hop}\;\langle\!\langle NodeIdent\rangle\!\rangle$

// **Please notice** that a *BusAdr* needs *not* to be the same when different output bus adapters (normally, but not necessarily from different nodes) address the same node on the same bus !
(20)

RoutingTable

$CASident \quad : NodeIdent$
$gate \quad : NodeIdent \;\twoheadrightarrow\!\!\!\!\!\!+\;\; MapEntry$
$broadcasts \quad : BcGroup \;\twoheadrightarrow\!\!\!\!\!\!+\;\; \mathbb{P}\;\; NodeIdent$

(21)

NodeState

. . .
// These values are assigned *dynamically* by PowerOnReset and CAS :
$ownIdent : NodeIdent\_X$
$routingTab : RoutingTable$
. . .

(22)

NodeState

. . .
// These values are assigned *dynamically* when TCI-trns is running :
// Client's site:
$ta : TAID \twoheadrightarrow ReplState$ // a concrete implemention can/will store the whole process context
// here for all executions of `waitresp()`.

// Server's site:
$suppresreply : NodeIdent \times TAID \twoheadrightarrow BOOL$
. . .

3.6.6 *client*$\Longrightarrow\boxed{\mathsf{trns}}\Longrightarrow$*server* and *server*$\Longrightarrow\boxed{\mathsf{trns}}\Longrightarrow$*client*

$$\begin{array}{lll} k_K & == & K.ownIdent \\ k_{\mathcal{C}} & == & \mathcal{C}.ownIdent \\ M & \in & \_Message\_\_ \ \setminus \ \mathsf{dom}\ \mathtt{reset()} \\ B & \in & BcGroups \\ T & \in & BOOL \end{array}$$

(23) // Client does initial non-broadcast request, – unconfirmed and confirmed :

$$\begin{array}{ll} & \mathcal{E}\,[[\mathcal{K}, K, \mathtt{req}(n, \mathtt{\$noresp}, M)\ ]] \\ \leadsto & \mathcal{E}[[\mathcal{K}, \Delta K, \varepsilon \ \Vert\!\!+ \ \mathtt{deliver}(n, k_K, j, \left\{\begin{smallmatrix}\mathtt{ERR}\\ \mathtt{REQNW}\end{smallmatrix}\right\}, \mathtt{true}, M) \\ & \qquad\qquad \mid \quad j \notin \ \mathsf{dom}\ K.ta \\ & \qquad\qquad \wedge \quad K'.ta \ = \ K.ta \cup \{ j \ \mapsto \ \mathtt{inf} \}\ ]] \end{array}$$

(24)

$$\begin{array}{ll} & \mathcal{E}\,[[\mathcal{K}, K, \mathtt{req}(n, \mathtt{wait}(t), M)\ ]] \\ \leadsto & \mathcal{E}[[\mathcal{K}, \Delta K, \mathtt{waitresp}(j) \ \Vert\!\!+ \ \mathtt{deliver}(n, k_K, j, \mathtt{REQ}, \mathtt{false}, M) \\ & \qquad\qquad \mid \quad j \notin \ \mathsf{dom}\ K.ta \\ & \qquad\qquad \wedge \quad K'.ta \ = \ K.ta \cup \{ j \ \mapsto \ \mathtt{timeout}(c + K.localtime, \{k_K\}\ )\}\ ]] \end{array}$$

(25) // Client does initial broadcast request, – unconfirmed and confirmed :

$$\begin{array}{ll} & \mathcal{E}\,[[\mathcal{K}, K, \mathtt{broadcast}(B, \mathtt{\$noresp}, M)\ ]] \\ \leadsto & \mathcal{E}[[\mathcal{K}, K, \varepsilon \ \Vert\!\!+ \ \mathtt{deliver}(B, k_K, j, \mathtt{BC}, \mathtt{true}, M) \\ & \qquad\qquad \mid \quad j \notin \ \mathsf{dom}\ K.ta \\ & \qquad\qquad \wedge \quad K'.ta \ = \ K.ta \cup \{ j \ \mapsto \ \mathtt{inf} \}\ ]] \end{array}$$

(26)

$$\begin{array}{ll} & \mathcal{E}\,[[\mathcal{K}, K, \mathtt{broadcast}(B, \mathtt{waitset}(t, S), M)\ ]] \\ \leadsto & \mathcal{E}[[\mathcal{K}, \Delta K, \mathtt{waitresp}(j) \ \Vert\!\!+ \ \mathtt{deliver}(n, k_K, j, \mathtt{REQ}, \mathtt{false}, M) \\ & \qquad\qquad \mid \quad j \notin \ \mathsf{dom}\ K.ta \\ & \qquad\qquad \wedge \quad K'.ta \ = \ K.ta \cup \{ j \ \mapsto \ \mathtt{timeout}(c + K.localtime, S\ )\}\ ]] \end{array}$$

(27) // Server procedure called on server node :

$$\begin{array}{ll} & \mathcal{E}\,[[\mathcal{K}, K, \mathtt{deliver}(k_K, o, i, \mathtt{REQNW}, \mathtt{true}, m)\ ]] \\ \leadsto & \mathcal{E}[[\mathcal{K}, \Delta K, \mathtt{service}(o\!::\!i, m) \ \mid \ K'.suppresreply(o, i) = \mathtt{true}\ ]] \end{array}$$

(28)

$$\begin{array}{ll} & \mathcal{E}\,[[\mathcal{K}, K, \mathtt{deliver}(k_K, o, i, \mathtt{REQ}, \mathtt{false}, m)\ ]] \\ \leadsto & \mathcal{E}[[\mathcal{K}, \Delta K, \mathtt{service}(o\!::\!i, m) \ \mid \ K'.suppresreply(o, i) = \mathtt{false}\ ]] \end{array}$$

(29)

$$\begin{array}{ll}
 & \mathcal{E}[[\mathcal{K}, \Delta K, \mathtt{deliver}(B, o, i, \mathtt{BC}, T, m) \\
 & \qquad\qquad \mid \quad \{B_0, \ldots, B_b\} \;=\; K.\mathit{routing}.\mathit{broadcasts}(B) \setminus \{K.\mathit{ownIdent}\} \\
 & \qquad\qquad \wedge \quad K.\mathit{ownIdent} \in K.\mathit{routing}.\mathit{broadcasts}(B) \\
 & \qquad\qquad \wedge \quad K'.\mathit{suppresreply}(o, i) = T \;\;]] \\
\leadsto & \mathcal{E}[[\mathcal{K}, K, \qquad \mathtt{service}(o :: i, m) \\
 & \qquad\qquad ||\!\!\vdash \quad \mathtt{deliver}(B_0, o, i, \mathtt{BC}, T, m) \\
 & \qquad\qquad ||\!\!\vdash \quad \ldots \\
 & \qquad\qquad ||\!\!\vdash \quad \mathtt{deliver}(B_b, o, i, \mathtt{BC}, T, m) \;\;]]
\end{array}$$

(30)

$$\begin{array}{ll}
 & \mathcal{E}[[\mathcal{K}, K, \mathtt{deliver}(B, o, i, \mathtt{BC}, T, m) \\
 & \qquad\qquad \mid \quad \{B_0, \ldots, B_b\} \;=\; K.\mathit{routing}.\mathit{broadcasts}(B) \\
 & \qquad\qquad \wedge \quad K.\mathit{ownIdent} \notin K.\mathit{routing}.\mathit{broadcasts}(B) \;\;]] \\
\leadsto & \mathcal{E}[[\mathcal{K}, K, \qquad \mathtt{deliver}(B_0, o, i, \mathtt{BC}, T, m) \\
 & \qquad\qquad ||\!\!\vdash \quad \ldots \\
 & \qquad\qquad ||\!\!\vdash \quad \mathtt{deliver}(B_b, o, i, \mathtt{BC}, T, m) \;\;]]
\end{array}$$

(31)

$$\begin{array}{ll}
 & \mathcal{E}\,[[\mathcal{K}, \mathcal{C}, \mathtt{deliver}(k_{\mathcal{C}}, o, i, \mathtt{ERR}, \_, m) \;]] \\
\leadsto & \mathcal{E}[[\mathcal{K}, \Delta\mathcal{C},
\end{array}$$

// a TCI system/consistency error occurred somewhere in the TEns.

// CAS and TM may take adequate means.

$]]$

(32) // Server sends back reply message :

$$\mathcal{E}\llbracket \mathcal{K}, K, \texttt{reply}(o :: i, m) \;\mid\; K.\mathit{suppresreply}(o, i) = \texttt{true} \;\rrbracket$$
$$\leadsto \;\mathcal{E}\,\llbracket \mathcal{K}, K, \varepsilon \,\rrbracket$$

(33)

$$\mathcal{E}\llbracket \mathcal{K}, K, \texttt{reply}(o :: i, m) \;\mid\; K.\mathit{suppresreply}(o, i) = \texttt{false} \;\rrbracket$$
$$\leadsto \;\mathcal{E}\,\llbracket \mathcal{K}, K, \texttt{deliver}(o, k_K, i, \texttt{REPLY,true}, m) \;\rrbracket$$

(34) // Reply message or timeout reaches the client :

$$\begin{array}{l}\mathcal{E}\llbracket \mathcal{K}, \Delta K, \texttt{waitresp}(i) \;\|\; \texttt{deliver}(k_K, q, i, \texttt{REPLY}, \_, M) \\ \qquad\qquad\qquad \begin{array}{ll} \mid & K.\mathit{ta}(i) \;=\; \texttt{timeout}(\_, \{q\}) \\ \wedge & K'.\mathit{ta}(i) \;=\; \texttt{replydone} \;\rrbracket \end{array} \;\rrbracket\end{array}$$
$$\leadsto \;\mathcal{E}\llbracket \mathcal{K}, K, \texttt{answer}(M)\rrbracket$$

(35)

$$\begin{array}{l}\mathcal{E}\llbracket \mathcal{K}, \Delta K, \texttt{waitresp}(i) \;\|\; \texttt{deliver}(k_K, q, i, \texttt{REPLY}, \_, M) \\ \qquad\qquad\qquad \begin{array}{ll} \mid & K.\mathit{ta}(i) \;=\; \texttt{timeout}(t, S) \\ \wedge & S' = S \setminus \{q\} \\ \wedge & S' \neq \{\} \\ \wedge & K'.\mathit{ta}(i) \;=\; \texttt{timeout}(t, S') \;\rrbracket \end{array}\end{array}$$
$$\leadsto \;\mathcal{E}\llbracket \mathcal{K}, K, \varepsilon\rrbracket$$

(36)

$$\begin{array}{l}\mathcal{E}\llbracket \mathcal{K}, \Delta K, \texttt{waitresp}(i) \\ \qquad\qquad\qquad \begin{array}{ll} \mid & K.\mathit{ta}(i) = \texttt{timeout}(t, \_) \\ \wedge & t \;>\; K.\mathit{localtime} \\ \wedge & K'.\mathit{ta}(i) = \texttt{timeoutsignaled} \;\rrbracket \end{array}\end{array}$$
$$\leadsto \;\mathcal{E}\llbracket \mathcal{K}, K\texttt{answer}\;(\texttt{\$TIMEOUT})\rrbracket$$

(37)

$$\mathcal{E}\llbracket \mathcal{K}, K, \texttt{deliver}(k_K, \_, i, \texttt{REPLY}, \_, M) \;\mid\; K.\mathit{ta}(i) = \texttt{timeoutsignaled} \;\rrbracket$$
$$\leadsto \;\mathcal{E}\llbracket \mathcal{K}, K, \varepsilon \;\rrbracket$$

3.6.7 $\mathsf{trns}\Longrightarrow\boxed{\mathsf{busAdpt}}\Longrightarrow\mathsf{trns}$ and $\boxed{\mathsf{busAdpt}}\overset{electric}{\Longrightarrow}\boxed{\mathsf{busAdpt}}$

$$outport \;:\; NodeState \times NodeIdent \;\twoheadrightarrow\!\!\!\!\!\!|\;\; Adapter \times BusAdr$$

$$outport(K,n) = \begin{cases} (D,b) & \mathsf{if}\ K.routingTab.gate(n) = \mathtt{drain}(D,b,\_) \\ outport(K,n') & \mathsf{if}\ K.routingTab.gate(n) = \mathtt{hop}(n') \end{cases}$$

$X \;\in\; K.adapters \;\cap\; BusAdapterIn$
$M \;\in\; Msg\_G$
$M' \;\in\; BusMsg\_G$

---

(38) // send message to the gateway node as given by the routing table:

$$\begin{array}{rl} & \mathcal{E}[[\mathcal{K}, K, \mathtt{deliver}(M) \\ & \qquad\qquad \mid\quad M = (t,o,i,x,y,m) \\ & \qquad\qquad \wedge\quad t \;\in\; NodeIdent \\ & \qquad\qquad \wedge\quad t \neq K.ownIdent \\ & \qquad\qquad \wedge\quad (D_K, a) \;==\; outport(K,t)\;\;]] \\ \leadsto & \mathcal{E}[[\mathcal{K}, K, D_K\ \boxed{\mathsf{->}}\ \mathsf{putMsgToBus}\ (a, \mathtt{busmsg}\ (t, K.ownIdent, M)\ )\ ]] \end{array}$$

(39) // special treatment for `$CAS` :

$$\begin{array}{rl} & \mathcal{E}[[\mathcal{K}, K, \mathtt{deliver}(M) \\ & \qquad\qquad \mid\quad M = (\mathtt{\$CAS}, o,i,x,y,m) \\ & \qquad\qquad \wedge\quad K \neq \mathcal{C}\;\;]] \\ & \qquad\qquad \wedge\quad M' = (K.routing.CASident, o,i,x,y,m)\;\;]] \\ \leadsto & \mathcal{E}[[\mathcal{K}, K, \mathtt{deliver}(M')\ ]] \end{array}$$

(40) // two nodes on one bus perform a data transmission:

$$\begin{array}{rl} & \mathcal{E}[[\mathcal{K}, K, D_K\ \boxed{\mathsf{->}}\ \mathsf{putMsgToBus}\ (a, M')\ \ \|\ \ L, D_L\ \boxed{\mathsf{->}}\ \mathsf{watchBus} \\ & \qquad\qquad \mid\quad D_K \;\in\; K.adapters \;\cap\; BusAdapterOut \\ & \qquad\qquad \wedge\quad D_L \;\in\; L.adapters \;\cap\; BusAdapterIn \\ & \qquad\qquad \wedge\quad D_K.bus \;=\; D_L.bus \\ & \qquad\qquad \wedge\quad D_L.busAdr = a]] \\ \leadsto & \mathcal{E}[[\mathcal{K}, K, \varepsilon\ \|\ L,\ D_L\ \boxed{\mathsf{->}}\ \mathsf{incomingMessage}(M')\ ]] \end{array}$$

(41)

$$\begin{array}{rl} & \mathcal{E}[[\mathcal{K}, K, X\boxed{\mathsf{->}}\ \mathsf{incomingMessage}\ (\,\mathsf{busMsg}(K.ownIdent, \_, M)\,)\ ]] \\ \leadsto & \mathcal{E}[[\mathcal{K}, K, \mathtt{deliver}(M)\ \|\!\!+\ X\boxed{\mathsf{->}}\ \mathsf{watchBus}\ ]] \end{array}$$

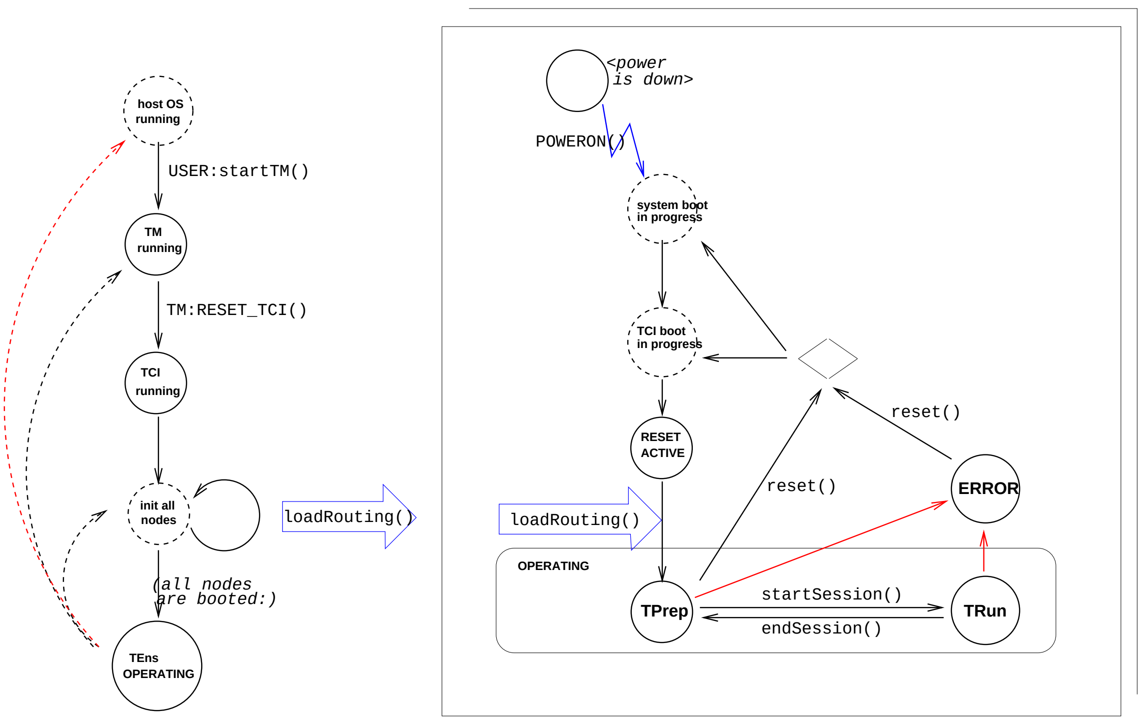
3.7 Application *and* Transport Layer : General Node State Control

3.7.1 Principles of Operation (Non-Normative)

One of the most ugly problems to formalize is the behavior of a complex (and only generically known!) distributed system w.r.t. **powering on**, **initialization** and **reset** (in the following subsumed as *init operations*).

This is because these init operations (1) let single nodes of the TEns go into or through some naturally undefined or pre-defined state, and (2) because they influence *all* the different layers of the architecture, which normally should only be specified by cleanly separated and closed mathematical systems.

Figure 7 Booting Cycle of a single Normal Node, and of the CAS node



TUB-TCI must implement a rather defensive way of *resetting*, which re-initializes all resources involved in communication in a hard manner. Only this — together with a topologically correct schedule, see below, — allows a well-defined reset behavior, even with most heterogenous hardware collections, — e.g. w.r.t. the deletion of pending PDUs from all the different pipelines and caches.

Furthermore we demand that *always all* nodes in a TEns must be reseted, and afterwards all nodes must be re-initialized! Every *partial* reset of a TEns may yield unspecified results!

The boot cycle for a *single* node is rather simple, cf. figure 7: Either a power-up or an explicit `reset()` *SRQ* will switch the node into an `reset_active` state.

In this state trns is not yet/no longer active. Instead, the node listens to some busses for „physically“ incoming messages. (The corresponding BusAdpts are collected in the set `bootListeners`.)

The only message understood in this state is `loadRouting()`. The service procedure for this *SRQ* loads the routing table of the node, starts the trns subsystem, and then switches the node to an `operating` state. From now in trns is running normally, as described above.

The overall power up sequence for a TEns is mainly determined by the *physical topology* of bus linkages and boot listeners. This is because the operability of the trns layer between two nodes is not given, until all nodes forming the physical communication path between them are in the `operating` state, i.e. have received an explicit initialization message from CAS. The resulting problems can easily be seen when looking at figures 8 and 9[34].

[34]This requirement is one ugly instance of the general problem of „architectural information leak“ mentioned above, see 2.9.

So the initialization sequence has to be planned rather carefully by some scheduling algorithm, using topology information only known to TM.

But on physical level the *CASNode* is the only root of the physical network, from which all other nodes are reachable. So the boot sequence has to be performed by CAS, but based on the topology information provided by TM.

Principally the initialization sequence is as follows :

1. On every node in a TEns (except the CAS-node), there must be some auto-start facility. This automatically starts the execution of the local TCI implementation demon. This can happen e.g. as soon as the operating system has been restarted. The liveness of the demon should be watched by the OS. It could be restarted when a certain reset button is pressed physically etc.
2. After this local TCI implementation has completed its internal initializations, it switches to the uninitialized state „`reset_active`“.
3. In this state there is at least one bus input adapter „listening“ on its bus, and waiting for a `loadRouting()` message to arrive. These bus input adapters are called „boot listeners“.
4. On the CAS node there is no autostart of TCI, but there must be a TM running.
5. The TM explicitely starts the CAS, thereby supplying topological and routing information about each individual node in the whole TEns.
6. The CAS initializes the TCI library, e.g. the routing table, running on its own node.
7. Then the CAS arbitrarily picks one of those nodes of the TEns which it can *already reach*, and tries to initialize it.
   At the beginning only the nodes connected via a single bus directly to the CAS node are reachable from the CAS, simply because all other nodes do not have any routing information yet.
   So the CAS sends a `loadRouting()` request to one of these reachable nodes.
8. When receiving a `loadRouting()` message on one of its boot listeners the receiving TCI implementation initializes its routing tables and its own *NodeIdent*, initializes its trns software and all selected bus drivers and switches into `operating` state. From now on this node is able to receive messages and to route messages to the other nodes to which it is connected.
9. Since now more nodes may have become reachable, CAS can return to step 7, – until all nodes in TEns are initialized.

The process of resetting is even more critical :

1. CAS must send a `reset()` request to some (arbitrarily chosen) node, which is *not needed* to reset any other node, i.e. to reach any node which is still in the state `operating`.
   After `reset()` has been received and evaluated, the trns layer of this node does not work any more at all, until a `loadRouting()` has been received on a boot listener, as described above.
   So if you shut down the wrong node, you may have to *walk* to the equipment and press a reset button physically `;-[`
2. After one node has been put to `reset_active`, some more nodes may have been become free for being reseted, according to the condition in step 1.

So step by step all nodes are put into the `reset_active` state, as depicted in figure 9.

As soon as the last node is in the `reset_active` state, the old TEns does not exist any more, and a new TEns can be established by rebooting the corresponding nodes by a procedure as described above.

Please notice, that the situation is (should be) exactly as immediately after the first powering on of the whole „hardware farm“, since every reception of a `reset()` messages puts any node into exactly the same state as its first power up does ! This is according to some of the basic design principles for TUB-TCI, namely simplicity and robustness.

3.7.2 The Confirmation Problem with `reset()` and `loadRouting()`

Above we mentioned the „architectural leak“, that the evaluation of certain *SRQ* messages affects the infra-structure needed to evaluate *SRQ* messages, cf section 2.9.

This has the ugly consequence that there are situations in which it is *impossible* to get a direct confirmation for an init operation, i.e. any corresponding `reply()` message.

Therefore the `loadRouting()` *SRQ* is provided by TUB-TCI in two flavors: confirmed or unconfirmed. A `reset` *SRQ* is always unconfirmed, as explained below.

Let us look at figure 8, and shall the first node initialized by the TM (running on the CASnode) be called N_1, the second one N_2. Then, since the trns of N_2 is not yet operating, but is required for a communication from N_1 to CASnode, N_1 has to be sent its `loadRouting()` message *unconfirmed*. Indeed the TM does not know, if *and when* this booting of N_1 will have succeeded. TM will probably use some timing *heuristics* to determine some waiting duration.

After waiting some time the TM tries to boot N_2, now requesting a confirmation. This is possible, because the trns subsystem needed for communication form N_2 to TM should have been established.

If booting of N_2 succeeds and a positive confirmation arrives, then CAS knows, that *both* boot operations (of N_1 and of N_2) have succeeded. (If N_1 had not succeeded, then N_2 would not even have received the `loadRouting()` *SRQ*).

With reset it is – again – worse:

Even on a pure conceptual level it is nearly impossible to model a „last reply“, i.e. a message which is guaranteed to be sent as the last activity of a system which is going down totally and switching all hardware in a defined reset state.

Such a reset is a *singularity*, a kind of „black hole“ in the universe of our TEns. It is nearly impossible to specify the behavior when approaching such a singularity in detail, while at the same time demanding its easy implementability with very different technologies.

So there cannot be any (direct) confirmation sent as a reply on a `reset` request.

What can be done, is to save a single datum to survive the „Big Bang“, e.g. in a flash ROM or a CMOS device. By this we support an *indirect* confirmation to reset: Each reset command is attributed by a distinct „reset sequence marker“, the value of which can be chosen arbitrarily by the CAS. The value of the last received reset sequence marker survives the (otherwise *total*) reset of the node. When rebooting a node with confirmation, the reset marker of the last reset command received (and performed) is contained in the reply to the boot request. When a node must be rebooted without confirmation (for the reasons mentioned above), a special inquiry message allows CAS to read the value of the last reset marker explicitely.

Figure 8 TCI Boot Sequence Rippling through a Test Ensemble.

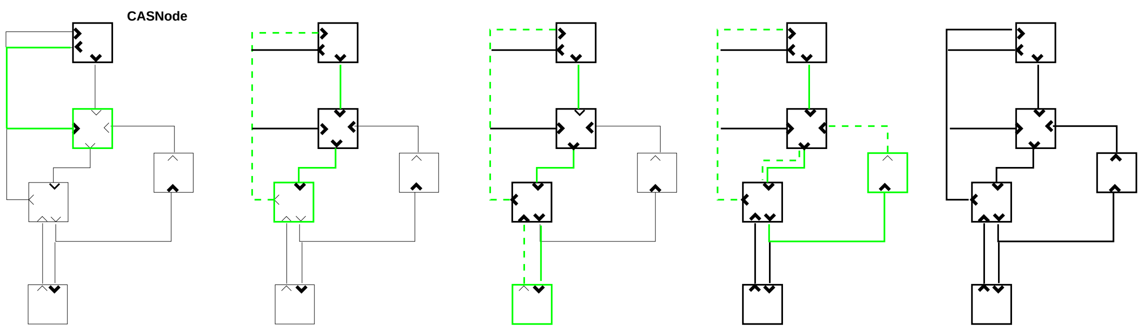
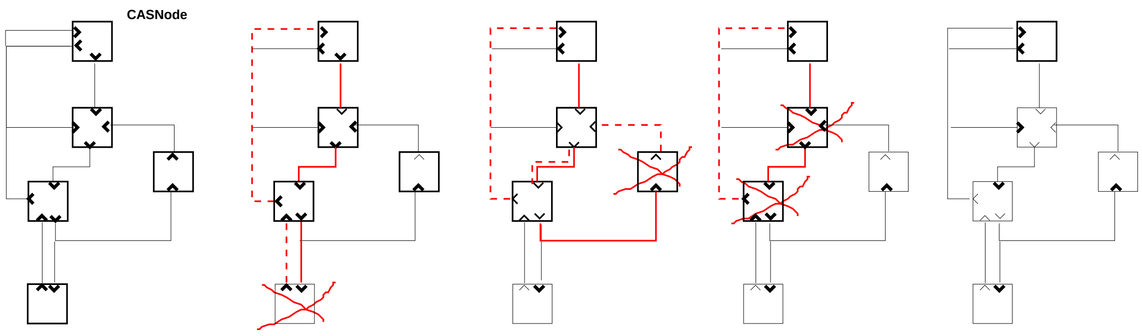


Figure 9 Reset Commands and (theoretically possible) Confirmations Rippling through a Test Ensemble.



3.7.3 Possible Alternatives and Variants of Design Decisions

This reset/boot concept induces some overhead and may – in special settings – lack performance. Some possible variants are described in section 5.1.2.

Please notice that it is one of those „minor design decisions“, that there is neither a means (1) for altering the routing table of a distinct node without resetting it, nor (2) for performing a reset of a node while preserving the routing info.

While both possibilities seem useful w.r.t. practical operations, they are not necessarily part of a slim kernel design as presented herein.

3.7.4 General Purpose `reply()` Messages

(42)

| *ErrorCode*

(43)

$$
\begin{array}{l}
GF \ ::= \ \texttt{generalFailure} \langle\!\langle\, [\ \ cod : ErrorCode \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ txt : TextString\ \ ] \rangle\!\rangle \\
GS \ ::= \ \texttt{OK} \\
\qquad\quad \mid\ \texttt{generalSuccess} \langle\!\langle\, [\ \_DATA\_\ ] \rangle\!\rangle \\
GR \ == \ GF \ \cup \ GS
\end{array}
$$

3.7.5 Messages

(44)

$$
\begin{array}{|l}
ResetEpoque \\
\hline
\texttt{\$powon} \ \in \ ResetEpoque
\end{array}
$$

(45)

$$
\begin{array}{|l}
SessionId
\end{array}
$$

(46)

$$
\begin{array}{|l}
ResetLevel\_TCI \ ::= \texttt{all} \\
\qquad\qquad\qquad\quad \mid \ \ldots
\end{array}
$$

(47)

$$
\begin{array}{|l}
ResetLevel\_host[NodeClass] \\
\hline
\texttt{\$noreset} \ \in \ ResetLevel\_host
\end{array}
$$

(48)

$$
\begin{array}{l}
SRQS\_from\_CAS\_to\_Node \quad ::= \\
\qquad\qquad \texttt{loadRouting} \ \langle\!\langle [\ ownIdent : NodeIdent \\
\qquad\qquad\qquad\qquad\qquad\quad routing : RoutingMap \quad] \rangle\!\rangle \\
\qquad\qquad\quad \Rrightarrow \ \texttt{initComplete} \ \langle\!\langle ResetEpoque \times Time \rangle\!\rangle \\
\qquad\quad \mid \ \texttt{inquireResetInfo} \\
\qquad\qquad\quad \Rrightarrow \ \texttt{resetInfo} \ \langle\!\langle ResetEpoque \times Time \rangle\!\rangle \\
\qquad\quad \mid \ \texttt{inquireHdwStateInfo} \langle\!\langle\, [\ select : \mathbb{N}\ ] \rangle\!\rangle \\
\qquad\qquad\quad \Rrightarrow \ \texttt{hdwInfo} \ \langle\!\langle \_DATA\_ \rangle\!\rangle \\
\qquad\quad \mid \ \texttt{startsession} \langle\!\langle [\ sessionId : SessionId \\
\qquad\qquad\qquad\qquad\qquad\quad maxDuration : Duration \cup \{\bot\}] \rangle\!\rangle \\
\qquad\qquad\quad \Rrightarrow \ GR \\
\qquad\quad \mid \ \texttt{stopsession} \\
\qquad\qquad\quad \Rrightarrow \ GR \\
\qquad\quad \mid \ \texttt{reset} \langle\!\langle\ ResetEpoque \setminus \{\texttt{\$powon}\} \ \times ResetLevel\_TCI \times ResetLevel\_host \rangle\!\rangle
\end{array}
$$

(49)

$$
\begin{array}{|l}
\mathit{Verdict} ::= \texttt{vERR} \mid \texttt{inconc} \mid \texttt{failed} \mid \texttt{passed} \\
\hline
\texttt{vERR} < \texttt{inconc} < \texttt{failed} < \texttt{passed}
\end{array}
$$

(50)

$$
\begin{array}{ll}
\mathit{SRQS\_from\_RBD\_to\_CAS} ::= & \\
\quad \texttt{setVerdict} \ \langle\!\langle [& \mathit{verdict} : \mathit{Verdict} \\
 & \mathit{origin} : \mathit{LUID} \\
 & \mathit{sourceLocator} : \_\_\mathit{DATA}\_\_[\mathit{origin}] \\
 & \mathit{localtime} : \mathit{Time} \ ] \rangle\!\rangle \\
\quad\quad \Rrightarrow \ \mathit{GR} & \\
\quad \texttt{getGlobalParameter} \ \langle\!\langle \boxed{a =} \mathit{Ident} \rangle\!\rangle & \\
\quad\quad \Rrightarrow \ \texttt{gobalParamValue} \ \langle\!\langle \_\_\mathit{DATA}\_\_[a] \rangle\!\rangle &
\end{array}
$$

// **Please notice** that the values of type *LUID* are used to identify/address/represent all living subsystems from the actor category, — here: the RBD raising the verdict. *LUIDs* and will be introduced below in section 3.8.4 on page 57.

3.7.6 Additional State Space

(51)

TM

$$\boxed{\texttt{->}} \ \mathsf{getGlobalParameter} : \boxed{a =} \mathit{Ident} \ \rightarrow \ \_\_\mathit{DATA}\_\_[a] \ \cup \ \mathit{GF}$$

...

(52)

$$\mid \mathcal{M} : \mathsf{TM}$$

(53)

NodeState

...
resetMark : *ResetEpoque*
resetTime : *Time*
bootTime : *Time*

sessionID : *TextString* $\cup \{\perp\}$
sessionStarted : *Time*
...

(54)

```
── NodeState_Init ──────────────────────────────
│ NodeState
│ K.ownIdent = $nodeNotInitialized
│ K.routingTab = {}
│ K.ta = {}
│ K.suppressReply = {}
│ K.sessionId = ⊥
│ K.resetTime = K.localTime
│ ...
└────────────────────────────────────────────────
```

(55)

```
── CASstate ────────────────────────────────────
│ nodes : ℙ NodeIdent
│ // busses : ℙ Bus
│ // adapters : ℙ BusAdapter
│ nodesRouting : NodeIdent ⇸ RoutingTable
│ bootingAdapters : ℙ BusAdapter
├──────────────────
│ bootingAdapters = ⋃{x ∈ nodes • x.bootListeners}
│ ...
└────────────────────────────────────────────────
```

(56)

$| \ \mathcal{C} \ : \ \mathit{CASstate}$

3.7.7 $\mathsf{CAS} \overset{2}{\Longrightarrow} \boxed{\mathsf{NodeServer}}$

$K : \mathit{NodeState}$
$\{D_0, \ldots, D_n\} \;==\; K.\mathit{bootListener}$
$\{I_0, \ldots, I_m\} \;==\; K.\mathit{adapters} \cap \mathit{BusAdapterIn}$
$\mathsf{AllBootListenersListening} \;==\; D_0\boxed{\texttt{->}}\mathsf{watchBus} \mathbin{|\!|\!+} \ldots \mathbin{|\!|\!+} D_n\boxed{\texttt{->}}\mathsf{watchBus}$
$\mathsf{AllInputAdaptersListening} \;==\; I_0\boxed{\texttt{->}}\;\mathsf{watchBus} \mathbin{|\!|\!+} \ldots \mathbin{|\!|\!+}\; I_m\boxed{\texttt{->}}\mathsf{watchBus}$

---

(57)
$\mathcal{E}[\![\mathcal{K}, \Delta K, \mathsf{PowerOnReset} \;\mid\; K \neq \mathcal{C}]\!]$
$\leadsto\; \mathcal{E}[\![\mathcal{K}, K, \mathsf{HOST\_POWERING\_ON}]\!]$
$\leadsto\; \mathcal{E}[\![\mathcal{K}, K, \mathsf{PerformReset}\ (\texttt{\$powon})]\!]$

(58) // *On power up all transport data is reset,*
// *and at least one bus input adapter („boot listener“) starts listening:*
$\mathcal{E}[\![\mathcal{K}, \Delta K, \mathsf{PerformReset}(x)$
$\quad\mid\; K \neq \mathcal{C}$
$\quad\wedge\; K' = [\mathit{NodeState\_Init} \;\mid\; \mathit{resetMark} \;=\; x\,]\;]\!]$
$\leadsto\; \mathcal{E}\,[\![\mathcal{K}, K,\; \mathsf{AllBootListenersListening}\;]\!]$

(59) // Initialization messages are recognized by „physical“ addressing.
// `service()` will not be reached since there is no valid $K.\mathit{ownIdent}$.
$\mathcal{E}[\![\mathcal{K}, \Delta K, \texttt{deliver}(\_, \_, i, \texttt{REQ}, \texttt{false}, \texttt{loadRouting}\,(N, R, \texttt{true}))$
$\quad\mid\; K.\mathit{ownIdent} = \texttt{\$nodeNotInitialized}$
$\quad\wedge\; K'.\mathit{ownIdent} = N$
$\quad\wedge\; K'.\mathit{routingTab} = R\;\;]\!]$
$\leadsto\; \mathcal{E}[\![\mathcal{K}, K, \texttt{deliver}(\texttt{\$CAS}, N, i, \texttt{REPLY}, \_, M') \mathbin{|\!|\!+} \mathsf{AllInputAdaptersListening}$
$\quad\mid\; M' = \texttt{initComplete}(K.\mathit{resetMark}, K.\mathit{localTime})\;\;]\!]$

(60)
$\mathcal{E}[\![\mathcal{K}, \Delta K, \texttt{deliver}(\_, \_, i, \texttt{REQNW}, \texttt{true}, \texttt{loadRouting}\,(N, R, \texttt{false}))$
$\quad\mid\; K.\mathit{ownIdent} = \texttt{\$nodeNotInitialized}$
$\quad\wedge\; K'.\mathit{ownIdent} = N$
$\quad\wedge\; K'.\mathit{routingTab} = R\;\;]\!]$
$\leadsto\; \mathcal{E}[\![\mathcal{K}, K, \mathsf{AllInputAdaptersListening}\;\;]\!]$

(61)

$$\begin{array}{ll} & \mathcal{E}[[\mathcal{K}, K, \texttt{service}(\_, \texttt{reset}(x, \texttt{all}, \texttt{\$noreset}))\\ & \qquad\qquad \| \ \textsf{AllOtherRunning\_TCI\_ThreadsOnThisNode}\ ]] \\ \leadsto & \mathcal{E}[[\mathcal{K}, K, \textsf{PerformReset}(x)]] \end{array}$$

(62)

$$\begin{array}{ll} & \mathcal{E}[[\mathcal{K}, K, \texttt{service}(\_, \texttt{reset}(x, \texttt{all}, R))\\ & \qquad\qquad \| \ \textsf{AllOtherRunning\_TCI\_ThreadsOnThisNode}\ ]] \\ \leadsto & \mathcal{E}[[\mathcal{K}, K, \textsf{RESET\_HOST\_SYSTEM}\ (R)]] \\ \leadsto & \mathcal{E}[[\mathcal{K}, K, \textsf{PerformReset}(x)]] \end{array}$$

(63)

$$\begin{array}{ll} & \mathcal{E}[[\mathcal{K}, K, \texttt{service}\,(o\!::\!i, \texttt{inquireResetInfo}\ )\ ]] \\ \leadsto & \mathcal{E}[[\mathcal{K}, K, \texttt{reply}\,(o\!::\!i, \texttt{resetInfo}(K.\mathit{resetMark}, K.\mathit{resetTime}))\ ]] \end{array}$$

(64)

$$\begin{array}{ll} & \mathcal{E}[[\mathcal{K}, K, \texttt{service}\,(o\!::\!i, \texttt{inquireHdwStateInfo (n))}\ ]] \\ \leadsto & \mathcal{E}[[\mathcal{K}, K, \texttt{reply}\,(o\!::\!i, \texttt{hdwInfo}(\textsf{nodeServer}\ \boxed{\texttt{->}}\ \textsf{inquireHdwStateInfo}(n))) \end{array}$$

(65)

$$\begin{array}{ll} & \mathcal{E}[[\mathcal{K}, \Delta K, \texttt{service}(o\!::\!i, \texttt{startSession}(i, d)) \\ & \qquad\qquad \mid \quad K'.\mathit{sessionId} = i \\ & \qquad\qquad \wedge \quad K'.\mathit{sessionStarted} = K.\mathit{localTime}\ ]] \\ \leadsto & \mathcal{E}[[\mathcal{K}, K, \texttt{reply}\,(o\!::\!i, \texttt{OK})]] \end{array}$$

(66)

$$\begin{array}{ll} & \mathcal{E}[[\mathcal{K}, \Delta K, \texttt{service}(o\!::\!i, \texttt{stopSession}(i, d))]] \\ \leadsto & \mathcal{E}[[\mathcal{K}, K, \texttt{reply}\,(o\!::\!i, \texttt{OK})]] \end{array}$$

(67)

$$\begin{array}{ll} & \mathcal{E}[[\mathcal{K}, \Delta K, \texttt{service}(o\!::\!i, \texttt{getGlobalParameter}(n)) \\ & \qquad\qquad \mid \quad d\ =\ \mathcal{M}\boxed{\texttt{->}}\textsf{getGlobalParameter}\,(n) \\ & \qquad\qquad \wedge \quad n \notin \mathit{GF} \\ \leadsto & \mathcal{E}[[\mathcal{K}, K, \texttt{reply}\,(o\!::\!i, \texttt{gobalParamValue}(d))]] \end{array}$$

3.8 General Concepts concerning Actor Subsystems

3.8.1 Actors, *Actor Classes* and Factories

A mentioned in the survey above (section 2.8.2), all subsystems which must be dynamically created are instances of the actor category. Dynamic creation is e.g. required by the need to reflect the changes in the physical world (e.g. concerning the types and configurations of node-internal hardware resources) when moving one ATS between executing nodes of different types and capabilites.

Actor subsystems can be characterized by the following facts:

- Each actor subsystem must be created explicitly by an RBD sending a `create()` message to CAS[35].
- Every actor is permanently hosted on one distinct node, its „hosting node“.

[35] Or directly to a NodeServer or a factory, if the variants listed in 5.1.3 are chosen.

- Each actor is throughout its lifetime instance of one distinct *actor class*.
- To create an actor subsystem of a given actor class on a given node, a corresponding factory for this actor class must be hosted on this node.
- The current state of an actor is totally determined by the the current states of a fixed collection of *configuration parameters*.
- Each actor is controlled throughout its lifetime by one or more RBDs using the services offered by this actor.
- Each actor must offer the services for **changing** the current values of its configuration parameters, and for **inquiring** their current values.
- Each actor *may* offer the service of performing some **run-time operations**. The execution of such a `rtOperation()` service mostly corresponds to some activity on the „physical semantics" of the represented hardware device.
  These services may either be defined to return a result, or may only be called unconfirmed.
- Each actor class totally defines for all of its instances (1) the names, types and constraints of all *configurationparameters*, as well as (2) the set of supported *run-time operations,*

3.8.2 Configuration Parameters

Central component of each actor class definition is the definition of the configuration parameters. The simple half of this definition can be seen as a „table" containing one column for (1) the name of the configuration parameter, (2) its basic type, (3) the „parameter change allowance mode", and (4) a default value, which may be undefined. In such a table each configuration parameter takes one line entry, – the first column containing only unique values, i.e. serving as a „key field" in the sense of a relational data base. Please refer to section 3.9.5 for an illustrative example.

The „parameter change allowance mode" of each configuration parameter determines in which system phases a change of this parameter is allowed:

`CO`	Cannot be set at all, is read-only and *constant* throughout the lifetime of the actor.
`RO`	Cannot be set at all, is read-only, but *not* guaranteed not to change.
`WI`	Can be written only once by giving it initially in the `create()` request.
`RW`	Can be altered as long as the „run-time" (TRun) has not yet been started.
`RWRT`	Can always be altered, even in „run-time" (TRun).

The more complicated half of the configuration parameter definition is the specification of the *legal values* for the configuration parameters. These can *not* be given in such a simple line-by-line basis, since it must be possible to restrict the legal values to a certain *set* of *combinations*. So constraints involving more than one parameter value must be denotatable. For this sake we use *Z-schemata*, as described below in section 3.9.

Let *cp-map* be a user-supplied data structure, which maps the set of configuration parameter names of a given actor class to none, one or more than one valid value each. Let *injective cp-map* be a *cp-map*, which maps configuration parameter names to at most one valid value each. Then the definition of the configuration parameters is relevant for the user when requesting different services:

- When sending a `create()` request, a *cp-map* must be supplied by the calling RBD, which describes the initial state, requested for the newly created actor subsystem. This map *must* contain a (single, unique) value for each configuration parameter

which has a default value of ⊥ and is not of allowance type `CO` or `RO`. Additionally it *may* give a value (or even a set of values) for any other parameter which is not of allowance type `CO` or `RO`.
The overall combination of parameter values induced by the user-defined *cp-map* should be a *valid* combination. Otherwise the `create()` request may be totally rejected, or parts of the requests contained in the *cp-map* entries may be disregarded.

- When requesting the `setconfigparams()` service, a *cp-map* must be supplied, indicating the new parameter values to which the actor shall change its state. Depending on the parameter change allowance mode of the parameters involved, these changes are possible in TPrep, maybe also in TRun. This service will always be is rejected if the values in the *cp-map* are inconsistent or infeasible.
- When calling `getconfigparams()` on a given actor, the momentary design returns a complete *injective cp-map* , containing the current values of *all* configuration parameters of that actor.

The `setconfigparams()` service can called both unconfirmed or confirmed. The `getconfigparams()` service must be called confirmed[36].

3.8.3 Run-Time Operations

The run-time operations (`rtOperation`s) are used by an RBD to control the operation of an actor during run-time, or to perform some external operation on a physical device via its representing actor subsystem.

When defining an actor class, a free data type has to be given which describes all supported real-time operations by giving their request message formats, maybe together with the possible reply message formats.

All implementations of a driver/factory for that actor class have to decode this message type and to perform the appropriate activities.

Run-time Operations can be defined both unconfirmed or confirmed, and have to be called as defined.

3.8.4 Addressing of Actors for performing operations, – the concept of *LUID*

Whenever a new actor is created, it is assigned a *limited unique id* (= *LUID*). This *LUID* is unique within a certain interval of real-time (a so called *epoque*) within the whole TEns (i.e. independently of the hosting nodes).

An RBD which wants to request a service from an actor (`setParam()`,`getParam()` or `rtOperation()`) must address this actor by its *LUID*.

On the level of implementation there are three sub-cases :

1. The hosting node of the given *id* is known, and it is the local node itself.
   In this case the actor is hosted „here“. Therefore the actor class is known, the appropriate driver/factory can be identified and all requests can be fulfilled by an API-call to the according driver/factory.
2. The hosting node of the given *id* is known, but it is *not* the local node itself.
   Then the request has to be passed on to the hosting node (using trns, according to the local *routing table*).
3. The hosting node of the given *id* is *not* known.

[36]Naturally, because the reply will contain the requested information.

In this case a `lookUp()` request is sent to the CAS, which returns either an error indication or the *NodeIdent* of the hosting node[37]. This information is added to the local cache and must never again be fetched from CAS for the duration of the *epoque*. Now that the hosting node is known, proceed as in 1. or 2.

3.8.5 Push-Channels and the QAS subsystem

All „spontaneuosly" active hardware devices (like in-ports, out-ports, timers, sensors etc.) must be represented by instances of „producer". This is a category of subsystems which enhances the category actor: The offered services are the same, but additionally a producer has an active behavior acting as a client and generating `value_event()` messages: Whenever e.g. a driver's demon polling the hardware, or an interrupt handler activated from outside, detects the incoming of a datum or the occurrence of an event, it calls the `value_event()` service, which is offered by the QAS subsystem.

The (ubiquitous) QAS subsystem realizes the „push-channels", and distributes the signal event to all currently subscribed drains hosted on the same node, and maybe to remote nodes which host further drains.

The final *drains* for all `value_event()` messages are always some (or no) subsystems of RBQ category, i.e. subsystems realizing the passive „signal receiver" semantics. A classical example is the famous „input queue" occurring in the definition of the „snapshot" executional semantics of the TTCN-3 „`alt`" statement [ttc01].

The QAS server is necessary, because a producer subsystem is just producing its `valueEvent()`s, but is not aware which RBQs currently are consuming its event stream, i.e. need to be notified. This is because these connections may be under programmed control, i.e. change dynamically during the execution of the ETS. This dynamic control is realized by `subscribe()` and `unsubscribe()` messages, by which an RBD can make any RBQ[38] to receive (from now on) all value events from a given actor .

3.9 Actor Class Dynamic Declaration Interface

3.9.1 Principles of Operation (Non-Normative)

As mentioned above, the specification of TCI does not define the set of possible actor classes, but provides the means to plug-in definitions of new actor classes and the corresponding factories.

The declaration of an *actor class* on the conceptual level consists of the definitions of (1) the collection of configuration parameters, and (2) the supported run-time operations.

On the implementation level additionally (3.a) the code of a corresponding factory and for the driver functionality has to be defined, (3.b) compiled for and installed on each node, on which the corresponding *actor* subsystems shall be created, and (3.c) made known to their NodeServers.

Please notice that in the current version of TUB-TCI the steps (1) and (2) are *purely conceptual*. A proposal for lifting the actor class definitions to the level of object language are discussed in 5.2.1.

Please notice that in the current version of TUB-TCI the realization of step (3) is left totally unspecified. From its (current) point of view the identification process of the distinct factory which corresponds to a given `create()` message is *hard-wired* into the

[37]Maybe together with additional information, e.g. indicating the actor's actor class.

[38]**Please notice** that the tight connection of one RBQ and one RBD to form one `component`, is not realized in TUB-TCI, but done on the level of *using* it.

nodeServer. Section 5.2.3 proposes a further specification to describe classes of hardware nodes and their capabilites. This would include clean means for declaring the hosted factories for a given node.

The definition of an actor class is done by creating a Z-schema called the *defining schema*. This schema must specify (1) the configuration parameters (PI) and (2) the supported Run-Time Operations (RT). It can be built either from scratch, or by derivation.

If the defining schema is built from scratch, it must include the schema *ActorClassDescription*, as given below. A Free Type has to be assigned to the variable RT, and a list of *ParameterInfo* has to be built and assigned to the variable PI. The latter is supported by the function *unpack*, which initializes a *ParameterInfo* with pairs of name and type extracted from a *schema*.

If the new *actor class* is a derivation from an existing class without introducing new configuration parameters, the new schema simply needs to include the schema of the class derived from. Otherwise, when additional configuration parameters or run-time operations are necessary, the schema can be initialized by including the schema valued function applications $extendparams(s, p)$, $extendops(s, r)$, or $extend(s, p, r)$, where s is the schema representing the class to inherit from, p is the schema representing the additional configuration parameters, and r is a data type representing the added Run-Time Operations.
// **Please notice** that this „definition by inclusion" does not induce any *a priori* meaningful „super-type/sub-type" relationship between the involved actor classes !

Each factory which is to be integrated into the TCI system must be specified by a further schema. This must combine (1) the schema of the (most specific) actor class corresponding to the factory, (2) maybe further constraints on the configuration parameters' values, and (3) the abstract schema *Actor_Factory* as given below. The latter represents some *method calls* . These must be implemented by the physical piece of code which realizes this factory. For each node on which the corresponding actor class shall be instantiable, an implementation of the factory has to be provided and installed.

3.9.2 Factories with Fully Specified Configuration

A Factory is said be of „fully specified configuration", if it never rejects a *cp-map* as inconsistent, which is valid according to its *defining schema*.

3.9.3 Additional State Space

(68)

$$ParameterUpdateAllowance ::= \mathtt{CO} \mid \mathtt{RO} \mid \mathtt{WI} \mid \mathtt{RW} \mid \mathtt{RWRT}$$
$$\mathtt{CO} < \mathtt{RO} < \mathtt{WI} < \mathtt{RW} < \mathtt{RWRT}$$

(69)

```
── ParameterDescription ──────────────────────────────
 type : __TYPE__
 updatemode : ParameterUpdateAllowance
 value :  type
 //allowed : ℙ type
 default : type ∪ {⊥}
 name : IDENT        // // This field seems quite REDUNDANT.
──────────────────────────
 default ∈ allowed  ∨  default = ⊥
──────────────────────────────────────────────────────
```

(70)

$$ParameterDescriptionList == IDENT \twoheadrightarrow\!\!\!\!\!\!\!+ \; ParameterDescription$$

(71)

```
── ActorClassDescription ─────────────────────────────
 PI : ParameterDescriptionList
 RT : __FREE_TYPE__
──────────────────────────────────────────────────────
```

// **Please notice** that the „places“ given by the identifiers in the domain of the value of *PI* of any defining schema, will be treated like like „top-level“ schema fields when deriving another actor class or when declaring a factory, i.e. they will be further constrained *individually*.

Also, when parameterizing the creation of a corresponding actor subsystems, the corresponding values may individually be supplied or not.

(72)

```
── Actor_Factory ─────────────────────────────────────

 ActorClassDescription

 [->] create : LUID × pack PI → GR
 [->] delete : LUID × LUID → DeleteFeedBack
 [->] changeParams : LUID × pack PI → ParamFeedBack
 [->] dumpParams : LUID → pack PI
 [->] executeRtOperation : LUID × RT → __DATA__
 // These are listed here just for survey, and will be introduced in the next
 //    sections, when discussing the corresponding functionalities they support.
 ...
──────────────────────────────────────────────────────
```

3.9.4 Auxiliary and Convenience Definitions

The following auxiliary functions are useful for the definition of new classes and constraints :

(73)

$$
\begin{array}{|l}
unpack\ :\ \_\_SCHEMA\_\_\ \rightarrow\ ParameterDescriptionList \\
pack\ :\ ParameterDescriptionList\ \rightarrow\ \_\_SCHEMA\_\_ \\
\hline
unpack(s)\ =\ \lambda\, x \bullet x.1 \mapsto [ParameterDescription \mid type = x.2;\ name = x.1]\ (\!|\ \boxed{\Downarrow} s\ |\!) \\
pack(p)\ =\ \boxed{\Uparrow}\ (\ \lambda\, x \bullet x.1 \mapsto x.2.value\ (\!| p |\!)\ ) \\
//\ \text{„}\boxed{\Downarrow}\text{“ means the pushing of a (meta-)value of type „SCHEMA“} \\
//\qquad \text{to a value from } IDENT \ \twoheadrightarrow\!\!\!\!\!\!\shortmid\ \ \_\_TYPE\_\_ \\
//\ \text{„}\boxed{\Uparrow}\text{“ means the lifting of a mapping from } IDENT \ \twoheadrightarrow\!\!\!\!\!\!\shortmid\ \ \alpha \\
//\qquad \text{to a binding value of an („automatically created“) corresponding} \\
//\qquad \text{schema type.}
\end{array}
$$

(74)

$$
\begin{array}{|l}
\text{— } ActorClassDescription \text{ —} \\
\ldots \\
V : IDENT\ \rightarrow\ \_\_DATA\_\_ \\
\hline
V(i)\ :\ PI(i)\ .type \\
V(i)\ ==\ PI(i)\ .value
\end{array}
$$

// For brevity we will further simply write

$$V\,\texttt{fieldname}$$

as equivalent to

$$V\,(\text{"}\texttt{fieldname}\text{"})$$

(75)

$$
\begin{array}{|l}
extendparams\ :\ \_\_SCHEMA\_\_ \times \_\_SCHEMA\_\_ \rightarrow \_\_SCHEMA\_\_ \\
extendops\ :\ \_\_SCHEMA\_\_ \times \_\_FREETYPE\_\_ \rightarrow \_\_SCHEMA\_\_ \\
extend\ :\ \_\_SCHEMA\_\_ \times \_\_SCHEMA\_\_ \times \_\_FREETYPE\_\_ \rightarrow \_\_SCHEMA\_\_ \\
\hline
extendparams(A, B) = [A \setminus PI \mid PI \subset A.PI\ \oplus\ (unpack\ B)] \\
extendops(A, C) = [A \setminus RT \mid RT = A.RT\ \oplus^{FDT}\ C\,] \\
//\ \text{The operator } \oplus^{FDT} \text{ shall combine ranges and domains of two Free Data Types} \\
extend(A, B, C) = extendparams(extendops(A, C),\ \ B)
\end{array}
$$

3.9.5 Example

There is a German idiom, often used in high school lesson hours filled with experimental chemistry, which says „You can see, that you see nothing“[39]. This verdict probably matches the previous section.

Since the definitions therein are *generic* definitions, their meaning can hardly become descriptive without *instantiating* them. Therefore we deviate from the standard way of our presentation and give an *example*, which shows the plugging-in of the declarations for two actor classes and one factory. As example we use a sequence of increasingly more specific declarations of `timer` subsystems, as depicted in table 5.

First we define the configuration parameters and the run-time operations separately :

[39] „Sie sehen, daß Sie nichts sehen.“

Table 5 Example for the Definition of Configuration Parameters

	type	*change allowance*	*default*	*range*
AC_Timer				
`minResolution`	*Duration*	$\leq$ `WI`		
`curResolution`	*Duration*			
`maxDuration`	*Duration*	$\leq$ `RO`		
AC_Timer_4711				
`minResolution`	*Duration*	$=$ `CO`		
`curResolution`	*Duration*			
`maxDuration`	*Duration*	$=$ `CO`		
`maxTransmissionDelay`	*Duration*	$=$ `RO`		
Fact_Timer_x4711_2.2				
`minResolution`	*Duration*	$=$ `CO`	`0.001`	
`curResolution`	*Duration*	$=$ `RW`	`0.01`	$\texttt{0.001} \leq v \leq \texttt{0.1}$
`maxDuration`	*Duration*	$=$ `CO`		$v < \texttt{3600.00}$
`maxTransmissionDelay`	*Duration*	$=$ `RO`		
	$\exists\, n : \mathbb{Z} \bullet V\ \texttt{curResolution} = n * sec(0.001)$			
	$V\ \texttt{maxDuration}\ /\ V\ \texttt{curResolution} \leq 2^{31} - 1$			

── *PI_Timer* ──

$minResolution : Duration$
$curResolution : Duration$
$maxDuration : Duration$

$$
\begin{aligned}
RT\_Timer ::= &\ \texttt{start} \\
&\mid \texttt{stop} \\
&\mid \texttt{reset}\langle\!\langle Duration\rangle\!\rangle \\
&\mid \texttt{read} \Rrightarrow\ \texttt{timerval}\langle\!\langle Duration\rangle\!\rangle
\end{aligned}
$$

No we use these both data types to build a defining schema for the actor class:

── *AC_Timer* ──

ActorClassDescription

$PI \subset unpack\ PI\_Timer$
$RT = RT\_Timer$
$PI\,(\text{"}\texttt{minResolution}\text{"}).mode \leq \texttt{WI}$
$PI\,(\text{"}\texttt{maxDuration}\text{"}).mode \leq \texttt{RO}$

// **Please notice** that the auxiliary function *unpack*() abuses its schema-type argument and reduces it to a simple mapping from names to types. This mapping is lifted to the object level.

Using further auxiliary functions we can make use of a kind of „copy-down inheritance“ to define a „subclass“ of *AC_Timer*. This subclass has (of course) a new name, and additional configuration parameters and run-time operations :

$$
\begin{array}{l}
\mathit{PI\_Timer\_4711} == [\mathit{maxTranmissionDelay} : \mathit{Duration}] \\
\mathit{RT\_Timer\_4711} ::= \texttt{setMark}(\mathbb{N}) \\
\qquad\qquad\qquad | \ \texttt{readMark}(\mathbb{N}) \ \Rrightarrow \ \texttt{timestamp}(\mathit{Time}) \\
\mathit{AC\_Timer\_4711} == [\ \mathit{extend}(\mathit{AC\_Timer}, \mathit{PI\_Timer\_4711}, \mathit{RT\_Timer\_4711}) \\
\qquad\qquad\qquad | \ \mathit{PI}(\text{"}\texttt{maxTransmissionDelay}\text{"}).\mathit{mode} = \texttt{RO} \\
\qquad\qquad\qquad\quad \mathit{PI}(\text{"}\texttt{minResolution}\text{"}).\mathit{mode} = \texttt{CO} \\
\qquad\qquad\qquad\quad \mathit{PI}(\text{"}\texttt{maxDuration}\text{"}).\mathit{mode} = \texttt{RO} \\
\qquad\qquad\qquad]
\end{array}
$$

At last we declare a factory, i.e. a concrete „driver“, which probably will be supported only on nodes of dedicated node classes[40].

We simply build a new schema which combines (1) the schema of the (most specific) actor class for which the factory is an implementation, (2) the general pre-defined *Actor_Factory* schema, and (3) maybe further constraints on the configuration parameters.

FACT_timer_x4711_2.2

AC_Timer_4711
Actor_Factory

$$
\begin{array}{lcl}
V\ \texttt{minResolution} & = & \mathit{sec}(0.001) \\
V\ \texttt{maxDuration} & < & \mathit{sec}(3600.00) \\
\mathit{PI}(\text{"}\texttt{curResolution}\text{"}).\mathit{mode} & = & \texttt{RW} \\
\mathit{PI}(\text{"}\texttt{curResolution}\text{"}).\mathit{default} & = & \mathit{sec}(0.01) \\
\multicolumn{3}{l}{\mathit{sec}(0.001) \leq V\ \texttt{curResolution} \leq \mathit{sec}(0.1)} \\
\multicolumn{3}{l}{\exists\, n : \mathbb{Z} \bullet V\ \texttt{curResolution} = n * \mathit{sec}(0.001)} \\
\multicolumn{3}{l}{V\ \texttt{maxDuration}\ /\ V\ \texttt{curResolution} \leq 2^{31} - 1}
\end{array}
$$

Please notice that this schema imposes real *dynamic constraints* on the configuration parameters' values, since the value of `maxDuration` varies depending on the value of `curResolution`.

3.10 Application Layer : Creation and Deletion of Actors

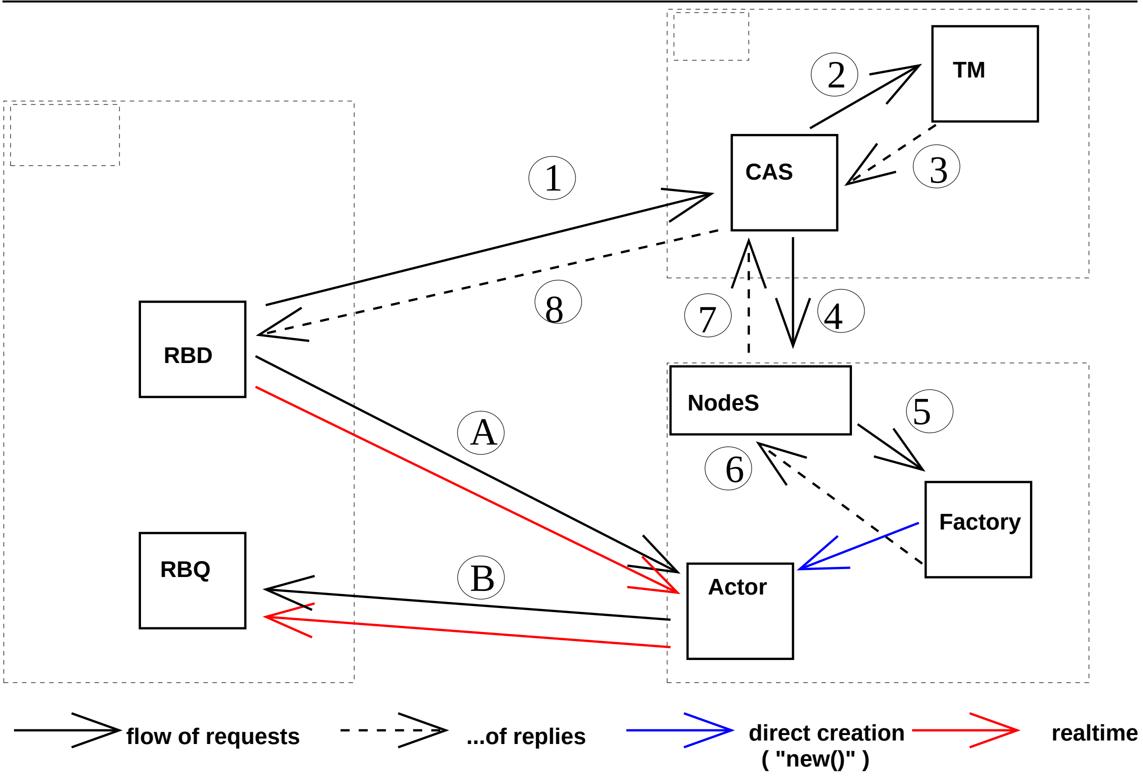
3.10.1 Principles of Operation (Non-Normative)

Different Notions of „Creation“ in the different „Semantic Spheres“. : In the **semantic sphere** of a programming language $\mathcal{L}$ the „creation“ of a new „object“ normally happens when evaluating an appropriate sentence from $\mathcal{L}$, like a TTCN-3 `create()`-statement, a JAVA `new`-expression, a C++ declaration of an automatic variable, etc. This evaluation returns a value of an abstract data type. This value then can be further processed by other statements provided by $\mathcal{L}$, — namely it can be stored into variables, and it can be passed as an argument value to functions of the appropriate type.

In the semantic sphere of the programming language $\mathcal{L}$ this value is really „abstract“. *Nothing is known* about this new object, beside its behavior as given *explicitly* by the

[40]The relation defining for which actor classes on which node classes which implementations exist, is not part of TCI. Instead it should be modeled in the „next higher“ level of related specifications, see section 5.2.3 below.

Figure 10 Collaboration Diagram for Creating a New Actor



pre-defined semantics of $\mathcal{L}$ and the applicable pre-defined or user-defined functions and procedures.

In the **semantic sphere** of „physical reality“ (e.g. some concrete hardware currently performing the execution of a program) the situation is quite different:

(1) Each dynamically created „implementing object“ (i.e. the object living in the implementation sphere, which corresponds to one abstract object from the language sphere) must be represented by an *address* . This address may be physical or symbolical, e.g. a code address, a driver entry point, a bus address, a handle for a method table (VTAB), etc.

(2) The implementing object is part of the physical reality and has many aspects and properties *beyond* the well-defined language semantics: A range of memory, which realizes an implementing object, may — since being something „physical“ — be swapped-in or swapped-out, aliased, write-protected, shared, damaged, etc.

(3) There can be very different kinds of implementations corresponding to the same abstract concept in the language sphere. E.g. a newly created `timer` object can either be mapped to a hardware timer (which is allocated from a limited set of such devices offered by a certain hardware node), – or can be realized by a piece of software. Between these variants an implementation can choose *invisibly* for the user.

In the **semantic sphere** of TCI, which is a kind of mediating *tertium comparationis*, the creation of an abstract object in the language sphere is realized by the construction and inventarization of a new actor subsystem of the appropriate actor class.

Operating transparently on remote actors : .
Performing *operations* on the abstract object in the sphere of the program must

be mapped onto some „physical“ operations on the implementing object. This must be supported in a transparent way, even when the `create()` statement had been performed on a remote node, and the reference to the abstract object has been passed from remote, e.g. as a message argument.

Therefore all references to (dynamically created) abstract objects must be translated to values of some communicable representation, in this section (locally!) referred to as „global identifiers“. Global identifiers, since being „absolute values“, can be embedded into bus messages and transmitted to a receiving node via some network infrastructure.

The receiving node has to re-translate each received global identifier into the hardware dependent implementing object. The last step can only be done on each node locally, using the node's internal knowledge. It yields the „physical address“ either of the „real“ object, which directly represents the abstract object from the language sphere, or of a „proxy object“, which supports the same interface, but maps most operations to messages sent via the network to the former.

To ensure this transparency w.r.t. deployment, TUB-TCI took the decisions — as mentioned above – to restrict the architecture to *one single central* instance each, for (1) deployment knowledge, (2) deployment decision, and (3) name-space administration.

> **Please notice** that (1) is totally contained in TCI, namely the CAS, while (2) is *external* to TCI, namely located in the TM. Please notice further that (3) is under control of CAS, not of TM.

The creation process of an actor : .

The creation of a new actor subsystems happens as follows (cf. fig 10) :

1. The source ATS has been compiled to an RBD, and each „`new()`“-like statement contained therein has been compiled to a `create()` service request, sent to CAS.
   The parameters of this request are (1) the indication of the *actor class* to create a new instance of, (2) a hint on which (physical) node the new actor should be realized, and (3) some actor class specific set constraints on the *configuration parameters* for the new actor subsystem.
2. The CAS now asks the TM to decide for the node on which the actor really will be created, based on the parameters (1) to (3) above and the *NodeIdent* of the originally requesting node[41].
3. If the TM can determine such a *hosting node*, its *NodeIdent* is returned, together with a new *cp-map*. The latter is needed, because the TM's knowledge of properties and idiosyncratics of the selected implementation may require additional information passed from TM to the factory. The new configuration parameter values calculated by TM may be more restrictive than the original parameter set, or even differ totally, and may be enriched by values for additional configuration parameters, only defined for the selected implementation.
   If no appropriate hosting node could be selected, an error indication is returned.
4. In TUB-TCI the global identifiers are realized by the values of type *LUID*.
   The CAS allocates a fresh *LUID* for the new actor, and passes a `DOcreate()` request to the hosting node. This request is parameterized with the *LUID* and the parameter set generated by the TM.
5. The NodeServer on the hosting node passes this request to the appropriate factory.

[41] In any case, on the conceptual level this *NodeIdent* is information „leaking“ from the trns layer, cf. section 2.9. On the implementation level this *NodeIdent* may be encoded in the corresponding message explicitely, or implicitly inquired from the trns layer.

6. The factory's creation method now can fulfill or reject the request.
7. This is signaled back to the CAS, which marks the new actor subsystem (as indexed by its *LUID*) either as `living` or as `error`.
8. In case of success a positive result is sent back to the original client, containing the *LUID* of the newly created actor subsystem, and the *NodeIdent* of its hosting node. By these coordinates the newly created actor is reachable as soon as (or even before) this reply message is received.

3.10.2 Creating Sub-Actors of an Actor

When a factory shall create an actor subsystem of some higher complexity, it may find it necessary to create *further* actors *implicitly*. Normally these additional actors (called „sub-actors" in this section) are needed to support the operation of the further (called „main actor"). E.g. a „`port`" actor could require the allocation of a dedicated „`timer`", — a „`messageFilter`" could require the creation of different „`codeConverters`", — some visualization device would need some „`graphicWindow`", etc. The advantage of such a sub-actor concept is, that these auxiliary subsystems can be configured and controlled by the standard means forseen by their actor class in general.

The approach chosen in TUB-TCI for supporting the creation of sub-actors can be called „local creation", — an alternative called „external creation" is sketched in section 5.1.3.

Here the factory (i.e. the `create` procedure in the factory's code) decides on its own, which actor classes must additionally be instantiated. The „physical" creation of the corresponding implementing subsystems is then performed locally and autarkicly. The factory simply *informs* the CAS concerning the new, additionally created sub-actors. By sending the service request message `registercreatesub`$(j, \mathsf{seq}\, \mathit{actorClass})$ to CAS, the latter is informed that the main actor with $\mathit{LUID} = j$ contains a sequence of sub-actors with the given actor classes.

Then the CAS allocates a new *LUID* for each position of that sequence, stores this id together with the given actor class, and remembers this *LUID* to represent an actor which is sub-actor of j. The *LUIDs* of the sub-actors are sent back to the factory by the message `subregistered`$(\mathsf{seq}\, \mathit{LUID})$.

The sub-actor relationship seems of practical relevance only if a `delete()` service will ever be requested from a sub-actor. Then the CAS can re-direct this request to the main actor.

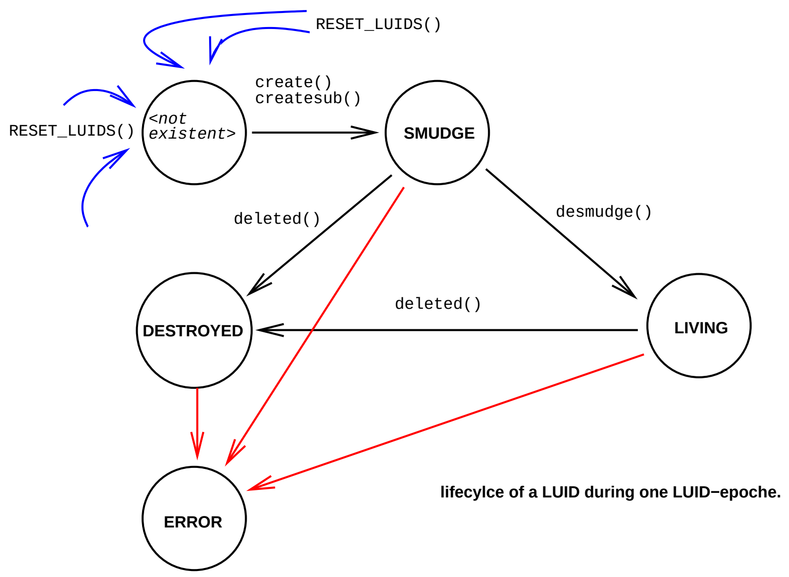
Please notice that the permission of configuration parameters of type *LUID* will make the creation of sub-actors controllable by the caller: Whenever the corresponding value passed with the `create()` request addresses an existing actor, then this device is taken as a sub-actor, — if the value is undefined, then a new sub-actor is created implicitly[42].

3.10.3 Messages

(76)

[42]The change allowance of this configuration parameter must be `WI` and a default of „`$autocreate`" could be provided!

Figure 11 Life Cycle of a LUID



$$ParameterValuePreference \;==\; Ident \nrightarrow \_DATA\_$$
$$\diamond \;:\; ParameterValuePreference \times Param \;==\; Ident \nrightarrow \_DATA\_$$

(77)

$NodePreference ::= \texttt{nodeset} \langle\langle \mathbb{P}\, NodeIdent \rangle\rangle \mid \texttt{self} \mid \texttt{any}$

$ParameterError ::= \texttt{parameterError} \langle\langle [\; requiredButMissing : \mathbb{P}\, Ident$
$readOnlyButGiven : \mathbb{P}\, Ident$
$illegalValue : \mathbb{P}\, Ident$
// (means illegal value *combination*!)
$] \rangle\rangle$

$ParameterFeedBack ::= \texttt{parameterDone} \langle\langle [\; changed : \mathbb{P}\, Ident$
$ignoredvalue : \mathbb{P}\, Ident$
$ignoredident : \mathbb{P}\, Ident$
$] \rangle\rangle$

$DeleteFeedBack ::= \texttt{deleted} \langle\langle\; \mathbb{P}\;\; LUID \;\rangle\rangle$

(78)

$$
\begin{array}{l}
\mathit{SRQS\_from\_RBD\_to\_CAS}\ [\ \boxed{\text{a=}}\mathit{ActorClass}, \mathit{ActorParams}\,[a]\ ]\ ::= \\
\qquad \texttt{create}\ \langle\!\langle\ [\ \ \mathit{class} : \mathit{ActorType} \\
\qquad\qquad\qquad \mathit{nodeSelect}\ :\ \mathit{NodePreference} \\
\qquad\qquad\qquad \mathit{params} : \mathit{ActorParams}\,[\mathit{class}]\ \ ]\rangle\!\rangle \\
\qquad\qquad \Rrightarrow\ \texttt{createsuccess}\ \langle\!\langle\ [\ \mathit{id} : \mathit{LUID} \\
\qquad\qquad\qquad\qquad \mathit{hostingNode} : \mathit{NodeIdent}\ \ ]\rangle\!\rangle \\
\qquad\qquad \Rrightarrow\ \texttt{createfailed}\ \langle\!\langle\ [\ \mathit{ParameterError}\ ]\rangle\!\rangle \\
\qquad \mid\ \texttt{delete}\ \langle\!\langle \mathit{LUID} \rangle\!\rangle \\
\qquad\qquad \Rrightarrow\ \mathit{DeleteFeedBack}
\end{array}
$$

(79)

$$
\begin{array}{l}
\mathit{SRQS\_from\_Trns\_to\_CAS}\ ::= \\
\qquad \texttt{lookup}\langle\!\langle \mathit{LUID} \rangle\!\rangle \\
\qquad\qquad \Rrightarrow\ \texttt{actorInfo}\ \langle\!\langle\ \mathit{NodeIdent}\ \times\ \mathit{ActorClass} \rangle\!\rangle \\
\qquad\qquad \Rrightarrow\ \texttt{unknownLuid}
\end{array}
$$

(80)

$$
\begin{array}{l}
\mathit{SRQS\_from\_CAS\_to\_Node}\ [\ \boxed{\text{a=}}\mathit{ActorClass}, \mathit{ActorParams}\,[a]\ ]\ ::= \\
\qquad\quad \texttt{DOcreate}\ \langle\!\langle\ [\ \ \mathit{class} : \mathit{ActorClass} \\
\qquad\qquad\qquad\quad \mathit{id} : \mathit{LUID} \\
\qquad\qquad\qquad\quad \mathit{params} : \mathit{ActorParams}\ [\mathit{class}] \\
\qquad\qquad\qquad\ \ ]\rangle\!\rangle \\
\qquad\qquad\quad \Rrightarrow\ \texttt{creationparamerror}\ \langle\!\langle\ [\ \mathit{ParameterError}\ ]\rangle\!\rangle \\
\qquad\qquad\quad \Rrightarrow\ \mathit{GF} \\
\qquad\qquad\quad \Rrightarrow\ \texttt{desmudge}\ \langle\!\langle\ \mathit{LUID}\ \times\ \mathbb{P}\ \mathit{LUID}\ \rangle\!\rangle \\
\qquad\quad \mid\ \texttt{DOdelete}\ \langle\!\langle\ \mathit{LUID} \times \mathit{LUID}\ \rangle\!\rangle \\
\qquad\qquad\quad \Rrightarrow\ \mathit{deleteFeedBack}
\end{array}
$$

(81)

$$
\begin{array}{l}
\mathit{SRQS\_from\_Factory\_to\_CAS}\ ::= \\
\qquad \texttt{registersubactors}\ \langle\!\langle\ \mathit{LUID}\ \times\ \mathsf{seq}\ \mathit{ActorClass}\ \rangle\!\rangle \\
\qquad\qquad \Rrightarrow\ \texttt{subactorsregistered}\ \langle\!\langle\ \mathsf{seq}\ \mathit{LUID}\ \rangle\!\rangle
\end{array}
$$

3.10.4 Additional State Space

(82)

$$\mathit{ActorPhase} ::= \texttt{nonexisting} \mid \texttt{smudge} \mid \texttt{living} \mid \texttt{destroyed} \mid \texttt{error}$$

(83)

NodeState

...
// These values are HARD WIRED into the implementation of TCI
// running on a given node :
$\mathit{findFactory} : \mathit{ActorClass} \nrightarrow \mathsf{Factory}$
...

(84)

Nodestate

...
$\mathit{actors} : \mathit{LUID} \nrightarrow (\boxed{a=}\mathit{ActorClass} \times \mathit{Node} \times \mathit{ActorParams}\ [a] \times \mathit{ActorPhase})$
...

// **Please notice** that the parameter settings stored in *actors* is just for diagnosis purpose. It helps to recognize an actor subsystem by relating it to the application of `create()`, which brought it into existence.

Here we store the „raw“ version as given by the user. We could instead/additionally store the „enriched“ version, as computed by TM.

(85)

CAS

...
$\mathit{isContainedIn} : \mathit{LUID} \nrightarrow \mathit{LUID}$
...

(86)

TM

...
//this function does need „strategic intelligence“, – nothing for TCI !
$\boxed{\texttt{->}}\ \mathsf{decideNode} : \mathit{NodeIdent} \times \boxed{A =}\mathit{ActorClass} \times \mathbb{P}\, A.\mathit{params} \times \mathit{NodePreference}$
$\nrightarrow (\ \mathit{NodeIdent} \times A.\mathit{params} \times \mathit{Duration}\ ) \cup \mathsf{dom}\ \texttt{createError}$

$(n, \_, \_) = \mathsf{decideNode}(o, \_, \_, N)$
$\Rightarrow\ n = n_0 \quad \wedge\ N = \{n_0\}$
$\vee\ n = o \quad \wedge\ N = \texttt{self}$
$\vee\ n \in N \quad \wedge\ N \subset \mathcal{C}.\mathit{nodes}$
$\vee\ n \in \mathcal{C}.\mathit{nodes} \quad \wedge\ N = \texttt{ANY}$

...

(87)

Actor_Factory

. . .
$\boxed{\texttt{->}}$ create : $LUID \times pack\ PI \rightarrow (GR \cup ParameterError)$
$\boxed{\texttt{->}}$ delete : $LUID \times LUID \rightarrow DeleteFeedBack$

. . .

3.10.5 $\boxed{\text{RBD}} \overset{3}{\Longrightarrow} \text{CAS}$

Here again we have an „architectural information leak“ (cf. section 2.9), since the information that a new actor has been successfully created should (of course !?) be memorized in the local *NodeState*. Then for this actor a future look-up from CAS will not be necessary.

$a : ActorClass$
$p : ActorParams\,[a]$
$q : \_\_DATA\_\_$
$N : NodePreference$
$n : NodeIdent$
$CREATETIMEOUT : Duration$

(88) // For sake of safety, fundamental calls are always evaluated as „expressions“
// e.g. called like `hdl = TCI_EVAL ( create(a,n,p) )`
$\mathcal{E}[[\mathcal{K}, K,\ "\texttt{(}"\ \texttt{create}\,(\,a, N, p\,)\ "\texttt{)}"\ ]]$
$\rightsquigarrow\ \mathcal{E}\,[[\mathcal{K}, K, \texttt{req}(\texttt{CAS}, \texttt{wait}(CREATETIMEOUT), \texttt{create}(a, N, p)\,)\ ]]$

(89)
$\mathcal{E}[[\mathcal{K}, \Delta K, \texttt{answer}(\texttt{createsuccess}\,(\,j, n\,)\,)$
$\quad |\quad j \notin K.actors$
$\quad \wedge\quad K'.actors(j)\ =\ (a, n, p, \texttt{living}))\ ]]$
$\rightsquigarrow\ \mathcal{E}[[\mathcal{K}, K,\ "\texttt{(}"\ j\ "\texttt{)}"\ ]]$

(90)
$\mathcal{E}[[\mathcal{K}, K,\ \texttt{answer}(m)$
$\quad |\quad m\ =\ \texttt{generalFailure}()$
$\quad \vee\quad m\ =\ \texttt{createfailed}(\texttt{parameterError}()\,)]]$
$\rightsquigarrow$ *// IMPLEMENTATION dependent error signaling*
// The LANGUAGE BINDING is free to select :
$//\,\alpha\ =\ \texttt{null}\ /\ \varepsilon\ /\ m\ /\ (e \in GF)$
$\rightsquigarrow\ \mathcal{E}[[\mathcal{K}, K\ ,\ \alpha\ ]]$

$\mathit{DELETETIMEOUT} : \mathit{Duration}$

---

(91)

$$\mathcal{E}\,[[\mathcal{K}, K, \texttt{delete}\,(j\,) \;\mid\; K.\mathit{actors}(j) \;=\; (\_,\_,\_,\texttt{living})\;]]$$
$$\leadsto\; \mathcal{E}\,[[\mathcal{K}, K, \texttt{req}(\texttt{CAS}, \texttt{wait}(\mathit{DELETETIMEOUT}), \texttt{delete}\,(j\,)\;)\;]]$$

(92)

$$\mathcal{E}[[\mathcal{K}, \Delta K, \texttt{answer}(\texttt{deleted}(J)\,)$$
$$\mid\; \forall\, x \in J \bullet K'.\mathit{actors}(x) \;=\; (\_,\_,\_,\texttt{destroyed}\,)\;]]$$
$$\leadsto\; \mathcal{E}\,[[\mathcal{K}, K, \varepsilon]]$$

(93)

$$\mathcal{E}[[\mathcal{K}, K, \texttt{answer}(\texttt{error()}\;)$$
$$\mid\; K'.\mathit{actors}(j) \;=\; (\_,\_,\_,\texttt{error}\,)\;]]$$
$$\leadsto\; //\; \mathit{IMPL\ DEPENDING\ ERROR\ SIGNALING}$$
$$\leadsto\; \mathcal{E}\,[[\mathcal{K}, K, \varepsilon]]$$

3.10.6 $\mathsf{RBD} \xRightarrow{3} \boxed{\mathsf{CAS}} \xRightarrow{2} \mathsf{NodeServer}$
 $\mathsf{Factory} \xRightarrow{5} \boxed{\mathsf{CAS}}$

$\mathcal{C} \;\in\; \mathsf{CAS}$

$\mathit{containsParts} == \mathcal{C}.\mathit{isContainedIn}^{\sim}$

$//$ information leak (cf. 2.9) : $o\!::\!i$ is decomposed and origin node identified !

(94)

$$\mathcal{E}[[\mathcal{K}, \Delta\mathcal{C}, \texttt{service}(o\!::\!i, \texttt{create}\,(t, N, p)\;)$$
$$\mid\;\; (n, q, T) \;==\; \mathcal{M}\boxed{\texttt{->}}\mathsf{decideNode}(\boxed{o}, t, p, N)$$
$$\wedge\;\; \mathcal{C}'.\mathit{actors}' = \mathcal{C}.\mathit{actors} \cup \{j \mapsto (t, n, p, \texttt{smudge})\}\;\;]]$$
$$\leadsto\; \mathcal{E}[[\mathcal{K}, \Delta\mathcal{C}, \texttt{req}(n, \texttt{wait}(T), \texttt{DOcreate}(t, j, q)\;)\;]]$$

(95)

$$\mathcal{E}[[\mathcal{K}, \Delta\mathcal{C}, \texttt{answer}(\texttt{desmudge}\,(j, J)\;)$$
$$\mid\;\; \mathit{containsParts}^{*}\;(\!|\{j\}|\!)\;\subset\; J$$
$$\wedge\;\; \forall\; x \in J \bullet \mathcal{C}'.\mathit{actors}(x).4 \;=\; \texttt{living}\;\;]]$$
$$\leadsto\; \mathcal{E}\,[[\mathcal{K}, \mathcal{C}, \texttt{reply}(o\!::\!i, \texttt{createsuccess}(j, \mathit{actors}(j).2)\;)\;]]$$

(96)

$$\mathcal{E}[[\mathcal{K}, \Delta\mathcal{C}, \texttt{answer}(\texttt{deleted}\,(J, \mathit{txt})\;)$$
$$\mid\;\; \mathit{containsParts}^{*}\;(\!|\{j\}|\!)\;\subset\; J$$
$$\wedge\;\; \forall\; x \in J \bullet \mathcal{C}'.\mathit{actors}(x).4 \;=\; \texttt{destroyed}\;]]$$
$$\leadsto\; \mathcal{E}\,[[\mathcal{K}, \mathcal{C}, \texttt{reply}(o\!::\!i, \texttt{failure}(\texttt{"could not create"}, \mathit{txt}\,)\,)\;]]$$

(97)

$$\mathcal{E}[[\mathcal{K}, \mathcal{C}, \texttt{service}(o\!::\!i, \texttt{registersubactors}\,(j, A)\;)\;]]$$
$$\leadsto\; \mathcal{E}[[\mathcal{K}, \mathcal{C}, \texttt{reply}(o\!::\!i, \texttt{subactorsregistered}(J)\;)$$
$$\mid\;\; A : \mathsf{seq}\;\mathit{ActorClass}$$
$$\wedge\;\; J : \mathsf{seq}\;\mathit{LUID}$$
$$\wedge\;\; \#A \;=\; \#J$$
$$\wedge\;\; \forall\, x \in \{1 \ldots \#A\} \;\bullet\; (\;J.x \;\notin \mathsf{dom}\;\mathcal{C}.\mathit{actors}$$
$$\wedge\,(J.x \mapsto (A.x, o, \bot, \texttt{smudge}) \;\in\; \mathcal{C}'.\mathit{actors}$$
$$\wedge\,(J.x \mapsto j) \;\in\; \mathcal{C}'.\mathit{isContainedIn}\;\;)\;\;]]$$

$\mathcal{C} \in \mathsf{ConfigurationControl}$
$\mathit{DELETETIMEOUT} : \mathit{Duration}$

(98)
$$\begin{aligned}
&\mathcal{E}[[\mathcal{K}, \mathcal{C}, \texttt{service}\,(\, o{::}i, \texttt{delete}\,(j'))\\
&\qquad\qquad \mid\ j' \mapsto (\_, n, \_, \texttt{living}) \in \mathcal{C}.\mathit{actors}\\
&\qquad\qquad \wedge\ (\mathit{ID}[\mathit{LUID}] \cup \mathcal{C}.\mathit{isContainedIn}\,)^* \ (\!|\{ j' \}|\!) \ = \ \{j\} \quad]]\\
\rightsquigarrow\ &\mathcal{E}[[\mathcal{K}, \mathcal{C}, \texttt{req}\,(n, \texttt{wait}(\mathit{DELETETIMEOUT}), \texttt{DOdelete}\,(j, j')\,)\,]]
\end{aligned}$$

(99)
$$\begin{aligned}
&\mathcal{E}\,[[\mathcal{K}, \mathcal{C}, \texttt{answer}(M) \ \mid\ M \ = \ \texttt{deleted}(J, \text{"}\texttt{OK}\text{"}) \ \wedge\ j' \ \in\ J\,]]\\
\rightsquigarrow\ &\mathcal{E}[[\mathcal{K}, \Delta\mathcal{C},\ \texttt{reply}\,(M)\\
&\qquad\qquad \mid\ \ \forall\, x \in J \bullet \mathcal{C}'.\mathit{actors}(x).4 = \texttt{destroyed}\\
&\qquad\qquad \wedge\ \ \mathcal{C}'.\mathit{isContainedIn} \ = \ J \lhd \mathcal{C}.\mathit{isContainedIn}\ \ ]]
\end{aligned}$$

(100)
$$\begin{aligned}
&\mathcal{E}\,[[\mathcal{K}, \mathcal{C}, \texttt{answer}(\texttt{deleted}(J, \mathit{txt})\,)\ \mid\ j' \ \notin\ J\,]]\\
\rightsquigarrow\ &\mathcal{E}[[\mathcal{K}, \Delta\mathcal{C},\ \texttt{reply}\,(\texttt{error}(\text{"}\texttt{could not delete}\text{"}, \mathit{txt}, J)\\
&\qquad\qquad \mid\ \ \forall\, x \in J \bullet \mathcal{C}'.\mathit{actors}(x).4 = \texttt{destroyed}\\
&\qquad\qquad \wedge\ \ \mathcal{C}'.\mathit{isContainedIn} \ = \ J \lhd \mathcal{C}.\mathit{isContainedIn}\ \ ]]
\end{aligned}$$

(101)
$$\begin{aligned}
&\mathcal{E}\,[[\mathcal{K}, \mathcal{C}, \texttt{answer}(M) \ \mid\ M \ \in\ \mathsf{ran}\,\mathit{GF}\ ]]\\
\rightsquigarrow\ &\mathcal{E}\,[[\mathcal{K}, \mathcal{C}, \texttt{reply}(o{::}i, M)\ ]]
\end{aligned}$$

3.10.7 $\mathsf{CAS}\xrightarrow{2}\boxed{\mathsf{Node}}\xrightarrow{7}\mathsf{Factory}$

// for sake of „authorization“ :
$o = NodeIdent(\texttt{CAS})$

(102)
$$\mathcal{E}[[\mathcal{K}, K, \texttt{service}(o{::}i, \texttt{DOcreate}\,(a,j,p,q)\,)\quad |\quad j \notin \mathsf{dom}\ K.actors\quad F\ ==\ K.findActorFactory(a)\ \wedge\ F \neq \bot$$
$$\rightsquigarrow\ \mathcal{E}[[\mathcal{K}, K, AWAIT\ (\ F\ \boxed{\texttt{->}}\ \mathsf{createMethod}\ (j,p,q)\ )\ ]]$$

(103)
$$\mathcal{E}[[\mathcal{K}, \Delta K, AWAIT\ M\quad |\quad M = (\ \texttt{desmudge}(J)\ )\quad |\quad \forall\, x \in J \bullet K.actors(x).4 = \texttt{smudge}\ \wedge\ K'.actors(x).4 = \texttt{living}\ ]]$$
$$\rightsquigarrow\ \mathcal{E}[[\mathcal{K}, \Delta K, \texttt{reply}(M)\ ]]$$

(104)
$$\mathcal{E}[[\mathcal{K}, \Delta K, AWAIT\ M\quad |\quad M = (\ \texttt{deleted}(J, txt)\ )\quad |\quad \forall\, x \in J \bullet K'.actors(x).4 = \texttt{destroyed}\ \ ]]$$
$$\rightsquigarrow\ \mathcal{E}[[\mathcal{K}, \Delta K, \texttt{reply}(M)\ ]]$$

(105)
$$\mathcal{E}[[\mathcal{K}, K, \texttt{service}(o{::}i, \texttt{DOcreate}\,(a,j,p)\,)\quad |\quad j \notin \mathsf{dom}\ K.actors\quad K.FindActorClass\,(a)\ = \bot\quad]]$$
$$\rightsquigarrow\ \mathcal{E}[[\mathcal{K}, \Delta K, \texttt{reply Error}(\text{"}\texttt{noFactoryKnown}\text{"})\quad |\quad (j\ \mapsto\ (a, K, p, \texttt{error}))\ \in\ K'.actors\ ]]$$

(106)
$$\mathcal{E}[[\mathcal{K}, K, \texttt{service}(o{::}i, \texttt{DOcreate}\,(a,j,p)\,)\quad |\quad j\ \in\ \mathsf{dom}\ K.actors\ \ ]]$$
$$\rightsquigarrow\ \mathcal{E}[[\mathcal{K}, \Delta K, \texttt{reply}(\texttt{Error}(\text{"}\texttt{LUIDinuse}\text{"}))\quad |\quad (j\ \mapsto\ (a, K, p, \texttt{error}))\ \in\ K'.actors\ ]]$$

(107)
$$\mathcal{E}[[\mathcal{K}, K, \texttt{service}(o{::}i, \texttt{DOdelete}(j,k)\ )\quad |\quad j,k\ \in\ \mathsf{dom}\ K.actors\ \wedge\ (j = k\ \vee\ \mathcal{C}.isPartOf\,(k,j)$$
$$\rightsquigarrow\ \mathcal{E}[[\mathcal{K}, K, AWAIT\ (\ F\ \boxed{\texttt{->}}\ \mathsf{deleteMethod}\ (j,k)\ )\ ]]$$

(108)
$$\mathcal{E}[[\mathcal{K}, \Delta K, AWAIT\ M\quad |\quad M = (\ \texttt{deleted}(J)\ )\quad |\quad \forall\, x \in J \bullet\ K'.actors(x).4 = \texttt{destroyed}\ ]]$$
$$\rightsquigarrow\ \mathcal{E}[[\mathcal{K}, \Delta K, \texttt{reply}(M)\ ]]$$

(109)
$$\mathcal{E}[[\mathcal{K}, \Delta K, AWAIT\ M\quad |\quad M\ \in\ \mathsf{ran}\ GF\ ]]$$
$$\rightsquigarrow\ \mathcal{E}[[\mathcal{K}, \Delta K, \texttt{reply}(M)\ ]]$$

3.11 Application Layer : Operation on Actors

3.11.1 Principles of Operation (Non-Normative)

Each actor subsystem offers three kinds of services: (1) changing and (2) inquiring of its configuration parameters, and (3) execution of run-time operations. Service (2) must expect a reply messages, the other two may be called unconfirmed.

As described above, there is one single definition of configuration parameters for each actor class. This definitions maps identifiers to value types, default values and parameter change allowance modes. These configuration parameters are addressed by services (1) and (2).

The valid data (= „commands") for the run-time operation service (3) is defined with the actor class by a free data type, separately.

The procedure for identifying and addressing an actor is as follows:

- Any successful `create(`$t, n, p,$`)` request is answered by a `createsuccess(`j, N`)` reply, which returns j as the *LUID* for the freshly created subsystem, and N as the *NodeIdent* of its hosting node.
  Every successive `setconfigparam()`, `getconfigparam()` or `rtOperation()` request operating on j must be sent via trns to the hosting node N.
- In case that the RBD has received the *LUID* identifying the actor from remote, e.g. as a message parameter, the hosting node may not be known. Then its *NodeIdent* will be fetched *once* from CAS. This is done by a `lookUp()` service request generated automatically by the trns subsystem[43].
- When the *SRQ* message finally reaches the hosting node of the actor, the implementation of TUB-TCI must decode the message and pass the call to the appropriate implementing code[44].

3.11.2 Messages

(110)

```
SRQS_from_RBD_to_Actors [ ActorParams [ ActorClass ] ]  ::=
            setconfigparams ⟨⟨[ actid : LUID
                                atomic : BOOL
                                data : Ident ⇸⇸ ActorParams
                                ]⟩⟩
                ⇛paramerror()

            getconfigparams ⟨⟨[ actid : LUID ]⟩⟩
                ⇛paramdump⟨⟨Ident ⇸⇸ ActorParams⟩⟩

            rtOperation ⟨⟨[ actid : LUID ]⟩⟩
                ⇛RTvalue⟨⟨__DATA__⟩⟩
```

[43] Depending on the concrete application scenario, these mapping informations could be calculated „statically" at compile time or link time. Then they could be downloaded to all nodes in advance, i.e. before starting the test case performance, thus eliminating any delay for looking up hosting nodes during TRun. The current version of TUB-TCI does not foresee any means for such an approach.

[44] A possible method for automating this mapping is discussed in section 5.2.3 as an extension.

3.11.3 Additional State Space

(111)

$$
\begin{array}{|l}
\textit{NodeState} \\
\hline
\ldots \\
\textit{// Convenience functions:} \\
\mathit{classOf} : \mathit{LUID} \pfun \mathit{ActorClass} \\
\mathit{driverFor} : \mathit{LUID} \pfun \mathit{Factory} \\
\hline
\mathit{classOf}(j) = a \quad \Leftrightarrow \quad (j \mapsto (a, \_, \_, \_)) \in \mathit{actors} \\
\mathit{driverFor}(j) = \mathit{findFactory}(\mathit{classOf}(j)) \\
\ldots \\
\hline
\end{array}
$$

(112)

$$
\begin{array}{|l}
\textit{Factory} \\
\hline
\ldots \\
\boxed{\texttt{->}}\ \mathsf{changeParamas} \ : \ \mathit{LUID} \ \times \ \mathit{pack} \ \mathit{PI} \ \rightarrow \ \mathit{ParamFeedBack} \\
\boxed{\texttt{->}}\ \mathsf{dumpParamas} \ : \ \mathit{LUID} \ \rightarrow \ \mathit{pack} \ \mathit{PI} \\
\boxed{\texttt{->}}\ \mathsf{executeRtOperation} \ : \ \mathit{LUID} \ \times \ \mathit{RT} \ \rightarrow \ \_\_\mathit{DATA}\_\_ \\
\ldots \\
\hline
\end{array}
$$

3.11.4 $\boxed{\textsf{RBD}}\overset{8}{\Longrightarrow}\textsf{Actor}$: Configuration Alteration, Status Enquiring and Run-Time Actions of Actors

$\square_x \in\ X\ \ == \ \{\texttt{setconfigparam},\ \texttt{rtOperation}\}$
$\square_y \in\ Y\ \ == \ X\ \cup\ \{\texttt{getconfigparam}\ \}$
$M_x\ \ == \ \square_x(j,\ldots)$
$M_y\ \ == \ \square_y(j,\ldots)$
$j : LUID$

(113)
$$\mathcal{E}[[\mathcal{K}, K, M_X\ \texttt{";"}\ \mid\ (j \mapsto (N,\_,\_,\texttt{living})) \in K.actors\ ]]$$
$$\rightsquigarrow\ \mathcal{E}\,[[\mathcal{K}, K, \texttt{req}(N, \texttt{\$noresp}, M)]]$$
(114)
$$\mathcal{E}[[\mathcal{K}, K, \texttt{"("}\ \ M_Y\ \ \texttt{")"}\ \mid\ (j \mapsto (N,\_,\_,\texttt{living})) \in K.actors\ ]]$$
$$\rightsquigarrow\ \mathcal{E}\,[[\mathcal{K}, K, \texttt{req}(N, \texttt{wait}, M_Y\ )\ ]]$$
(115)
$$\mathcal{E}\,[[\mathcal{K}, K, \texttt{answer}(m)]]$$
$$\rightsquigarrow\ \mathcal{E}\,[[\mathcal{K}, K, \texttt{"("}\ \ m\ \texttt{")"}\ ]]$$

$$\mathcal{E}[[\mathcal{K}, K, \left\{\texttt{"("}\right\}\ M_X\ \left\{\begin{matrix}\texttt{")"}\\ \texttt{";"}\end{matrix}\right\},\quad \mid\quad (j \mapsto (N,\_,\_,x)) \in K.actors \quad \wedge\quad x \neq \texttt{living}\ \ ]]$$
(116)
// IMPLEMENTATION dependent error recovery
// The LANGUAGE BINDING is free to select :
// $\alpha\ = \texttt{null}\ /\ \varepsilon\ /\ m\ /\ (e \in \textsf{ran}\ GF)$
$$\rightsquigarrow\ \mathcal{E}\,[[\mathcal{K}, \Delta K, \alpha\ \mid\ (j \mapsto (a,\_,\_,\texttt{error}))\ \in K'.actors\ ]]$$

3.11.5 $\boxed{\textsf{trans}}\overset{4}{\Longrightarrow}\textsf{CAS}$: Looking up Actors

$\Box \in \{\texttt{setconfigparam}, \texttt{getconfigparam}, \texttt{rtOperation}, \ldots, \texttt{delete}, \texttt{enquire}, \texttt{connect}, \ldots\}$
$M \;==\; \Box\,(\,j, \ldots)$
$J \;:\; LUID$

(117)
$$\mathcal{E}[[\mathcal{K}, K, M \;\mid\; (j \;\mapsto\; (\_,\_,\_,X)) \in \; K.actors \;\wedge\; X \in \{\texttt{destroyed}, \texttt{error}\}]]$$
$$\rightsquigarrow \; ERROR$$

(118)
$$\mathcal{E}[[\mathcal{K}, K, M \;\mid\; j \notin \textsf{dom}\; K.actors]]$$
$$\rightsquigarrow \; \mathcal{E}\,[[\mathcal{K}, K, \texttt{req}(\texttt{CAS}, \texttt{wait}, \texttt{lookup}\,(j)\;)\;]]$$

(119)
$$\mathcal{E}\,[[\mathcal{K}, K, \texttt{answer}(\texttt{nodeinfo}\,(N, a)\,)\;]]$$
$$\rightsquigarrow \; \mathcal{E}\,[[\mathcal{K}, \Delta K, M \;\mid\; (j \mapsto (a, \_, N, \texttt{living})\,) \;\in K'.actors\;]]$$

(120)
$$\mathcal{E}\,[[\mathcal{K}, K, \texttt{answer}(m) \;\mid\; m \;=\; \texttt{actorinfo}(\texttt{unknown})\;]]$$
$$\rightsquigarrow \; ERROR$$

(121)
$ERROR \;==$ *// IMPLEMENTATION dependent error recovery*
// The LANGUAGE BINDING is free to select :
$// \alpha \;=\; \texttt{null} \;/\; \varepsilon \;/\; m \;/\; (e \in \textsf{ran}\; GF)$
$$\rightsquigarrow \; \mathcal{E}\,[[\mathcal{K}, \Delta K, \alpha \;\mid\; (j \mapsto (a, \_, \_, \texttt{error})\,) \;\in K'.actors\;]]$$

3.11.6 $\textsf{trans}\overset{4}{\Longrightarrow}\boxed{\textsf{CAS}}$: Looking up Actors

(122)
$$\mathcal{E}[[\mathcal{K}, C, \texttt{service}\,(\,o{::}i, \texttt{lookup}(j)\;) \;\mid\; (j \;\mapsto\; (N, a, \_, \texttt{living})\,) \;\in\; C.actors\;]]$$
$$\rightsquigarrow \; \mathcal{E}\,[[\mathcal{K}, C, \texttt{reply}(o{::}i, \texttt{nodeinfo}(N, a)\;]]$$

(123)
$$\mathcal{E}[[\mathcal{K}, C, \texttt{service}\,(\,o{::}i, \texttt{lookup}(j)\;) \;\mid\; j \notin \textsf{dom}\; C.actors \;\vee\; C.actors\,(j)\,.4 \;\neq \texttt{living}\;]]$$
$$\rightsquigarrow \; \mathcal{E}\,[[\mathcal{K}, C, \texttt{reply}(o{::}i, \texttt{unknownLuid})\;]]$$

3.11.7 $\textsf{RBD}\overset{8}{\Longrightarrow}\boxed{\textsf{Actor}}$: Configuration Alteration, Status Enquiring and Run-Time Actions of Actors

$$\frac{\begin{array}{l}(\ j \mapsto (K, a, p, \texttt{living})\ )\ \in\ K.actors \\ D = K.driverFor(j)\ \wedge\ D \neq \bot\end{array}}{}$$

(124)

$$\begin{array}{rl} & \mathcal{E}[[\mathcal{K}, K, \texttt{service}\ (\ o,\ i,\ \texttt{setconfigparam}\ (j, L, m)\ ) \\ & \qquad\qquad |\ r\ ==\ D\boxed{\texttt{->}}\ \mathsf{changeParams}\ (\boxed{\mathsf{j}}, L, m)\quad]] \\ \rightsquigarrow & \mathcal{E}\ [[\mathcal{K}, K, \texttt{reply}(\ o,\ i,\ r\ )]] \end{array}$$

(125)

$$\begin{array}{rl} & \mathcal{E}[[\mathcal{K}, K, \texttt{service}\ (\ o,\ i,\ \texttt{getconfigparam}\ (j)\ ) \\ & \qquad\qquad |\ r\ ==\ D\boxed{\texttt{->}}\ \mathsf{dumpParams}\ (\boxed{\mathsf{j}})\quad]] \\ \rightsquigarrow & \mathcal{E}\ [[\mathcal{K}, K, \texttt{reply}(\ o,\ i,\ r\ )]] \end{array}$$

(126)

$$\begin{array}{rl} & \mathcal{E}[[\mathcal{K}, K, \texttt{service}\ (\ o,\ i,\ \texttt{rtOperation}\ (j, q, r)\ ) \\ & \qquad\qquad |\ r\ ==\ D\boxed{\texttt{->}}\ \mathsf{executeRtOperation}\ (\boxed{\mathsf{j}}, q, r)\quad]] \\ \rightsquigarrow & \mathcal{E}\ [[\mathcal{K}, K, \texttt{reply}(\ o,\ i,\ r\ )]] \end{array}$$

3.12 Application Layer : QAS and Push-Channels for `Value_Events()` delivered from Producersto RBQs

3.12.1 Principles of Operation (Non-Normative)

Any hardware devices which is capable of generating or detecting „externally determined events“ (asynchronously and spontaneuosly) is represented by a producer subsystem. Whenever the hardware device detects the corresponding situation (e.g. a timer at expiration, an incoming datum at a port, a pushed button), the representing producer subsystem sends a `value_event()` message to the QAS subsystem.

The *drains*, to which the event signal shall finally be delivered, are subsystems of category RBQ, i.e. they offer the `rtOperation(putQ())` run-time operation service.

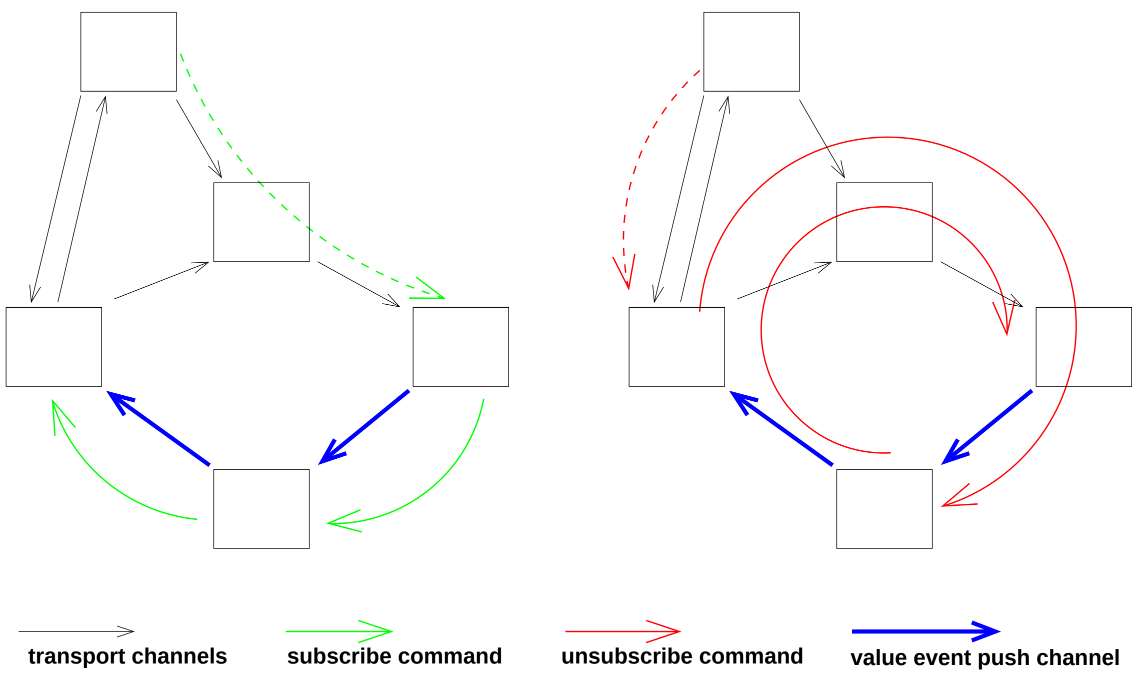
To minimize the load on the RBQ implementations and the system traffic, each RBQ must explicitely *subscribe* for a producer to receive the `value_events()` generated by it. Since therefore the stream of signals is under program-control, and may change dynamically during run-time, the `value_events()` cannot be delivered from the producer to the RBQs statically. The QAS subsystem distributes the `value_event()` messages to all currently subscribed RBQs, and to those remote nodes which host further drains.

The subscription process is complicated because, (1) it must consider that the transport layer communication lines need not to be bidirectional, cf. figure 12: The request to `subscribe()` a certain source has firstly to be sent to the node which hosts the source. Then starting from there, it has to be passed along the chain of all nodes on the *physical* communication line, as given by the *routing* information. This induces another complication, as (2) this is a typical example of „architectural information leak“ (cf. above 2.9 on page 24): all information from the „physical“ layer is normally strictly hidden from all application logic.

The protocol realizes the subscription from producer to RBQ by a chain of low-level subscriptions from *node* to *node*. Therefore all nodes in the physical chain from the node hosting the source to the node hosting the target, are „aware“ of the signal flow. This has the benefit that each new subscription needs only to append the missing last segments to a maybe already existing push-channel in the neighborhood.

The `unsubscribe()` mechanism also has an „information leak“, as during the sub-

Figure 12 Complex Situation for `subscribe()` and `unsubscribe()`



scription process the origin node of the (technical) `DOsubscribe()` message is memorized. This node is identical with the node sending the run-time `value_events()`, and its identity is required later to parameterize the `DOunsubscribe()` message.

3.12.2 Messages

(127)

$$SRQS\_RBD\_to\_QAS ::= \texttt{subscribe} \langle\!\langle [\ target : LUID;\ source : LUID\ ] \rangle\!\rangle \mid \texttt{unsubscribe} \langle\!\langle [\ target : LUID;\ source : LUID\ ] \rangle\!\rangle$$

(128)

$$SRQS\_Producers\_to\_QAS ::= \texttt{value\_event} \langle\!\langle [\ id : LUID\quad data : DATA] \rangle\!\rangle$$

// **Please notice** that the interface $\boxed{\text{QAS}} \overset{6}{\Longrightarrow} \boxed{\text{RBQ}}$ is already existing, implicitly realized by the service `rtOperation(putQ())`.
// Note for TTCN-3 users :
The „timeout“ event of a „timer subsystem“ is just an ordinary sub-type of the normal run-time `value_event()`.
The „timeout“ result of a remote procedure call is generated by TUB-TCI trns layer.

3.12.3 Additional State Space

(129)

NodeState

$\ldots$

$subscribedBy \; : \; LUID \;\twoheadrightarrow\!\!\!\!\!\!+\; \mathbb{P}\,(NodeIdent \cup LUID)$

$\ldots$

$\forall \; x \; \in \; (LUID \; \cap \; \bigcup \; \mathsf{ran} \; subsriptedBy)$

$\quad \bullet \; (x \; \mapsto (ownIdent, \_, \_, \mathtt{living})) \; \in \; actors \; \wedge classOf(x) \sqsupseteq \mathsf{RBQ}$

(130)

NodeState_Init

$\ldots$

$subscribedBy \; = \; \{\}$

$\ldots$

(131)

Factory_supporting_Producer

Actor_Factory

$\boxed{\texttt{->}}$ $\mathsf{streamControl}\;(\{\mathtt{on}, \mathtt{off}\})$
$\boxed{\texttt{->}}$ $\mathsf{watchHdw}$
$\boxed{\texttt{->}}$ $\mathsf{detectedExtEvent}\;(\_\_DATA\_\_)$

(132)

Factory_supporting_RBQ

Actor_Factory

$\mathsf{dom}\; \mathtt{putQ}(\_\_DATA\_\_) \;\; \subset \;\; \mathsf{dom} \;\; \mathsf{RT}$

$//\mathsf{dom}\; \mathtt{putValueEvent}(\_\_DATA\_\_) \;\; \subset \;\; \mathsf{dom} \;\; \mathsf{RT}$
$//RT \;\; ::= \;\; \mathtt{putValueEvent}(\_\_DATA\_\_)$

3.12.4 $\mathsf{producer} \xrightarrow{11} \boxed{\mathsf{QAS}} \xrightarrow{6} \mathsf{RBQ}$: Signaling Run-Time Events

$j, A \ : \ LUID$
$M \ : \ \_\_DATA\_\_[K.classOf(j)]$
$D == K.driverFor(j)$
$N \ : \ NodeIdent$

(133)
$\mathcal{E}[[\mathcal{K}, K, D \boxed{\texttt{->}} \mathsf{streamControl}(\texttt{on}) \ ]]$
$\leadsto \ \mathcal{E}[[\mathcal{K}, K, \varepsilon \ \|\!\!\vdash \ D \boxed{\texttt{->}} \mathsf{watchHdw} \ ]]$

(134)
$\mathcal{E}[[\mathcal{K}, K, \ D \boxed{\texttt{->}} \mathsf{watchHdw} \ \| \ D \boxed{\texttt{->}} \mathsf{streamControl}(\texttt{off}) \ ]]$
$\leadsto \ \mathcal{E}[[\mathcal{K}, K, \varepsilon \ ]]$

(135)
$\mathcal{E}[[\mathcal{K}, K, D \boxed{\texttt{->}} \mathsf{watchHdw} \ ]]$
$\leadsto \ \mathcal{E}[[\mathcal{K}, K, D \boxed{\texttt{->}} \mathsf{DetectExtEvent}(M) \ ]]$
$\leadsto \ \mathcal{E}[[\mathcal{K}, K, D \boxed{\texttt{->}} \mathsf{watchHdw} \ \|\!\!\vdash \ \texttt{value\_event}\,(j, M) \ ]]$

(136) // A locally generated event is distributed :
$\mathcal{E}[[\mathcal{K}, K, \texttt{value\_event}(j, M)$
$\qquad | \quad (j \mapsto (\_, N, \_, \texttt{living})\,) \in \ K.actors$
$\qquad \wedge \quad S == K.subscribedBy(j) \ \wedge \ S \neq \{\} \ \ ]]$
$\leadsto \ \mathcal{E}\,[[\mathcal{K}, K, \mathsf{DELIVERVALUEEVENT}\,(S, j, M) \ ]]$

(137) // Distribution to other *Nodes*:
$\mathcal{E}\,[[\mathcal{K}, K, \ \mathsf{DELIVERVALUEEVENT} \ (\{N\} \cup S, j, M \ ) \ ]]$
$\leadsto \ \mathcal{E}\,[[\mathcal{K}, K, \ \mathsf{DELIVERVALUEEVENT} \ (S, j, M) \ \|\!\!\vdash \ \texttt{req}(N, \texttt{\$noresp}, \texttt{value\_event}(j, M) \ ) \ ]]$
// `$noresp` *makes that TRNS layer will expand to* ε*,*
// *and the new thread will disappear again.*

(138) // Distribution to *local* subscribers :
$\mathcal{E}\,[[\mathcal{K}, K, \ \mathsf{DELIVERVALUEEVENT} \ (\{A\} \cup S, j, M \ ) \ ]]$
$\leadsto \ \mathcal{E}[[\mathcal{K}, K, \ \mathsf{DELIVERVALUEEVENT} \ (S, j, M)$
$\qquad \|\!\!\vdash \ D \boxed{\texttt{->}} \mathsf{execRtOperation} \ (j, \texttt{pushq}(m)) \ \ ]]$

(139)
$\mathcal{E}\,[[\mathcal{K}, K, \ \mathsf{DELIVERVALUEEVENT} \ (\{\} \ , \_, \_)]]$
$\leadsto \ \mathcal{E}\,[[\mathcal{K}, K, \ \varepsilon \ ]]$

(140)
$\mathcal{E}[[\mathcal{K}, K, \texttt{service} \ (o{::}i, \texttt{value\_event}\,(j, M)\,) \ ]]$
$\leadsto \ \mathcal{E}\,[[\mathcal{K}, K, \mathsf{DELIVERVALUEEVENT}\,(K.subscribedBy(j), j, M)]]$

3.12.5 $\mathsf{RBD}\overset{10}{\Longrightarrow}\boxed{\mathsf{QAS}}$: Subscribe and unsubscribe Signal Stream

n_T, n_S : $LUID$
k_K, k_X, k_S, k_T : $NodeIdent$
$(n_S \mapsto (a_S, k_S, \_, \mathtt{living})) \in K.actors$
$(n_T \mapsto (a_T, k_T, \_, \mathtt{living})) \in K.actors$
$a_S \sqsubseteq RBD\_Producer$
$a_T \sqsubseteq RBQ$
// Every subsystem serving as source/drain must be
// of category producer/RBQ.
X : $NodeIdent \cup LUID$

---

(141) *// If source and target both local : Try to start the device.*
$\mathcal{E}[[\mathcal{K}, K, \mathtt{subscribe}(n_T, n_S)$
$\quad | \quad k_T = K.ownIdent \wedge k_S = K.ownIdent \quad]]$
$\rightsquigarrow \mathcal{E}\,[[\mathcal{K}, K, \mathsf{CHECKSTART}(n_T, n_S)]]$

(142) *// If source is already subscripted : Just memorize new target.*
$\mathcal{E}[[\mathcal{K}, \Delta K, \mathtt{subscribe}(n_T, n_S)$
$\quad | \quad k_T = K.ownIdent \wedge K_S \neq K.ownIdent$
$\quad \wedge \quad K.subscribedBy(n_S) \neq \{\}$
$\quad \wedge \quad K'.subscribedBy(n_S) = K.subscribedBy(n_S) \cup \{n_T\} \quad]]$
$\rightsquigarrow \mathcal{E}\,[[\mathcal{K}, K, \varepsilon]]$

(143) *// If target is remote, or source remote and unsubscribed :*
// Start whole subscribing chain below!
$\mathcal{E}[[\mathcal{K}, K, \mathtt{subscribe}(n_T, n_S)$
$\quad | \quad k_T \neq K.ownIdent$
$\quad\quad \vee \; (\; k_S \neq K.ownIdent \wedge K.subscribedBy\,(n_S) = \{\}\,) \;]]$
$\rightsquigarrow \mathcal{E}\,[[\mathcal{K}, K, \mathtt{req}(k_S, \mathtt{\$noresp}, \mathtt{DOsubscribe}(n_T, n_S, \bot)\;)\;]]$

(144) *// Device has not subscribed yet : start it via Factory/Driver :*
$\mathcal{E}[[\mathcal{K}, \Delta K, \mathsf{CHECKSTART}(X, n_S)$
$\quad | \quad K.subscribedBy(n_S) = \{\}$
$\quad \wedge \quad K'.subscribedBy(n_S) = \{X\}$
$\quad \wedge \quad D = K.findDriver(K.classof\,(n_S)) \quad]]$
$\rightsquigarrow \mathcal{E}\,\big[[\mathcal{K}, K, \Delta\, D\, \boxed{\texttt{->}}\, \mathsf{streamControl}(\mathsf{on})]\big]$

(145) *// Device has been subscribed already : just memorize new drain.*
$\mathcal{E}[[\mathcal{K}, \Delta K, \mathsf{CHECKSTART}(X, n_S)$
$\quad | \quad K.subscribedBy(n_S) \neq \{\}$
$\quad \wedge \quad K'.subscribedBy(n_S) = K.subscribedBy(n_S) \cup \{X\} \quad]]$
$\rightsquigarrow \mathcal{E}\,[[\mathcal{K}, K, \varepsilon]]$

// Whole chain, going from target to drain node :

(146) // If drain is not local on current node, then add *next* node (w.r.t. drain-wards direction)
// to subscribers and pass message to this :

$$\begin{aligned} &\mathcal{E}[\![\mathcal{K}, K, \texttt{service}(o::i, \texttt{DOsubscribe}(n_T, n_S, k_L))\\ &\qquad \mid \quad k_T \neq K.ownIdent \\ &\qquad \wedge \quad k_X = \boxed{\begin{cases} k & \text{if } K.routing.gate(k_T) = \texttt{hop}(k) \\ k_T & \text{if } K.routing.gate(k_T) = \texttt{driver}(\ldots) \end{cases}} \\ &\qquad]\!] \\ \rightsquigarrow\ &\mathcal{E}[\![\mathcal{K}, K, \mathsf{DOSUBSCRIBE}(n_T, n_S, k_L, k_X) \\ &\qquad \Vert\!\!+\ \texttt{req}\ (k_X, \texttt{\$noresp},\ \texttt{DOsubscribe}\,(n_T, n_S, \boxed{K.ownIdent}\,)\ )\ ]\!] \end{aligned}$$

(147) // If drain node finally reached : Just memorize subscribers

$$\begin{aligned} &\mathcal{E}[\![\mathcal{K}, K, \texttt{service}(o::i, \texttt{DOsubscribe}(n_T, n_S, k_L))\\ &\qquad \mid \quad k_T = K.ownIdent\ ]\!] \\ \rightsquigarrow\ &\mathcal{E}\,[\![\mathcal{K}, K, \mathsf{DOSUBSCRIBE}(n_T, n_S, k_L, n_T)\ ]\!] \end{aligned}$$

(148) // If node is *not* hosting the source :
// just memorize drain node and source node for later `unsubscribe()`.

$$\begin{aligned} &\mathcal{E}[\![\mathcal{K}, \Delta K, \mathsf{DOSUBSCRIBE}(n_T, n_S, k_L, X)\\ &\qquad \mid \quad k_S \neq K.ownIdent \\ &\qquad \wedge \quad K'.subscribedBy(n_S) = K.subscribedBy(n_S) \cup \{X\} \\ &\qquad \wedge \quad K'.subsrcnodes(n_S) = k_L\ ]\!] \\ \rightsquigarrow\ &\mathcal{E}[\![\mathcal{K}, K, \varepsilon\ ]\!] \end{aligned}$$

(149) // If node is hosting the source : just start the device

$$\begin{aligned} &\mathcal{E}[\![\mathcal{K}, K, \mathsf{DOSUBSCRIBE}(n_T, n_S, \_, X)\\ &\qquad \mid \quad k_S = K.ownIdent\ ]\!] \\ \rightsquigarrow\ &\mathcal{E}[\![\mathcal{K}, K, \mathsf{CHECKSTART}\ (X, n_S)\ ]\!] \end{aligned}$$

(150) // If both drain and source are local : Just turn off sending actor.

$$\begin{aligned} &\mathcal{E}[\![\mathcal{K}, \Delta K, \texttt{unsubscribe}(n_T, n_S)\ \mid\ k_S = K.ownIdent\ \wedge k_T = K.ownIdent\ ]\!] \\ \rightsquigarrow\ &\mathcal{E}\,[\![\mathcal{K}, K, \mathsf{CHECKOFF}\ (n_T, n_S)\ ]\!] \end{aligned}$$

(151) // If drain or source is remote : Start whole chain from drain to source

$$\begin{aligned} &\mathcal{E}[\![\mathcal{K}, K, \texttt{unsubscribe}(n_T, n_S)\ \mid\ k_S \neq K.ownIdent\ \vee\ k_T \neq K.ownIdent\ ]\!] \\ \rightsquigarrow\ &\mathcal{E}\,[\![\mathcal{K}, K, \texttt{req}(k_T, \texttt{\$noresp}, \texttt{DOunsubscribe}(n_T, n_S))\ ]\!] \end{aligned}$$

(152) // Delete last subscriber:

$$\begin{aligned} &\mathcal{E}[\![\mathcal{K}, \Delta K, \mathsf{CHECKOFF}(X, n_S)\\ &\qquad \mid \quad K'.subscribedBy\ n_S = K.subscribedBy(n_S) \setminus \{X\} \\ &\qquad \wedge \quad K'.subscribedBy\ n_S = \{\} \\ &\qquad \wedge \quad D = K.driverFor(n_S)\ ]\!] \\ \rightsquigarrow\ &\mathcal{E}[\![\mathcal{K}, K, D\ \boxed{\texttt{->}}\ \texttt{streamControl}\ (\texttt{off})\ ]\!] \end{aligned}$$

(153) // Delete a subscriber, not the last!

$$\begin{aligned} &\mathcal{E}[\![\mathcal{K}, \Delta K, \mathsf{CHECKOFF}(X, n_S)\\ &\qquad \mid \quad K'.subscribedBy\ n_S = K.subscribedBy(n_S) \setminus \{X\} \\ &\qquad \wedge \quad K'.subscribedBy\ n_S \neq \{\}\ ]\!] \\ \rightsquigarrow\ &\mathcal{E}\,[\![\mathcal{K}, K, \varepsilon\ ]\!] \end{aligned}$$

Table 6 Interactions with Actor Subsystems, via trns or via HsLink

Kind of Interaction	source	target	normal speed	high sp.
			`request(...)`	
Parameter Change (j,p)	RBD	actor	$(n,$ `paramchange`$(j,p))$	—
Parameter Inquiry (j)	RBD	actor	$(n,$ `paraminq`$(j))$	—
Execute RT Operation (j,r)	RBD	actor	$(n,$ `rtOperation`$(j,r))$	YES `drn` type
Value Event detected (j,d)	producer	RBQ	$(\perp,$ `rtOperation(pushQ`$(d)))$ (`value_event()`)	YES `src` type

j	=	*LUID* of actor
p	=	parameter assignment description
n	=	hosting node of actor j
d	=	data value of some incoming external real-time event
r	=	description of a run-time operation, as defined by actor class

(154) // Unsubscribe of X reaches node hosting the source:

$$\mathcal{E}[[\mathcal{K}, K, \texttt{service}(o::i, \texttt{DOunsubscribe}(X, n_S)) \;|\; K.nodeOf(n_S) = K.ownIdent\;]]$$
$$\rightsquigarrow \mathcal{E}\,[[\mathcal{K}, K, \textsf{CHECKOFF}\;(X, n_S)\;]]$$

(155) // Delete last consumer for remote source: Unsubscribe own node from next node
// („next“ w.r.t. sourceward direction.)

$$\mathcal{E}[[\mathcal{K}, \Delta K, \texttt{service}(o::i, \texttt{DOunsubscribe}(X, n_S))$$
$$|\;\; k_S \neq K.ownIdent$$
$$\wedge\;\; K'.subscribedBy(n_S) = K.subscribedBy(n_S) \setminus \{X\}$$
$$\wedge\;\; K'.subscribedBy(n_S) = \{\}\;\;]]$$
$$\rightsquigarrow \mathcal{E}[[\mathcal{K}, K, \texttt{req}(K.subscrsourcenode(n_S), \texttt{\$noresp}, \texttt{DOunsubscribe}(K.ownIdent, n_S)\;)\;]]$$

(156)

$$\mathcal{E}[[\mathcal{K}, \Delta K, \texttt{service}(o::i, \texttt{DOunsubscribe}(X, n_S))$$
$$|\;\; k_S \neq K.ownIdent$$
$$\wedge\;\; K'.subscribedBy(n_S) = K.subscribedBy(n_S) \setminus \{X\}$$
$$\wedge\;\; K'.subscribedBy(n_S) \neq \{\}\;\;]]$$
$$\rightsquigarrow \mathcal{E}\,[[\mathcal{K}, K, \varepsilon\;]]$$

3.13 Application *and* Transport Layer : HsLink

3.13.1 Principles of Operation (Non-Normative)

The modularity, adaptability and reliability of the basic (text-based) communication model — as developed so far — have been paid for by some execution overhead w.r.t. encoding, routing and decoding.

Because this can make its application infeasible in certain practical situations, TUB-TCI provides a second, parallel way of communication dedicated to high speed transmission. Since these operate with „binary encoded“ or „native“ data, it can be used for minimal delay signal transport between subsystems of compatible, maybe proprietary encoding[45].

In some concerns the HsLinks are treated/behave in a *dual* way compared to the „normal speed“ trns-based control and signal flow. The concept of HsLinks is characterized as follows:

- For the transfer of value events and run-time commands which occur frequently with *high bandwidth* and *known* sources and drains, a „*High Speed Link*“ (= HsLink) can be allocated.
- A HsLink is a *steady* connection between one or more sources and one or more targets.
- Each HsLink must be allocated, reserved and prepared during TPrep. It stays alive during the whole of a TRun, but the actual transmission can be switched on and off.
- There are two kinds of HsLinks, cf. table 6, namely:
  - Either the single source is a subsystem of category Producer, the drains are one or more RBQs, and the transmitted messages are `value_events()`. This is a one-to-many signal flow, and called HsLink of `src`-type.
    By this type of HsLink `value_events()` (like the arrival of a PDU at a certain port) can be signaled to local and remote consumers with minimal delay.
    A HsLink of `src`-type is uniquely identified by the only producer subsystem serving as its *source*.
  - Or the single drain is some actor, the one or more sources are RBDs, and the transmitted messages trigger some run-time operations (`rtOperation()` messages). This is a many-to-one signal flow, and called HsLink of `drn`-type.
    By this type of HsLink a messages like `startTimer()` can be executed on any actor (maybe transparently on a remote node) with minimal delay.
    A HsLink of `drn`-type is uniquely identified by the only actor subsystem serving as its *drain*.
- In contrast to message transmission via trns, the HsLink-messages will (1) not be encoded textually, but are in some *arbitrary native „binary“ encoding*, (2) this encoding will always be known statically, and (3) it is always known in advance, to which drains the messages must be delivered.
- The responsibility for the functional correctness of any HsLink is out of the scope of TUB-TCI: Since at the moment there is no representation of *encoding rules* on the object language level[46], the *compatibility* of the binary encodings used by sources and drains must be validated by an external system, which has the necessary knowledge (e.g. realized as a part of TM).
  In spite of the lack of a formal declaration mechanism, nevertheless a clear specification of applied encoding rules should be given informally (on documentation level) with any factory which supports the binary HsLink interfaces.
- To minimize the load on the communication infrastructure used for HsLinks, those of `src`-type may be dynamically subscribed and unsubscribed by their consumers during TRun.

[45] For the reader familiar with the corresponding ETSI standards: The HsLink mechanism can be considered as a generic, dynamic way of integrating a TRI-system into TUB-TCI (cf. [tri02], [tci03]).

[46] Section 5.2.2 will discuss a possible corresponding extension of the specification.

In contrast to the trns-based „normal speed" push-channels, this subscription is realized as a simple „switch-on/switch-off" protocol, which does not affect the allocation and reservation of the HsLinks, but only enables/disables the data transmission.
For this protocol to have any physical effect, the involved HS-BusAdapters must provide a *control channel*, i.e. a transmission channel in the opposite direction of the signal transmission.

- Every actor/RBQ which shall be used as a drain in a HsLink must implement a „binary version" of the `rtOperation()/rtOperation(putQ())`-service.
  On concept level this corresponds to the inclusion of the schemas *Factory_supporting_HsRTOP/ Factory_supporting_HsRBQ*.
  Every producer which shall be used as a source in a HsLink must call the „binary version" of the `value_event()`-service in course of its client behavior. Additionally it may include the schema *Factory_supporting_HsProducer* and implement the `switchHsValueProduction()` service, which allows on/off-control of signal production for sake of load minimization.

A central issue concerning with HsLinks is, that they are introduced for sake of efficiency, and therefore have to be implemented closely following the specific needs and properties of the involved hardware. This is supported by TUB-TCI (e.g. by treating the concrete data flow opaquely), while nevertheless all configuration and control activity is lifted up to the necessary level of abstraction.

Since the communication resources of each hardware node shall be used optimally, the basic operation of *creating* an HsLink can best be done by the *receiving* node, — mostly because the kind of the underlying communications channel can induce very different modes of access to the HsLink it realizes, cf. figure 13:

- Consider e.g. a IP driven bus, when a HsLink is realized by a dedicated „Socket". *All* other nodes on this bus – beside the one for which the HsLink originally has been created – can also write to this HsLink immediately.
- The opposite is true for e.g. some serial-peer-to-peer-timeslot-ring. Here slots can be reused depending on the physical topology, and only that subset of nodes *physically* between the source node and the target node can use a given slot to read or write values.

So the node site HsLink creation procedure may return a *set* of *NodeIdents*, indicating which nodes are capable (for „physical" reasons) to use that HsLink from now on for writing. This set may contain significantly more nodes than only the one which originally requested for the HsLink.

Each HsLink address is represented as tupel (a, b, c) for a sending node. The sending node always knows through which of its own bus adapters $(= a)$ together with which bus address $(= b)$ to reach the target node[47].

But only the target node can create the „channel number" $(= c)$ for the new HsLink, which allows the receiving driver to identify the HsLink and its drain optimally (cf. figure 13). This channel number is identical for all source nodes writing on this HsLink via the same bus, and it is globally unique for a certain combination of target not, bus and HsLink.

[47]**Please notice**, that b can differ for the same target node on the same bus with different source nodes.

3.13.2 Additional State Space

(157)

$$| \; \mathit{HsChannelNum} == \mathbb{N}$$

(158)

$$| \; \mathit{HsBusAdr}$$

(159)

$$| \; \mathit{CodingObject}$$

(160)

$$\begin{array}{|lcl} \mathit{HsKind} & == & \{ \texttt{src}, \texttt{drn} \} \\ \mathit{HsState} & == & \{ \texttt{on}, \texttt{off} \} \end{array}$$

(161)

```
┌─ NodeState ──────────────────────────────────────────────
│ ...
│ HsInRouting  : NodeIdent ⇸ [D : HsAdapter;  r : HsBusAdr]
│                // The bus address r is needed for reverse control messages!
│ HsOutRouting : NodeIdent ⇸ [D : HsAdapter;  b : HsBusadr ]
│
│ HsInLinks    : ( LUID × HsKind × CodingObject)
│                     ⇸ [D : HsAdapter;  r : HsBusAdr;
│                        c : HsChannelNum;  x : HsState
│                        writers : ℙ NodeIdent  ]
│                // The bus address r is needed for reverse control messages!
│
│ HsOutLinks   : ( LUID × HsKind × CodingObject)
│                     ⇸  ℙ [D : HsAdapter;  b : HsBusAdr;
│                     c : HsChannelNum;  x : HsState  ]
│ HsLocalSubs  : LUID ⇸ ℙ LUID
│ ...
├──────────────────
│ dom HsInLinks ∩ dom HsOutLinks = {}
│ ⋃ ran HsLocalSubs ⊂ actors
└──────────────────────────────────────────────────────────
```

// Inrouting and outrouting could be from ℙ type, – different design !!!

(162)

Factory_supporting_HsRTOP

$Actor\_Factory$

$\boxed{\texttt{->}}$ HsRtOperation$(LUID \times RT)$

(163)

Factory_supporting_HsProducer

$extendParams(Factory\_supporting\_Producer, [coding : CodingObject])$
$PI(\text{"}\texttt{coding}\text{"}).mode = \texttt{CO}$

$\boxed{\texttt{->}}$ switchHsValueProduction$(LUID \times \{\texttt{on}, \texttt{off}\})$

(164)

Factory_supporting_HsRBQ

$extendParams(Factory\_supporting\_RBQ, [coding : CodingObject])$
$PI(\text{"}\texttt{coding}\text{"}).mode = \texttt{CO}$

$\boxed{\texttt{->}}$ HsPutQ$(LUID \times INDAT)$

(165)

HsAdapter_In

$bus : Bus$
$busAdr : HsBusAdr$

$\boxed{\texttt{->}}$ watchBus $()$
$\boxed{\texttt{->}}$ incomingMessage $(HsChannelNum \times \_\_DATA\_\_)$
$\boxed{\texttt{->}}$ controlInSlot $(HsChannelNum \times \{ \texttt{on}, \texttt{off} \})$

$\boxed{\texttt{->}}$ allocateInSlot $NodeIdent \rightarrow HsChannelNum \times \mathbb{P}\, NodeIdent \times HsState$

(166)

HsAdapter_Out

$bus : Bus$
$revertBusAdr : HsBusAdr$

$\boxed{\text{->}}$ putMessage $(HsBusadr \times HsChannelNum \times \_\_DATA\_\_)$

$\boxed{\text{->}}$ watchBusR
$\boxed{\text{->}}$ detectedChannelSwitch$(HsBusAdr \times HsChannelNum \times \{\texttt{on}, \texttt{off}\})$

3.13.3 Messages

(167)
$$\begin{aligned} SRQS\_TM\_to\_Node \;::=\; & \texttt{HScreateinlink} \langle\!\langle [\, actor : LUID \\ & \quad type : HsKind \\ & \quad srcnode : NodeIdent \,]\rangle\!\rangle \\ & \quad \Rrightarrow \texttt{HScreated} \langle\!\langle [\, srcs : \mathbb{P}\, NodeIdent \\ & \quad\quad cid : HsChannelNum \,]\rangle\!\rangle \\ & \texttt{HSregisteroutlink} \langle\!\langle [\, actor : LUID \\ & \quad type : HsKind \\ & \quad hosting : NodeIdent \\ & \quad channel : HsChannelNum \,]\rangle\!\rangle \end{aligned}$$

(168)
$$\begin{aligned} SRQS\_Producers\_to\_QAS \;::=\; & \texttt{HSvalue\_event} \langle\!\langle [\, id : LUID \\ & \quad data : DATA]\rangle\!\rangle \end{aligned}$$

3.13.4 $\mathsf{TM} \overset{6}{\Longrightarrow} \boxed{\mathsf{Node}}$: Create High Speed In-Links and Out-Links

j : *LUID*
k_K, k_X, k_S, k_T : *NodeIdent*
x : *HsKind*
c : *HsChannelNum*
s : *HsState*
$\mathcal{N}$: $\mathbb{P}$ *NodeIdent*

```
(169) // Create an incoming slot, writeable by $K_S$
      // If $kind = \texttt{src}$, then $K_S$ will be the *only* writer
      // and must later be addressed in channel-on/off commands
      $\mathcal{E}[[\mathcal{K}, \Delta K, \texttt{service}\ (o::i,\ \texttt{HScreateinlink}(j, x, K_S)\ )$
                    $|\ \ (D, r)\ =\ K.\mathit{HsInRouting}\ (K_S)$
                    $\wedge\ \ (c, \mathcal{N}, s)\ =\ D\ \boxed{\texttt{->}}\ \mathsf{HsAllocateInSlot}(K_S)$
                    $\wedge\ \ K'.\mathit{HsInLinks}\ =\ K.\mathit{HsInLinks} \cup \{j::x\ \mapsto\ (D, r, c, s, \mathcal{N})\}\ ]]$
  $\leadsto$ $\mathcal{E}\,[[\mathcal{K}, K, \texttt{reply}\ (o::i, \texttt{HScreated}(\mathcal{N}, c))\ ]]$
      // **Please notice** that the activation state is *not* told to CAS

(170) // Memorize that actor $j$ is either writable ($\texttt{drn}$) or *required* ($\texttt{src}$) by HS channel $c$ on node $K_N$
      $\mathcal{E}[[\mathcal{K}, \Delta K, \texttt{service}\ (o::i,\ \texttt{HSregisteroutlink}\ (j, x, K_N, c)\ )$
                    $|\ \ (D, b)\ =\ K.\mathit{HsOutRouting}\ (K_N)$
                    $\wedge\ \ K'.\mathit{HsOutLinks}\ =\ K.\mathit{HsOutLinks} \cup \{j::x\ \mapsto\ (D, b, c, \texttt{off})\}\ ]]$
  $\leadsto$ $\mathcal{E}\,[[\mathcal{K}, K, \texttt{reply}\ (o::i, \texttt{OK})\ ]]$
```

3.13.5 $\boxed{\mathsf{Producer}} \overset{11}{\Longrightarrow} \mathsf{QAS}$: Generate Value Events For High Speed Channels

j : *LUID*
M : *Message*
// M' = native binary encoding of M

```
      // Every Factory supporting HS_producer hat to enhance the transition rule (135) on page 81
      //  by the following rule (171) :

(171) // *Locally* generated value event :
      $\mathcal{E}[[\mathcal{K}, K, \texttt{value\_event}(j, M)\ ]]$
  $\leadsto$ $\mathcal{E}\,[[\mathcal{K}, K, \mathsf{HSINPUT}\ ((j, \texttt{src}),\ M')\ ]]$
```

3.13.6 $\boxed{\mathsf{QAS}}\overset{-/9}{\Longrightarrow}\mathsf{QAS} \cup \mathsf{RBQ}$: Delivering HS Value Events

j : *LUID*
N : *NodeIdent*
A : $LUID_{RBQ}$ *// designates an „Actor“ of kind „RBQ“.*
H : $LUID_{HSLINK}$ *// designates an OUTGOING „high speed link“ object.*

// "`drn`" type : many-to-one mode, i.e. *alternatives* of delivery :

(172) // Pass "`drn`" type messages to hosting node of actor:
$\mathcal{E}[[\mathcal{K}, K, \mathsf{HSINPUT}((j, \mathtt{drn}), M\,)$
$\quad | \quad (j \mapsto (\_, \_, K.ownIdent, \_)\,) \notin K.actors$
$\quad \wedge \quad (j, \mathtt{drn}) \mapsto (D^N, b, c) \in K.HsOutlinks \;]]$
$\rightsquigarrow \mathcal{E}\,\Big[[\mathcal{K}, K, D^N\,\boxed{\mathsf{->}}\,\mathsf{putMessage}\,(b, c, M)\,\Big]]$

(173) // Execute "`drn`" type messages if actor is local:
$\mathcal{E}[[\mathcal{K}, K, \mathsf{HSINPUT}((j, \mathtt{drn}), M\,)$
$\quad | \quad (j \mapsto (\_, K.ownident, \_, \mathtt{living})\,) \in K.actors$
$\quad \wedge \quad F = K.findDriver(K.classOf(j)) \;]]$
$\rightsquigarrow \mathcal{E}\,\Big[[\mathcal{K}, K, F\,\boxed{\mathsf{->}}\,\mathsf{HsRtOperation}(j, M)\,\Big]]$

// "`src`" type : one-to-many mode, i.e. *conjunction* of delivery :

(174) // Send "`src`" type message to all consumers:
$\mathcal{E}[[\mathcal{K}, K, \mathsf{HSINPUT}((j, \mathtt{src}), M\,)$
$\quad | \quad S = K.HsOutLinks(\!|\{(j, \mathtt{src})\}|\!) \cup K.HsLocalSubs(j) \;]]$
$\rightsquigarrow \mathcal{E}\,[[\mathcal{K}, K, \mathsf{DELIVERVALUEEVENT}(S, M)\,]]$

(175) // Feed local consumers with data:
$\mathcal{E}[[\mathcal{K}, K, \mathsf{DELIVERVALUEEVENT}\;(\{n_T\} \cup S, M)$
$\quad | \quad (n_T \mapsto (\_, K.ownIdent, \_)\,) \in K.actors$
$\quad \wedge \quad F = K.findDriver(K.classOf(n_T)) \;]]$
$\rightsquigarrow \mathcal{E}\,\Big[[\mathcal{K}, K, F\,\boxed{\mathsf{->}}\,\mathsf{HsPutQ}(n_T, M) \;|\!|\!+\; \mathsf{DELIVERVALUEEVENT}\,(S, M)\,\Big]]$

(176) // Pass on to subscribed nodes:
$\mathcal{E}[[\mathcal{K}, K, \mathsf{DELIVERVALUEEVENT}\;(\{X\} \cup S, M)$
$\quad | \quad X = (D^N, b, c, \mathtt{on}) \;]]$
$\rightsquigarrow \mathcal{E}[[\mathcal{K}, K, D^N\,\boxed{\mathsf{->}}\,\mathsf{putMessage}\,(b, c, M) \;\|\; \mathsf{DELIVERVALUEEVENT}\,(S, M)\,]]$

(177) // Ignore switched off channels for lowering channel load:
$\mathcal{E}[[\mathcal{K}, K, \mathsf{DELIVERVALUEEVENT}\;(\{X\} \cup S, M)$
$\quad | \quad X = (D^N, b, c, \mathtt{off}) \;]]$
$\rightsquigarrow \mathcal{E}\,[[\mathcal{K}, K, \mathsf{DELIVERVALUEEVENT}\,(S, M)\,]]$

(178) // Done:
$\mathcal{E}[[\mathcal{K}, K, \mathsf{DELIVERVALUEEVENT}\;(\{\}, M)\,]]$
$\rightsquigarrow \mathcal{E}[[\mathcal{K}, K, \varepsilon\,]]$

3.13.7 $\mathsf{RBD}\overset{10}{\Longrightarrow}\boxed{\mathsf{QAS}}$: Subscribe and Unsubscribe HsLink

n_S : *LUID*
$T : \{\mathtt{on}, \mathtt{off}\} \;\wedge\; \neg(\mathtt{on}) = \mathtt{off} \;\wedge\; \neg(\mathtt{off}) = \mathtt{on}$
someoneListens : $K \times LUID \rightarrow BOOL$
$someoneListens(k, n) \;=\; (\; k.hsLocalSubs = \{\}\;)$
$\qquad\wedge\;\; \forall\,(D, b, c, x) \;\in\; k.outLinks(\!|\{(n, \mathtt{src})\}|\!) \;\bullet\; x = \mathtt{off}\;\;]]$

(179) // Three possibilities to reach a *new on/off state* for an HsInLink :
$\mathcal{E}[[\mathcal{K}, K, \mathsf{SWITCH}\;(n_S, T) \;\mid\; (n_S, \mathtt{src}) \mapsto (D, b, c, T) \;\in\; K.HsInLinks\;]]$
$\leadsto\; \mathcal{E}\,[[\mathcal{K}, K, \varepsilon\,]]$

(180)
$\mathcal{E}[[\mathcal{K}, K, \mathsf{SWITCH}\;(n_S, T) \;\mid\; (n_S \;\mapsto\; (\_, K.ownIdent, \_)\,) \;\in\; K.actors$
$\leadsto\; \mathcal{E}\;\big[[\mathcal{K}, K, K.factoryFor(n_S)\;\boxed{\texttt{->}}\;\mathsf{switchHsValueProduction}\;(\;T\;)\;\big]]$

(181)
$\mathcal{E}[[\mathcal{K}, K, \mathsf{SWITCH}\;(n_S, T) \;\mid\; (n_S, \mathtt{src}) \mapsto (D, b, c, (\neg\;T)\,) \;\in\; K.HsInLinks\;]]$
$\leadsto\; \mathcal{E}\;\big[[\mathcal{K}, K, D\;\boxed{\texttt{->}}\;\mathsf{HsControlInSlot}\;(r, D.busAdr, c, T)\;\big]]$

(182) // (always *locally)* executed (un-)subscription initializes rippling :
$\mathcal{E}[[\mathcal{K}, K, \mathtt{HSsubscribe}(n_T, n_S) \;\mid\; K'.HsLocalSubs(n_S) = K.HsLocalSubs(n_S) \cup \{n_T\}\;\;]]$
$\leadsto\; \mathcal{E}\,[[\mathcal{K}, K, \mathsf{SWITCH}(n_S, \mathtt{on})\;]]$

(183)
$\mathcal{E}[[\mathcal{K}, K, \mathtt{HSunsubscribe}(n_T, n_S) \;\mid\; K'.HsLocalSubs(n_S) = K.HsLocalSubs(n_S) \setminus \{n_T\}\;\;]]$
$\leadsto\; \mathcal{E}\,[[\mathcal{K}, K, \mathsf{TESTOFF}(n_S)\;]]$

(184)
$\mathcal{E}[[\mathcal{K}, K, \mathsf{TESTOFF}(n_S) \;\mid\; \neg\; someoneListens(K, n_S)\;]]$
$\leadsto\; \mathcal{E}\,[[\mathcal{K}, K, \mathsf{SWITCH}(n_S, \mathtt{off})\;]]$

(185)
$\mathcal{E}[[\mathcal{K}, K, \mathsf{TESTOFF}(n_S) \;\mid\; someoneListens(K, n_S)\;]]$
$\leadsto\; \mathcal{E}\,[[\mathcal{K}, K,\; \varepsilon\,]]$

3.13.8 $\boxed{\textsf{HsBusAdpt}} \overset{electric}{\Longrightarrow} \textsf{HsBusAdpt}$ HS Bus Communication

$j \ : \ LUID$
$K, L \ : \ NodeState$
$D \in K.HsAdapters\_Out \quad \wedge \quad E \in L.HsAdapters\_In$
$c \ \in \ \{\texttt{on}, \texttt{off}\}$

(186)

$$\mathcal{E}[[\mathcal{K}, K,\ D \boxed{\texttt{->}} \textsf{putMessage}(b, c, M) \parallel L,\ E \boxed{\texttt{->}} \textsf{WatchBus} \quad | \quad D.bus = E.bus \ \wedge \ E.busAdr = b \ ]]$$
$$\rightsquigarrow \ \mathcal{E}[[\mathcal{K}, K, \varepsilon \parallel L,\ E \boxed{\texttt{->}} \textsf{WatchBus} \,|||\!\!+ E \boxed{\texttt{->}} \textsf{incomingMessage}(c, M)\ ]]$$
// The identity of the busses is located in the ideal space and realized electrically

(187)

$$\mathcal{E}[[\mathcal{K}, L,\ E \boxed{\texttt{->}} \textsf{incomingMessage}(c, M)\ ]]$$
$$\rightsquigarrow \ \mathcal{E}[[\mathcal{K}, L,\ \textsf{HSINPUT}(\ K.HsInlinks^{\sim}\ (E, c, \texttt{on}), M\ )\ ]]$$

(188) // *Reverse* communication from input to output adapter :

$$\mathcal{E}[[\mathcal{K}, L, E \boxed{\texttt{->}} \textsf{HsControlInSlot}(r, b, c, x), \parallel K, D \boxed{\texttt{->}} \textsf{WatchBusR} \quad | \quad D.bus = E.bus \ \wedge \ r \ = \ D.revertAdr\ ]]$$
$$\rightsquigarrow \ \mathcal{E}[[\mathcal{K}, L, \varepsilon \parallel K, D \boxed{\texttt{->}} \textsf{WatchBusR} \,|||\!\!+ \textsf{ConsumerSwitched}(D, b, c, x)\ ]]$$

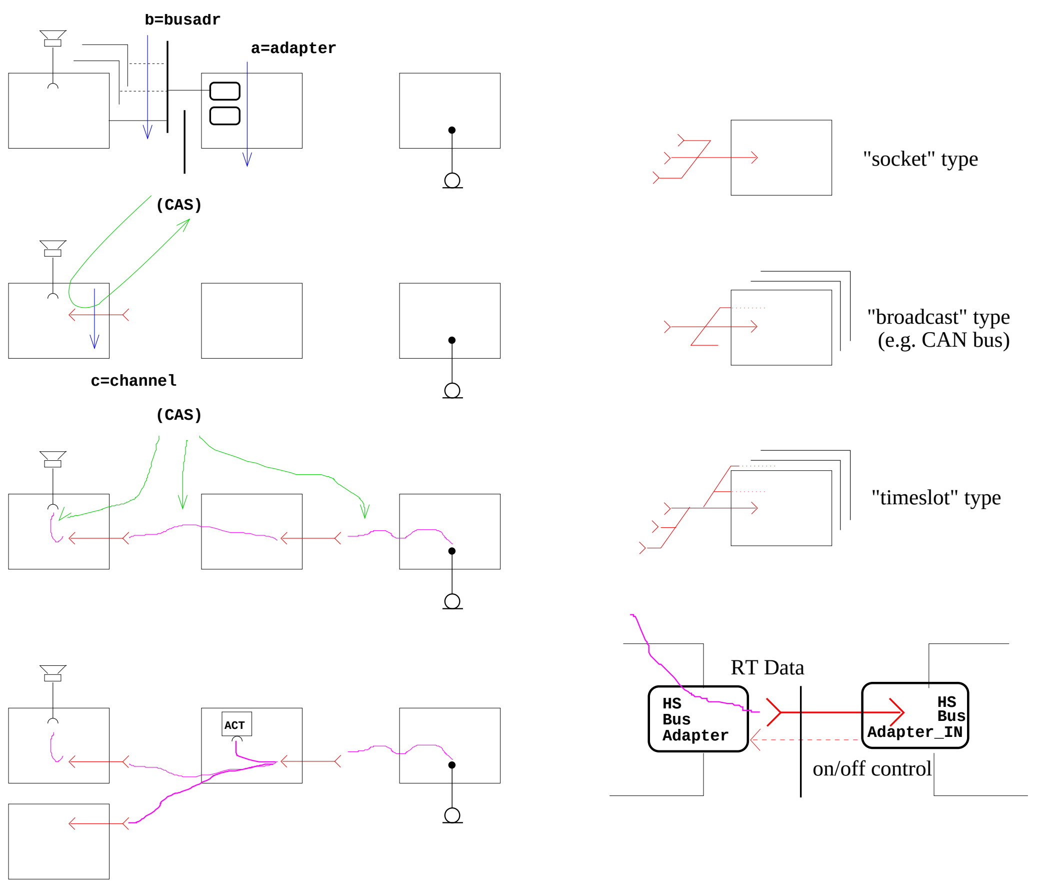
(189)

$$\mathcal{E}[[\mathcal{K}, \Delta L, \textsf{ConsumerSwitched}\ (D, b, c, \texttt{on}) \quad | \quad (j, \texttt{src}) \mapsto (D, b, c, \texttt{off}) \ \in \ L.HsOutSlots \quad \wedge \quad L'.HsOutSlots = L.HsOutSlots \oplus \{(j, \texttt{src}) \mapsto (E, b, c, \texttt{on})\}\ ]]$$
$$\rightsquigarrow \ \mathcal{E}\ [[\mathcal{K}, L, \textsf{SWITCH}\ (n_S, \texttt{on})\ ]]$$

(190)

$$\mathcal{E}[[\mathcal{K}, \Delta L, \textsf{ConsumerSwitched}\ (D, b, c, \texttt{off}) \quad | \quad (j, \texttt{src}) \mapsto (D, b, c, \texttt{on}) \ \in \ L.HsOutSlots \quad \wedge \quad L'.HsOutSlots = L.HsOutSlots \oplus \{(j, \texttt{src}) \mapsto (E, b, c, \texttt{off})\}\ ]]$$
$$\rightsquigarrow \ \mathcal{E}\ [[\mathcal{K}, L, \textsf{TESTOFF}\ (n_S)\ ]]$$

Figure 13 Creation of HsLinks / Variants of HsLinks determined by the underlying Technology



3.14 Population Layer : Predefined and Required Actor Classes

3.14.1 $\mathsf{CAS}\overset{6;\,7}{\Longrightarrow}\boxed{\mathsf{Factory}}$ and $\mathsf{RBD}\overset{8}{\Longrightarrow}\boxed{\mathsf{Actors}}$: Library of Predefined General Purpose Actor Class Declarations.

(191)

AC_Timer

$ActorClass[configurationParameters]$

$$\begin{array}{lll} configurationParameters & ::= & \texttt{timer\_cparams}\ \langle\!\langle [minResolution : Duration \\ & & \quad curResolution : Duration \\ & & \quad maxTransmissionDelay : Duration \\ & & \quad maxValue : Duration \\ & & \quad]\rangle\!\rangle \end{array}$$

$parameterInfo(\texttt{minResolution}) \subset [PI \mid mode = \texttt{CO}]$
$parameterInfo(\texttt{maxDuration}) \subset [PI \mid mode = \texttt{RO}]$
$PI\ (\text{"}\texttt{curResolution}\text{"}).mode \leq \texttt{RW}$

$$\begin{array}{lll} run - timeOperations & ::= & \texttt{start} \\ & \mid & \texttt{stop} \\ & \mid & \texttt{reset}\langle\!\langle Duration\rangle\!\rangle \\ & \mid & \texttt{read} \Rrightarrow \texttt{timerval}\langle\!\langle Duration\rangle\!\rangle \end{array}$$

(192)

AC_Executable

$ActorClass[configurationParameters]$

$$\begin{array}{lll} configurationParameters & ::= & \texttt{executable\_cparams}\ \langle\!\langle [codebase : Executable\_URL \\ & & \quad heapsize : \mathbb{N} \\ & & \quad defaultstacksize : \mathbb{N} \\ & & \quad \ldots \\ & & \quad]\rangle\!\rangle \end{array}$$

$parameterInfo(\texttt{codebase}) \subset [PI \mid mode = \texttt{WI};\ default = \{\}]$
$parameterInfo(\texttt{heapsize}) \subset [PI \mid mode \geq \texttt{RW}]$

$run - timeOperations ::= \texttt{readdiagnosis} \Rrightarrow \texttt{diagdata}\langle\!\langle \_\_DATA\_\_\rangle\!\rangle$

(193)

AC_Component

$$\mathit{ActorClass}[\mathit{configurationParameters}]$$

$$\begin{array}{lll}
\mathit{configurationParameters} & ::= & \texttt{component\_cparams}\ \langle\!\langle[\mathit{executable} : \mathit{LUID} \\
 & & \quad \mathit{componentClassIdentifier} : \mathit{TextString} \\
 & & \quad \mathit{stacksize} : \mathbb{N} \\
 & & \quad \ldots \\
 & & \quad]\rangle\!\rangle
\end{array}$$

$$\mathit{parameterInfo}(\texttt{executable}) \subset [\mathit{PI} \mid \mathit{mode} = \texttt{WI}\,;\ \mathit{default} = \{\}\,]$$
$$\mathit{parameterInfo}(\texttt{stacksize}) \subset [\mathit{PI} \mid \mathit{mode} \geq \texttt{WI}]$$

$$\begin{array}{lll}
\mathit{run} - \mathit{timeOperations} & ::= & \texttt{start} \\
 & \mid & \texttt{stop} \\
 & \mid & \texttt{getStatus} \Rrightarrow \texttt{componentState}\langle\!\langle\{\texttt{halted,running}\}\rangle\!\rangle \\
 & & \ldots
\end{array}$$

(194)

AC_RBQ

// . . . could be called „AC_SignalDrain“ !!

$$\mathit{ActorClass}[\mathit{configurationParameters}]$$

$$\begin{array}{lll}
\mathit{configurationParameters} & ::= & \texttt{rbq\_cparams}\ \langle\!\langle[\ \texttt{valueType}\ \langle\!\langle\ \_\_\mathit{SIMPLETYPE}\_\_\ \rangle\!\rangle \\
 & & \quad]\rangle\!\rangle
\end{array}$$

$$\mathit{run} - \mathit{timeOperations}\ ::=\ \texttt{writeQ}\ \langle\!\langle\ \boxed{\text{getconfigurationparameter}(\text{"}\texttt{valueType}\text{"})}\ \rangle\!\rangle$$

$$\mathit{parameterInfo}(\texttt{valueType}) \subset [\mathit{PI} \mid \mathit{mode} = \text{"}\texttt{RO}\text{"} \ \vee\ \mathit{mode} = \text{"}\texttt{WI}\text{"}\ ]$$

(195)

AC_Watchpoint

$$ActorClass[configurationParameters]$$

$$\begin{array}{lll}
configurationParameters & ::= & \texttt{watchpoint\_cparams}\ \langle\!\langle [executable : LUID \\
 & & \quad flavourdependentpositioninformation : DATA^2 \\
 & & \quad flavour : \mathbb{P}\ \{\texttt{break}, \texttt{protocol}, \texttt{condition}\} \\
 & & \quad active : BOOL \\
 & & \\
 & & \quad curvalue : DATA^1 \\
 & & \quad condition : DATA^1 \twoheadrightarrow BOOL \\
 & & \quad //\ condition : \mathbb{P}\ DATA^1 \\
 & & \quad \ldots \\
 & & \quad]\rangle\!\rangle
\end{array}$$

$$parameterInfo(\texttt{executable}) \subset [PI \mid mode \geq \texttt{WI};\ default = \{\}]$$
$$parameterInfo(\texttt{flavour}) \subset [PI \mid mode \geq \texttt{WI}]$$

$$\begin{array}{lll}
run - timeOperations & ::= & \texttt{start} \\
 & \mid & \texttt{stop} \\
 & \mid & \texttt{getcurvalue} \Rrightarrow \texttt{watchedvalue}\ \langle\!\langle\ DATA^1\ \rangle\!\rangle \\
 & \mid & \texttt{getStatus} \Rrightarrow \texttt{breakpointData}\langle\!\langle DATA^3 \rangle\!\rangle \\
 & & \ldots
\end{array}$$

(196)

AC_TransCoder[in : CodingIdent, out : CodingIdent]

$$ActorClass[configurationParameters]$$

$$\begin{array}{lll}
configurationParameters & ::= & \texttt{transcoder\_cparams}\ \langle\!\langle [inputCoding : CodingId \\
 & & \quad outputCoding : CodingId \\
 & & \quad]\rangle\!\rangle
\end{array}$$

$$parameterInfo(\texttt{inputCoding}) \subset [PI \mid mode = \texttt{WI};\ default = \{\}]$$
$$parameterInfo(\texttt{outputCoding}) \subset [PI \mid mode = \texttt{WI};\ default = \{\}]$$

$$\begin{array}{lll}
run - timeOperations & ::= & \texttt{putQ}\langle\!\langle \_DATA\_ \rangle\!\rangle \\
 & & \qquad \Rrightarrow \texttt{value\_event}\ \langle\!\langle \_DATA\_ \rangle\!\rangle \\
 & & //\ \textit{implements an } \mathsf{RBQ}\text{- }\textit{and an } \mathsf{RBD}\textit{-like behavior !}
\end{array}$$

(197)

$AC\_GlobalTestParameters\ [\ testParameterList\ ]$

$ActorClass[configurationParameters]$

$configurationParameters\ ==\ testParameterList$

4 Data-Link Layer

4.1 XML, the Language of the Toasters

In the low level data-link layer two different ways of encodings are foreseen by TUB-TCI: firstly the standardized, XML based encoding, which must be supported by any implementation, secondly the proprietary binary encodings for high-speed-channels (HsLink, see section 3.13), which need not to be standardized or compatible.

All activities in a TUB-TCI-system are defined by service request interactions between subsystems. These are realized as the exchange of `deliver()`-messages and other messages.

Therefore one dedicated encoding must be chosen, and be declared normative, which maps all possible `deliver()` messages uniquely and invectively to XML-encoded fragments.

To construct this basic encoding, several mappings have to collaborate to translate the Z formulae of the main specification into „physical" XML. As far as possible *existing standards* should be used for these mappings. The following areas have to be covered:

- All *constructors* into free types, which are defined as a fixed set by the TUB-TCI-core-language itself, must (and the containing *schema name* may) be represented by an XML `ELEMENT` tag value, i.e. an XML „element type".
- The same is true for the constructors defined on „population layer", e.g. the structures containing configuration parameter values of a given actor. That means that a canonical construction method must be defined to translate the schema definitions of an actor class into an XML content definition, similar to above.
- The encoding of primitive types and values (`int`, `float`, `string`, etc) should be cited from some existing, simple standard, e.g. `SOAP`([soa00]), `Schema` ([xml02]), XER ([xer02]).
- Large binary encoded parameter values (e.g. the actual *code* to be executed by an RBD) must be transported by introducing a specialized XML `NOTATION`.
- The use of XML `ATTRIBUTE`s should be avoided.

4.1.1 XML Conformance Levels

Despite the normative representation of data as XML Elements, there are still many degrees of freedom in implementing the concrete textual appearance.

We suggest to allocate some *Information Objects* and to define the formal conformance level of a concrete TCI implementation as a *subset* of this conformance objects, i.e.

$$conformanceLevel\ ==\ \mathsf{TCI\_implementation}\ \rightarrow\ \mathbb{P}\ conformanceObjects$$

Some illustrating examples for Conformance Objects can be found in table 7.

Table 7 Examples of XML Encoding Conformance Level Objects

`global.orgs.NN.NN.NN.tci.busformat.xml.featuregroup ...`

⟨*almost any*⟩	`.standard`	= Default value Implements neither more nor less than the standard
	`.TMPplus`	Implements more than the standard, but has not yet been registered as conformance object.
	`.TMPminus`	Implements less than the standard. Only allowed for internal use while *developing* an implementation.
`.compression`	`.none`	Plain text on the wire !
	`.plus`	Does *additionally* understand a proprietary compressed format.
	`.minus`	Does *only* understand a proprietary compressed format.
`.canonical`	`.none`	No canonicalization
	`.plus`	*generates* canonicalizes XML, e.g. no blanks around tags, no additional attributes etc.
	`.minus`	Does *only* understand canonical XML, e.g. does *not* understand additional blanks etc.

4.2 High Speed Binary Real-Time Channels

In the current approach the declaration of the formats of the HsLinks is not represented formally, — not even on the concept level, not to speak of the object language.

So the TM and/or the *compiler* (which generates the ETS, i.e. the code of the RBDs) and/or the run-time library must be *aware* of the actual ways of encoding and the compatibility relation on the high-speed I/Os of all currently installed actors. Only this allows to reject inconsistent connection requests, or even to repair them by inserting adapting trans-coder-subsystems automatically into the signal-flow.

In the main model all those checks have to be hand-coded. The extension suggested in 5.2.2 below will add the formats of encoding to the conceptual level and to the object language, to support further automating of signal flow management.

5 Perspectives of Future Work

At the time of writing, TUB-TCI is only a first proposal for an overall model-based specification of the somehow „abstract" behavior of an heterogenous test ensemble.

In practice, today most of the labor is invested in the adaption work of *concrete* electric devices of concrete vendors, model types and serieses. Therefore the feasibility of our approach can only be proved by applying it to this concrete practice.

In course of such an implementation activity, there will be two axis for further concretization: (1) considering and exploring *variants* of the existing approach, and (2) completing the overall test definition architecture by further (more basic and smaller) specifications (cf. figure 14).

The main specification text of TUB-TCI, i.e. the specification as presented in section 3, did present just *one consistent* model (referred to as „*main model*“ in the following).

For sake of readability, possible variants and alternatives (w.r.t. details, but also to fundamental decisions) have mostly not even been mentioned when presenting the main model in section 3.

In the first section we therefore list some possible variants which appeared in course of the modeling process, and which seem to have fundamental significance.

The second section briefly describes some related specifications, which would be helpful when implementing TCI.

5.1 Open Issues, Variants in Design Decisions

5.1.1 Locally Generated Identifiers

The centralized allocation of identifier values for dynamically created actors (= *LUID*s) has been introduced in the main model as a strong means for providing overall referential integrity, cf. section 3.10.1.

Since it does introduce substantial overhead even in trivial situations, a modified approach could define like

$$LUID \;==\; NodeIdent \times \mathbb{N}$$

where the second component of *LUID* is a numeric identifier allocated locally by the NodeServer, whenever a local factory's creation method is called. In this case the hosting node of any actor would be identified by the value of its *LUID* immediately. The initial `lookup()` request to CAS would be omissible.

But then the responsibility for referential integrity of *LUID*s must be provided by the *NodeIdent* allocation mechanism, e.g. by using fresh *NodeIdents* for each `reset()`-epoque. The loss of synchronization w.r.t. actors and *LUIDs* would then be trapped as a node-addressing error on the trns-level.

5.1.2 Finer Control of Reset Levels

For a finer control of reset one could distinguish different reset levels, e.g.

- reset trns layer, invalidate all *TAID*s.
- delete all actors, invalidate all *LUID*s.
- terminate all active threads representing RBDs, and reset all actors, but keep all actors allocated and in the „`living`“ state (cf. figure 11).

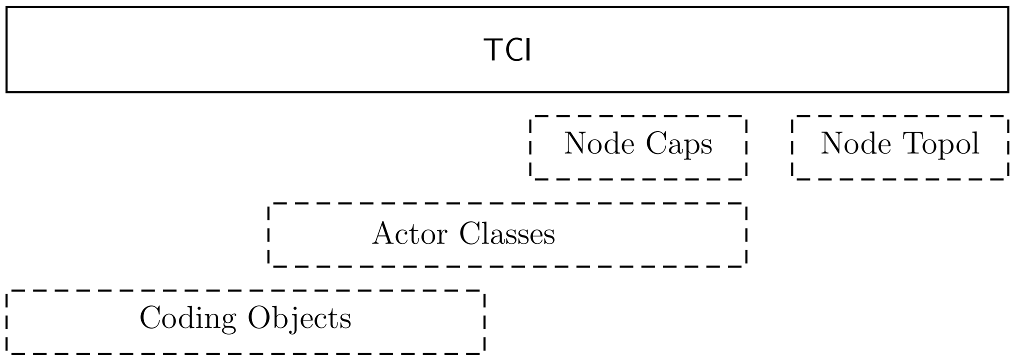
For *large* test ensembles these partial resets could become appropriate w.r.t. performance. This will become apparent not without some experience in concrete experiments.

On the other hand this change of concept would make all analysis and proofs w.r.t. consistency and safety more complicated or even infeasible.

5.1.3 „External“ Reuse of Actor Factories

The main model allows factories to create additionally (more than one) „sub-actor“ subsystems as auxiliary devices, when explicitely called to create one single „main actor“, see 3.10.2 above. In the main model there is a specialized call from Factory to CAS, allocating a sequence of *LUID*s for a given sequence of actor classes, and declaring them to be sub-actors.

Figure 14 TCI as Part of a Larger Architecture



A different approach would be to treat the creation of sub-actors by the factory of the main-actor *totally canonical*, i.e. letting the factory call the factories for the sub-actors „externally“, exactly like a top-level „create“ call from any RBD.

This would have the advantage of more *re-usability*, since factories of complex actor subsystems can reuse more primitive devices (timer, ports, clocks) from third party vendors. The price to pay is that all run-time communication from main-actor to sub-actors has also to be performed by the „official channels“, i.e. TCI-communication.

Please notice that in this case (a) either the CAS has no information about the sub-actor relationship, or (b) the protocol of the `create()` message must be extended by the corresponding data.

5.2 Related and Future Specifications

In a first implementation of TUB-TCI, many definitions and parameters will be realized in a „hard-wired“ manner.

Nevertheless further specifications of semantic models and their encodings are desirable to reach full flexibility and a higher degree of possible automation. The necessary auxiliary specifications are shown in figure 14 and will be briefly described in the following sections.

5.2.1 Encoding of Actor Class Definitions

In the main model the Actor Class values are communicated without being realized on the object level of the language. If using only the main model, the NodeServers have to decode the identifying name of the actor class in any `create` request „manually“ and call the appropriate Factory in a hard-wired manner.

The correctness of configuration parameter values is checkable by the factory also only by hard-coding, — the same is true for all verifications of parameter values done in advance by TM or others.

It is desirable (for automated allocation of resources, automated validation of parameter value combinations, definition of local negotiation mechanisms etc.) to realize the actor class definitions also on the object level of the language. For this purpose the XML encoding rules must be enhanced only by some encoding for the *constraint expressions* contained in the *Z-schemas*, which define the actor class. For this purpose some appropriate existing standard mapping should be chosen.

5.2.2 Coding Objects

For a more precise declaration of the High-Speed Run-Time Communication Channels (= HsLinks) the transmitted data type and its *encoding* must be representable in the declaration language.

The further imposes no problems, since a complete language for type and abstract data construction must be available anyway. But the declaration of the *encoding* is still totally open.

From the viewpoint of TUB-TCI, different encodings would be represented by a set of „Coding Objects“. There must be (1) a set of predefined Coding Objects, and (2) a mechanism for constructing new Coding Objects, i.e. some means offered by the object language.

The existing standardized approach of ECN ([ecn01]) seems not sufficient, and a natural and clean definition of a language for encoding construction is still a major issue for R&D.

5.2.3 Node Class Capabilities Description Format

The two pre-going extensions are required by the next specification module (in figure 14 called „*NodeCaps*“) which allows to declare the capabilites of the hardware nodes contained in a given Test Ensemble (TEns) : The definability of actor classes, as introduced above, implies the definability of Factories, needed as the central capabilities of each hardware node. Additionally, the (*Z*-based) constraint language can be used to specify e.g. upper limits on the number of instances of certain factories.

It is an open issue which means for *abstraction* are practically needed with this declaration language, e.g. if there is a class system with inheritance, or a class-less object system with „copy-down“ inheritance, which kind of parameterization is appropriate, etc.

A similar mechanism is needed for *Busses* (on different levels of abstraction, i.e. from data-link-level wires to OS-level message channels). This declaration language will probably substantially import from the Coding Object declaration language.

5.2.4 Node Topology Description Format

As soon as Node Classes (including the set of hosted Factories) and the classes of the connecting Busses are describable, a language for describing the topology of a total Test Ensemble (TEns) is definable, and seems highly promising w.r.t. further automization.

Acknowledgments

The author wants to thank his colleagues Baltasar Trancón-y-Widemann and Jacob Wieland for support and discussion. Without them this paper would not have been written.

Special thanks also to Ina Schieferdecker and the colleagues from testing technologies, Berlin.

References

[BCPR99] Mohammed Bennattou, Leo Cacciari, Régis Pasini, and Omar Rafiq. Principles and tools for testing open distributed systems. In *IFIP TC5 12th International Workshop on Testing Communicating Systems*. Kluwer Academic Publishers, 1999.

[Büs03] Robert Büssow. *Integrating Z and Model Checking*. PhD thesis, Technische Universität Berlin, to appear 2003.

[ecn01] ECN — Specification of Encoding Control Notation. Technical report, ETSI TS 101 969 V1.1.1, Sofia-Antipolis, May 2001.

[GCI96] *Generic Compiler/Interpreter Interface*. INTOOL CGI / NPL 038 (v.2.2), december 1996.

[Gri99] Wolfgang Grieskamp. *A Set-Based Calculus and its Implementation*. PhD thesis, Technische Universität Berlin, 1999.

[mic] microsoft research/FSE. *AsmL 1.5 online documentation* `http://www.research.microsoft.com/fse/asml/doc`.

[San98] T. Santen. On the semantic relation of Z and HOL. In *Proceedings of the 11th Int Conf. on the Z Formal Method, — ZUM'98*, LNCS, pages 96–115. Springer, 1998.

[soa00] Simple object access protocol, SOAP. Technical report, W3C, `http://www.w3c.org/TR/SOAP`, 2000.

[Spi92] J.M. Spivey. *The Z Notation — A Reference Manual*. Series in Computer Science. Prentice Hall International, 2 edition, 1992.

[tci03] TTCN-3 Control Interface (TCI). Technical report, ETSI Working Draft, Sofia-Antipolis, to appear 2003.

[Tör99] Maria Törö. Decision on tester configuration for multiparty testing. In *IFIP TC5 12th International Workshop on Testing Communicating Systems*. Kluwer Academic Publishers, 1999.

[tri02] TRI — the TTCN-3 Runtime Interface. Technical report, ETSI TR 102 043 V1.1.1, Sofia-Antipolis, April 2002.

[tsp97] Test Synchronization Protocol 1 Plus (TSP1+) Specification. Technical report, ETSI TC-MTS, ETSI Standard ES 201 770, Sofia-Antipolis, Jan 1997.

[ttc01] Methods for Testing and Specification (MTS); Part 1: TTCN-3 Core Language. Technical report, ETSI ES 201 837-1 (V1.0.11), Sofia-Antipolis, May 2001.

[UK99] Andreas Ulrich and Hartmut König. Architectures for testing distributed systems. In *IFIP TC5 12th International Workshop on Testing Communicating Systems*. Kluwer Academic Publishers, 1999.

[VGSB$^+$99] Theofanis Vassiliou-Gioles, Ina Schieferdecker, Marc Born, Mario Winkler, and Mang Li. Configuration and execution support fort distributed tests. In *IFIP TC5 12th International Workshop on Testing Communicating Systems*. Kluwer Academic Publishers, 1999.

[xer02] ASN.1 encoding rules — xml encoding rules (xer). Technical report, ITU-T Rec X.692, 2002.

[xml02] W3C Candidate Recommendation, `http://www.w3.org/XML/Schema`. *XML Schema*, `http://www.w3.org/XML/Schema`, 2002.