TUB-TCI
An Architecture for Dynamic Deployment of Test Components

Markus Lepper, Baltasar Trancón y Widemann, and Jacob Wieland

Technische Universität Berlin, Fakultät IV, ÜBB
Institut für Softwaretechnik und Theoretische Informatik, Sekr. FR 5–13
Franklinstr. 28/29, D–10587 Berlin
{lepper,bt,ugh}@cs.tu-berlin.de

Abstract. The test definition language TTCN-3 is currently under standardization by ETSI/ITU-T. Its intended field of application is testing and performance measurement of communication hard- and software. TTCN-3 does include mechanisms for specifying remote hardware access and for distributed execution of testing code components.
But for running compiled TTCN-3 code distributed onto distinct nodes of different vendors an architecture is needed which offers standardized means for dynamic, program controlled deployment, configuration and status inquiry of active and passive resources.
TUB-TCI is a proposal for such an architecture, characterized by (1) totally generic definition of component classes, (2) coexistence of standardized (XML based) and specialized (high-speed) communication channels, (3) a model-based, strictly formal definition given in Z and (4) a simple, minimized but powerful execution model.
Especially because of the integration of non-standard, high speed data channels TUB-TCI seems applicable for dynamical routing of real-time signals in general, beyond the field of test execution.

Keywords: Dynamic Deployment, Conformance Testing, XML based configuration.

1 Design Principles and Features of TUB-TCI

1.1 The TCI Problem in General

TCI stands for "Test Configuration Infrastructure" and is a topic discussed by a recently installed working group of ETSI, continuing the work presented in [6]. Protocol conformance testing and performance measurement — as far as communication technology is concerned, but possibly also in other fields — will increasingly have to deal with heterogenous ensembles of hardware nodes, in which tests are performed by co-operating pieces of code *distributed* to distinct hardware nodes, each of them may be based on very different technologies.

Any compiler, when translating a so called "abstract test suite" ("ATS") written e.g. in TTCN-3 (cf. [8]) to a collection of pieces of executable code ("executable test suite", "ETS"), can easily do this for different execution platforms.

D. Hogrefe and A. Wiles (Eds.): TestCom 2003, LNCS 2644, pp. 279–294, 2003.

But to really make these code objects co-operate, the pure language definitions and their semantics do not suffice. Indeed an infrastructure is needed which provides additional information concerning the concrete set-up, and which allows user data and control information to be passed between different nodes in a transparent way.

This arises from the following basic contradiction:

- Advanced concepts of testing, like in TTCN-3, provide the means for *dynamic*, program-controlled creating, deleting, configuring and linking test components. This feature will become more and more important, especially for automated, batch driven "over-night" tests, as well as for periodically scheduled, automated in-field "online-tests" in dynamically changing topologies.
  So the actual test code execution needs some information concerning (1) the specific capabilities and API definitions of the involved hardware nodes, their current topology and the currently valid set-up of addressing and routing, and (2) about the creation and configuration commands for certain hardware resources (timers, ports, local I/O-devices) and the required parameterization information, which is specific for the type and vendor of the hardware device.
- Both requirements conflict with the requirement of *abstractness*, i.e. that the same abstract test suite specification and the corresponding compiled code should be able to run in different hardware settings of divergent topologies: neither (1) the deployment strategies, which need information on the current hardware configuration, nor (2) the driver specific parameters should (or even: can) be contained in the source text of the test program.
- Consequently, there has to be a kind of "merging of semantics": The ETS-code will provide certain pieces of information, concerning e.g. "abstract timers", "abstract ports", etc., which are complete in the sense of the semantics of the abstract test suite, but only more or less sufficient w.r.t. the needs of the addressed particular hardware drivers. They must be completed by the TCI infrastructure, using some strategic knowledge concerning the concrete set-up, and passed to the different "drivers" of the selected hardware node, which are the only subsystems able to really perform the allocation and configuration.

Currently all these issues are addressed by the *hand-coding* of so called "test adapters", pieces of code which realize the mapping from the abstract semantics of e.g. TTCN-3 to the concrete interfaces of the concrete hardware setting. These code objects can be re-used only in a limited way, so the work has to be done from scratch for each family of test situations. This work is a source of further possible *errors* in the test process, and can hardly be considered intellectually challenging engineering.

A central intention of TUB-TCI is to relieve the implementor of the test adapter by some ability of *self-organization* of the underlying infrastructure, establishing means for declaring the additionally needed pieces of information in a generic way.

1.2 Basic Paradigms and Current State of TUB-TCI

These are the issues which must be addressed by any TCI, and TUB-TCI is a possible solution proposed by the authors. Its central design goals are

- Versatility,
- Robustness, and
- Precise Specification.

It may further be characterized by

- Co-existence of different paradigms, e.g. central and distributed knowledge, unified and specialized communication channels.
- Being equally well suited for interactive and batch driven testing.
- Preserving referential integrity.

TUB-TCI currently exists as a complete model-based specification (cf. [3]). In contrast to the official TCI specification by ETSI (cf. [4]), which concentrates on defining APIs callable by the executable test suite, our approach also defines the *behaviour* of the single subsystems and the rules of their co-operation, and thus specifies the behaviour of the total system.

There are severe advantages of such a model based specification which uses some abstract but (potentially) executable language:

- It allows to concentrate on the *intended functionality* and its realization by state transitions and message interchange operations. In contrast, when using a concrete programming language for modeling, i.e. "implementing", more than half of the text would have to deal with the specific idiosyncratics of this language, e.g. encoding of type sums, process scheduling, memory management etc.
  So the specification text turns out to be much shorter and clearer than the text of a conventional implementation, but nevertheless *is* a kind of implementation. So e.g. the grade of robustness of the rules could be tested by paper-and-pencil evaluations, in which human intention simulated the behaviour of "malicious" test components.
  The final translation of the mathematically defined state transitions (and the pure functional auxiliaries) into a conventional programming language can in most cases be done in a rather straight-forward way. Here any programming language and any operating system can be chosen, since the (non time-critical) configuration information is exchanged as standardized XML fragments.
- A model in an abstract, mathematically founded language should allow a wider and deeper *discussion* in the community, since basic mathematics are a language known to every engineer and project manager, — independent of her/his experiences with distinct programming languages.
- If the appropriate tools for the chosen modeling language are available, execution ("simulation") and tests can be performed on the mathematical model

immediately. Even *automated verification* of certain properties of the specification are possible, using model-checking techniques, which is practically impossible with general-purpose programming languages[1].

TUB-TCI is intended as a base for *discussion*. While being complete, consistent and ready to work, principal decisions do — of course — have some alternatives, and we do expect revisions when creating the first implementation.

The project has already proved that it is possible to define the behaviour of a complex and generic architecture *completely* through all system layers and all execution phases by precise mathematical means, provided that the level of *abstraction* is chosen for each system layer appropriately.

1.3 Principles of the Architecture

TUB-TCI assumes distributed testing to be performed by collaboration of distinct *subsystems*, hosted on the same or on different hardware nodes. Each test session is an alternating sequence of preparation phases (TPrep), in which components are installed and hardware resources allocated, and test execution phases (TRun), in which the behaviour of the total system is under control of the running "executable test suite". The main issue of any TCI design is that the state of the whole system is *dynamic*, that is, allocation, installation and control of components may happen under program control, i.e. in TRun.

TUB-TCI does *not* specify any strategic knowledge needed for deployment decisions, nor does it define any concrete classes of devices. However, it precisely defines the generic means for "installing" these pieces of information into a system, — the former by run-time services requested or received by a "test manager", the latter by a formalism for declaring new device classes.

TUB-TCI is specified by giving its operational semantics as a collection of transition rules. Most of these involve two subsystems, thereby defining the possible *message passing*. These rules and the data space they operate on are given as Z formulae[2]. Table 1 completely lists all services offered by all categories of subsystems.

All message formats are given as Z schemata, too. As soon as message passing crosses the boundaries of hardware nodes, it is implemented as an exchange of XML fragments. For implementation or practical standardization some canonical representation for Z schemata as XML schemata will have to be chosen. The Z, and not the XML representation is first-class resident in the TUB-TCI specification, because the dynamic semantics can only be formulated in the former.

[1] Due to the limitation of resources, these goals could not be addressed in our project and are left to future work.

[2] The notation used is indeed a *derivation* from Z: Strict requirements needed for the mathematical foundation of the Z semantics could be discarded, so that some shorthand notations (e.g. for type sums, step semantics, parallelism, reflection, i.e. converting schema *definitions* from/to data *values* describing schemas, and call of external driver functions) could be added for the sake of readability.

W.r.t. the work of the authors, the appropriate structure and semantics of the meta-language is indeed a central topic of *research* on its own.

TUB-TCI is *minimal*, as it defines the minimal required, specialized subset of the functionality of an "object broker", a "routing mechanism" and a "communication protocol". So TUB-TCI indeed does contain a "micro-CORBA", "micro-IP" and a "micro-TCP". This does in no case imply that we want to re-invent existing technologies, — contrarily: the state transitions specified on these layers of TUB-TCI can easily be mapped on existing implementations coming from "large", general purpose implementations (TCP/IP, CORBA), or can be implemented in few lines of code from scratch.

Because of this minimality, the specification of TUB-TCI does span the whole range of architectural layers, from top-level declaration of "semantic types" down to the lowest level containing the (loosely specified) primitive bus driver communication handshakes.

The architecture consists of the following layers:

- Basic node initialization and communication mechanism.
- Meta-Services for defining new actor classes and their access modes.
- Runtime services for creating, configuring and destroying active and passive entities.
- Runtime services for user data transmission.

2 Subsystems Forming a TUB-TCI Setting

In TCI it is assumed that a multitude of hardware nodes co-operate. These nodes are connected by communication channels of widely divergent technologies. On each node there is a "testing sandbox": Parallel to all other activities running on a node, this sandbox contains all testing or measuring code, which is controlled by TCI.

All activities in TUB-TCI happen as *Service Requests* between two *subsystems*, which are implemented by message exchange. We define different categories of such subsystems, the most important of which are characterized as follows:

2.1 TM = Test Manager

Even in case of distributed execution, testing in practice will always be controlled by one central instance, which launches, starts and stops the diverse test phases and calculates the intermediate and final verdicts and results. This is true for batch driven as well as for interactive test execution.

Such a central instance is modeled by one single instance of the "Test Manager" (TM) category of subsystems. A TM must be "external" to each TCI concept, as its behaviour should not and cannot be specified therein.

Contrarily, the currently active TM is an arbitrary program, of which only two aspects are specified:

(1) Each TM is an active component issuing service requests for installing and controlling other active or passive components (i.e. it is a subclass of the Actor subsystem, and *must* fulfill the role of an RBD, see below, section 4. Additionally

it *may* act as signal drain, i.e. an RBQ, see below section 3.2.2). More precisely: The TM is the *only initially active* component, and as such the source of all other activities in a TCI system.

(2) A TM *offers* only one single service called "`decideNode()`", which, given an indication for the class of which a new component shall be created, together with an appropriate parameter set, delivers an indication for the hosting node of this new component and a (perhaps) modified parameter setting.

So TM is specified only as being the subsystem containing all deployment strategies and the total routing information, — the implementation of both is (currently) outside the scope of each TCI.

2.2 CAS = Central Access Server

As it is with TM, there is always only one single instance of CAS in each TCI setting. Contrarily to TM, the behaviour of CAS is totally specified by TUB-TCI.

The (single instance of) CAS caches all deployment information calculated by TM, calculates unique IDs for all newly created objects of global scope and holds a catalog of these for answering the corresponding inquiries, controls the sequentialization of `reset()` commands, keeps track of the initialization state of all nodes and gathers all verdict and error messages for passing them to TM.

In contrast to TM the CAS must be hosted on a node which is *reachable* from all other nodes (c.f. section 3.1)[3].

2.3 NodeS = Node Server and Factories

On each node of the TCI setting there is exactly one Node Server running. The NodeS has total control of the testing sandbox of its hosting node, in which all test components will be living. It is also responsible for bootstrapping of the underlying basic communication layer (Trans, see section 2.5) and its routing information. Furthermore, it is the recipient for all `DOcreate()`and `DOdelete()` commands, which install new subsystems on this node.

For really performing the creation, the NodeS delegates `DOcreate` requests for all new subsystems of a certain class to the corresponding Factory. There is one factory running on each node for each class of subsystems which can be created on this node.

2.4 Actors = Dynamically Created Active and Passive Subsystems (Components)

All subsystems which can be dynamically created or allocated are subsumed as "Actors". This includes hardware resources (IO ports, timers or their respective drivers), code images loaded onto a distinct node, and also running code, e.g. "jobs" or "threads", created by instantiating parts of these images.

[3] Of course the node hosting the CAS must also be able to reach the node hosting the TM.

Actors realized by active running code can act in two different roles, an active and a passive one, called RBD and RBQ resp.[4].

RBDs are the only "user code" running in a TUB-TCI environment and the only sources of activity.

The passive RBQ interface is offered by all those components which can act as signal *drains*, that is they can consume a data stream generated by some other hardware Actor or by some running RBD.

2.5 Trans = Inter-Node Communication and Node Initialization

As induced by the TTCN-3 language, on "user data level" any TCI architecture must support synchronous as well as asynchronous communication. Furthermore, on "system level" communication necessary for component control and configuration must be realized.

Both levels of communication are mapped to a basic layer called Trans, which realizes the (small) necessary portions of "data link", "transport" and "network" layer of the ISO-OSI model.

We support two primitive communication acts: unsolicited, "UDP-like" datagrams, and solicited datagrams, i.e. one request and one single response.

When sending an unsolicited datagram, control returns to the sending component code immediately. When sending a solicited datagram, the communication is synchronous. A time out duration value has to be given and control returns to the sending code either with the return message or with a timeout indication. A central feature yielding the *robustness* of TUB-TCI is that there is only one single source for time-out generation: the Trans subsystem of the node hosting the client.

3 Scenarios of Dynamic Behaviour

3.1 Node Topology, Node Initialization and Routing

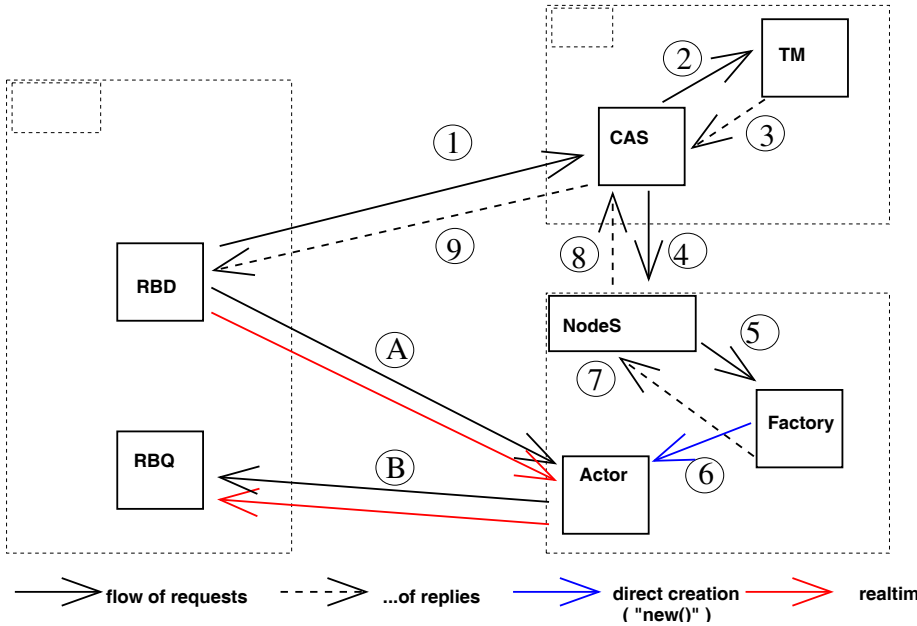
The hardware nodes forming a TCI setting must be connected by some communication channels. It is *not* required that each such channel is bi-directional. It *is* required, however, that each node has at least one input and one output channel, and that the node hosting the CAS is initial and final, i.e. can reach any other node and is reachable by any other node.

When powering up (the testing sandbox internal to) a distinct node, this node is in an `uninitialized` state. The only function it performs is listening on dedicated ports for a `loadRouting()` service request. This message contains (1) the assignment of one globally unique identifier for this node (*NodeId*), (2) a routing table indicating for each *NodeId* the bus driver and bus address, if the corresponding node is directly reachable, or the *NodeId* of the node which is to

[4] The names RBD and RBQ are historically determined, cf. [6], and stand for "Runtime Behaviour / Dynamic" resp. "Runtime Behaviour / Queue".

Table 1. Offered Services

TM	called by CAS :	`decideNode()`
CAS	called by TM :	`CASreset()`
		`bootNodes()`
	called by RBD :	`getGlobalParameter()`
		`create()`
		`delete()`
		`setverdict()`
	called by Trans :	`lookup()`
	called by Factory :	`registersubactors()`
		`deleted()`
		`desmudge()`
NodeS	called by CAS :	`reset()`
		`loadRouting()`
		`sendHdwStateInfo()`
		`startsession()`
		`stopsession()`
		`DOcreate()`
		`DOdelete()`
		`HScreatelink()`
		`HSregisteroutlink()`
Factory	called by NodeS :	`DOcreate()`
		`DOdelete()`
Actor	called by RBD :	`setconfigparams()`
		`getconfigparams()`
		`RToperation()`
RBQ	called by QAS :	`RToperation ( putQ() )`
		`HSputQ()`
QAS	called by RBD :	`subscribe()`
		`unsubscribe()`
		`HSsubscribe()`
		`HSunsubscribe()`
	called by Actor(-demon/-irqHdl)	`value_event()`
		`HSvalue_event()`
Trans	called by Trans (= 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()`



① – ⑨ : Sequence of Requests/Replies for creating an Actor.
Ⓐ : Run-Time Control on Actors
Ⓑ : Real-Time Signal Flow from Spontaneous Signal Sources.

Fig. 1. Subsystems and Collaboration Diagram for creating a new actor

be used as a gateway, and (3) a routing table assigning a set of *NodeIds* to each *broadcast group*.

The initialization and re-initialization procedures can be rather critical and complicated, depending on the topology of the network: Direct confirmation of successful initialization or reset may not be possible until third nodes are initialized, which are needed as gateway to reach the CAS node. Figure 2 just wants to give an impression how complicated the correct schedule of init-commands (upper sequence in the figure) and reset-commands (lower sequence) can turn out, and that in a complicated topology both schedules are neither identical nor just simply inverse.

Furthermore the initialization sequence may have to consider latency requirements. The concrete schedule of `loadRouting()` and `reset()` messages requires strategic knowledge and is left to the TM, external to the TUB-TCI specification. The specification *does* require, that every `reset()` must be distributed to *all* nodes in the setting, so that only a *total* reset is legal for sake of robustness.

3.2 Inter-component Communication

On *application* level, i.e. from the viewpoint of a compiled TTCN-3 code and its runtime library, runtime communication happens between Actors, i.e. dynamically created components.

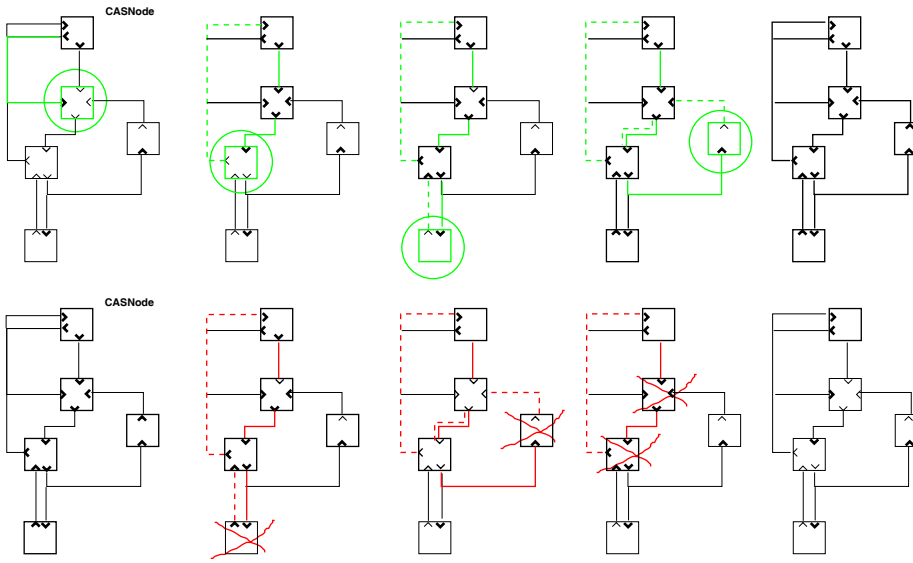


Fig. 2. Sequences of Initialization (◯) and Reset (X) in a non-trivial Network Topology

We distinguish two totally disjoint flavors of communication, each of which can be realized by two totally different mechanisms:

3.2.1 Single-Drain Communication. Single-drain communication channels are related to one single actor which is the *target* of all messages. These messages are operation control commands issued by a parallel running component. Examples are: `start`,`stop` and `launch` commands of timers, `write` commands to outgoing data ports etc.

The corresponding messages can be sent solicited or unsolicited.

The timing of events in these channels is typically irregular: Multiple active components (RBDs) independent from each other can send messages to one single drain spontaneously.

3.2.2 Single-Source Communication. Single-source channels are related to one single actor which is the *source* of all communication. They are mainly used for realizing a stream of user data, e.g. ticks generated by a `timer` or incoming messages received by a `port`.

The corresponding messages can only be sent unsolicited.

The frequency of these messages is often *regular*. More than one listener can *subscribe* a distinct source, and will from then on be notified of each event by their "RBQ" interface, until they *unsubscribe* the channel.

Both kinds of communication can be realized in two ways:

3.2.3 User Level Communication by Service Requests. The first possibility of runtime user level communication is realized by mapping it to the standard service request mechanism:

In the single-drain case an `RToperation()` message is sent by the emitting RBD to the target Actor which shall be controlled or configured. The argument of this message is a further schema representing the distinct action to perform. All possible real time control messages and their formats are defined with the class of the target Actor, see section 3.3.

In the case of single-source the RBD of the Actor sends a `subscribe()` or `unsubscribe()`message to a subsystem called QAS(= Queue Access Server)[5]. This message is parameterized with the id of the Actor the outgoing data stream of which shall be consumed.

The driver of this (passive) component will simply generate a single `value_event` message to QAS each time it generates or receives a data event.

On the other side the Actor the active code of which performs such a subscribing must also implement the passive RBQ interface. This means that the among the set of `RToperation()` it understands there must be the `putQ()` command, which pushes the data into its data input queue.

The scheduling and dispatching of all these messages is done automatically and totally dynamic by the operational semantics of TUB-TCI. The signal flow will always be minimized in so far as each node hosting a subscriber and/or needed as a gateway will only receive one single copy of each data event.

All data in this flavor of communication is encoded as an XML object, which is derived from the corresponding schema defined with the Actor Class. So there is unlimited compatibility: Each RBQ can subscribe each source, and one single RBQ can subscribe multiple sources simultaneously, de-multiplexing the incoming data using the tags contained therein.

3.2.4 User Level Communication by High Speed Channels. Secondly there is the possibility for a given node and its factories to offer specialized *high speed channels* (HS-channels).

These are communication channels which bypass the Trans layer, but are defined directly on "driver level". In contrast to the "normal" communication they may be based on specialized hardware links between nodes (e.g. time slot busses), may use specialized addressing protocols and transmit *binary coded* data, maybe in a proprietary format. Both flavors of communication (single-drain component control or single-source data stream consumption) can be realized by HS-channels.

Their intended purposes are (1) the monitoring of data with high bandwidth generated on a external node, and (2) the transmission of single events which should reach the target with minimal latency.

[5] For sake of this paper, QAS can be considered to be one single virtual subsystem, which is directly reachable for any RBD. Indeed it will be *implemented* on each node as part of the node's NodeS.

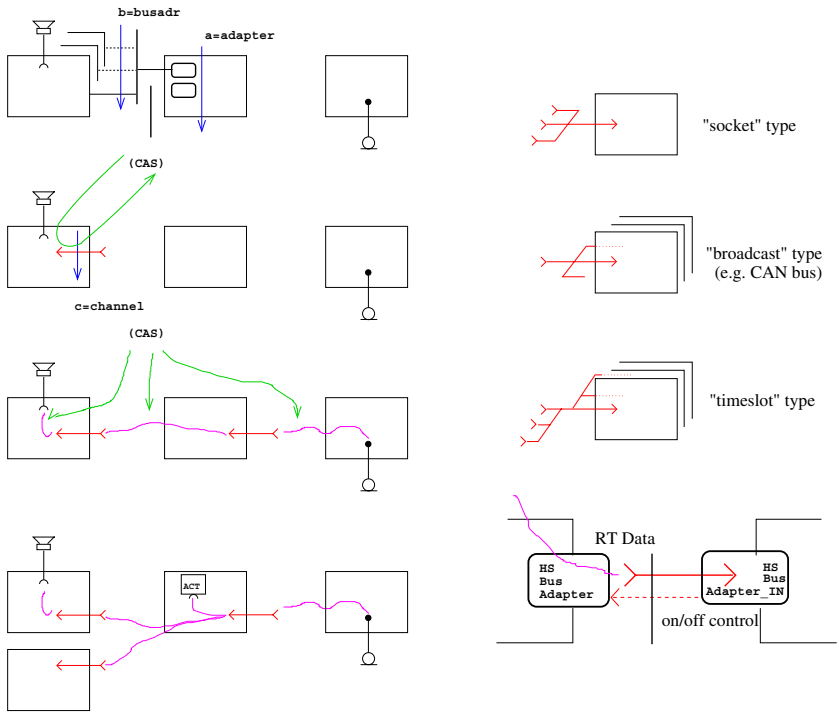


Fig. 3. Creation of HsLinks / Variants of HsLinks determined by the underlying Technology

The principle of total dynamic configuration and scheduling naturally contradicts these purposes. So here we have chosen a different strategy: All HS-channels can (and should) be allocated and configured by the TM *in advance*. When linking and loading the compiled code it has to be parameterized with the IDs of the HS-channels corresponding to the source level communication statements.

Only in those cases where the HS-channel is bi-directional, the `HSsubscribe()` and `HSunsubscribe()` messages issued by the consumer do have an effect: This effect is just "switching on and off" the data stream from the sending node, as soon as the first consumer subscribes or the last consumer unsubscribes. This back channel should of course be implemented also on driver level for sake of optimal performance and lowest latency.

Figure 3 tries to depict that the main achievement of TUB-TCI lies in the abstraction from the different flavours of low-level communication (many-to-one, one-to-many, etc., as depicted in the right column of the figure), and from the different way of addressing the nodes. The left column shows the typical two-stage approach for establishing HS-channels by the CAS (controlled by strategic knowledge of the TM): First the "logical in-ports" are allocated, thereby resolving all specific requirements of the lower communication layer. So in a second step the concrete connections into these ports can be established in a uniform way.

3.3 Actor Class Declaration and Instantiation

As mentioned above, the declaration of the classes of which Actors can be created on a given node is outside the scope of TUB-TCI. Even most basic foundation classes like "active code" or "thread" are unknown to this layer of architecture. In fact only the mechanisms for defining and deriving classes is part of the TUB-TCI.

Each Actor Class is given as a Z schema, and declares for each instance (1) the collection of configuration parameters and (2) the runtime operations applicable. The latter is just a *free type* describing the possible control messages the Actors of this class understand during runtime. The former gives for each parameter a basic type, arbitrarily chosen constraints on the possible values, and the "update allowance", which describes if this parameter cannot be set at all (`CO` and `RO`), can be written once when creating the Actor(`WI`) or re-written during setup time (`RW`) or can be changed even at runtime (`RWRT`).

3.4 Example

Let us illustrate this mechanism by the example of a "`timer`" Actor Class:

First we define the configuration parameters and the real-time operations separately[6]:

PI_Timer

$$\begin{array}{l} minResolution : Duration \\ curResolution : Duration \\ maxValue : Duration \end{array}$$

$$\begin{array}{rcl} RT\_Timer & ::= & \texttt{start} \\ & | & \texttt{stop} \\ & | & \texttt{reset}\langle\langle Duration\rangle\rangle \\ & | & \texttt{read} \Rrightarrow \texttt{timerval}\langle\langle Duration\rangle\rangle \end{array}$$

No we use these both data types and build a schema which includes the schema *ActorClassDescription*:

AC_Timer

$$\begin{array}{l} ActorClassDescription \\ \hline PI \subset unpack\ PI\_Timer \\ RT = RT\_Timer \\ PI\,(\texttt{"minResolution"}).mode = \texttt{CO} \\ PI\,(\texttt{"maxDuration"}).mode = \texttt{CO} \end{array}$$

[6] Please note the slight enhancements and weakenings of the Z notation, used for purpose of readability: The notation $\alpha \Rrightarrow \beta$ simultaneously defines one case of a free type used as service request (α) together with the case (β) of another free type, representing the *corresponding* reply. This correspondence is considered informal, i.e. only for reader's information. The operation "*unpack*" belongs to our reflection tools: it takes a schema and lifts it to a data structure.

Now we can derive the definition of a special timer, offering additional operations and requiring more parameters:

$$
\begin{array}{l}
PI\_Timer\_4711 \;==\; [maxMarkerCount : \mathbb{N}] \\
RT\_Timer\_4711 \;::=\; \texttt{setMark}(\mathbb{N}) \\
\qquad\qquad\quad \mid \texttt{readMark}(\mathbb{N}) \;\Rightarrow\; \texttt{timestamp}(Time) \\
AC\_Timer\_4711 \;==\; [\; extend(AC\_Timer, PI\_Timer\_4711, RT\_Timer\_4711) \\
\qquad\qquad\quad \mid PI\,(\texttt{"maxMarkerCount"})\,.mode \;=\; \texttt{WI} \\
\qquad\qquad\quad]
\end{array}
$$

At last we declare a factory, i.e. a concrete "driver", which probably will be supported only on nodes of dedicated node classes. 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 generic *Factory* schema (cf. figure 4), and (3) further constraints on the configuration parameters.

FACT_timer_x4711_2.2_runs_on_Tektronix_0815

$$
\begin{array}{l}
AC\_Timer\_4711 \\
Factory \\
\hline
V\ \texttt{minResolution} \;=\; sec(0.001) \\
V\ \texttt{maxDuration} \;<\; sec(3600.00) \\
PI(\texttt{"curResolution"})\,.mode \;=\; \texttt{RW} \\
PI(\texttt{"curResolution"}).default \;=\; sec(0.01) \\
sec(0.001) \leq V\ \texttt{curResolution} \leq sec(0.1) \\
\exists\, n : \mathbb{Z} \;\bullet\; V\ \texttt{curResolution} \;=\; n * sec(0.001) \\
V\ \texttt{maxDuration}\ /\ V\ \texttt{curResolution} \leq 2^{31} - 1
\end{array}
$$

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

4 Related and Future Work

4.1 Related Work

Our work on TCI is based on early industrial approaches as found in [2], and did have some influence to the recent development of TRI ([6]), a communication protocol which can be interpreted as one fixed instance of a TCI, but without formal definitions of behaviour.

There are numerous significant theoretical works concerning the algebraic aspects of distributed testing ([1,5,9]).

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

$$\texttt{CO} < \texttt{RO} < \texttt{WI} < \texttt{RW} < \texttt{RWRT}$$

ParameterDescription

$$\begin{array}{ll} type & : \_\_TYPE\_\_ \\ updatemode & : ParameterUpdateAllowance \\ value & : type \\ allowed & : \mathbb{P}\ type \\ default & : type \cup \{\bot\} \end{array}$$

$$default \in allowed \lor default = \bot$$

$$ParameterDescriptionList == IDENT \pfun ParameterDescription$$

ActorClassDescription

$$\begin{array}{ll} PI & : ParameterDescriptionList \\ RT & : \_\_FREE\_TYPE\_\_ \end{array}$$

Actor_Factory

ActorClassDescription

$$\begin{array}{lll} \texttt{->} \ \text{create} & : LUID \times pack\ PI & \rightarrow GR \\ \texttt{->} \ \text{delete} & : LUID \times LUID & \rightarrow GR \\ \texttt{->} \ \text{changeParamas} & : LUID \times pack\ PI & \rightarrow GR \\ \texttt{->} \ \text{dumpParamas} & : LUID & \rightarrow pack\ PI \\ \texttt{->} \ \text{executeRtOperation} & : LUID \times RT & \rightarrow \_\_DATA\_\_ \end{array}$$

// GR = General Result data type, $LUID$ = "local unique id" = a unique id for a dynamically created Actor.

Fig. 4. Basic definitions for Defining Actor Classes

On the other side there is, to the best of our knowledge, only one published approach concerning practical implementation: the industrial implementation of TSP1 [7], as part of the TINA project.

Our approach is somehow in the middle, — it does not cope with reasoning as the former, but it does contain precise specification of operational semantics and genericity, both lacking in the latter.

4.2 Next Steps

As mentioned above, currently TUB-TCI is a concept meant as a base for discussion. A first implementation is planned as part of a research project, and will certainly lead to modifications.

Additionally there are higher, lower or parallel layers of a possible TCI architecture, which still have to be specified:

- The Factories corresponding to these classes are currently assumed to be "hard wired" into the Node Server of the hardware node.
  This corresponds to the fact, that our specification does not yet model nodes, node classes and node vendors: in our declaration example on page 292 the corresponding dependencies are only given informally by choosing the "human readable name" of the factory to be "*FACT_timer_x4711_2.2_runs_on_Tektronix_0815*".
  An ubiquitous, standardized semantic basis for modeling and implementing (!) these relations probably will need the allocation of "*ASN.1 information objects*", cf. ISO/IEC 8824-2, i.e. entities of world-wide unique meaning assigned to vendors by a standardization board.
- This also applies to the *binary* data encoding used for the HS-Channels. Currently the data encoding is totally unspecified and the correct "routing" from sources to sinks is left to the informal knowledge of the TM. Different encodings, even vendor-specific, could be indicated by "information objects", so that Actors realizing *encoding converters* could be inserted automatically and correctly w.r.t. typing.

References

1. 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.
2. *Generic Compiler/Interpreter Interface*. INTOOL CGI / NPL 038 (v.2.2), december 1996.
3. Markus Lepper. TUB-TCI — A Generic Architecture for Distributed Test Execution. Technical report, Berlin, September 2002. `http://uebb.cs.tu-berlin.de/papers/published/TR02-08.ps`.
4. TTCN-3 Control Interface (TCI). Technical report, ETSI ES 201 837-5 , to appear 2003.
5. Maria Törö. Decision on tester configuration for multiparty testing. In *IFIP TC5 12th International Workshop on Testing Communicating Systems*. Kluwer Academic Publishers, 1999.
6. TRI — the TTCN-3 Runtime Interface. Technical report, ETSI TR 102 043 V1.1.1, Sofia-Antipolis, April 2002.
7. Test Synchronization Protocol 1 Plus (TSP1+) Specification. Technical report, ETSI TC-MTS, ETSI Standard ES 201 770, Sofia-Antipolis, Jan 1997.
8. 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.
9. Andreas Ulrich and Hartmut König. Architectures for testing distributed systems. In *IFIP TC5 12th International Workshop on Testing Communicating Systems*. Kluwer Academic Publishers, 1999.