Electronic Notes in Theoretical Computer Science 66 No. 2 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume66.html 18 pages

Scalable System-level CTI Testing through Lightweight Coarse-grained Coordination

Tiziana Margaria [1,2]

METAFrame Technologies GmbH
Dortmund, Germany

Bernhard Steffen [2]

University of Dortmund
Dortmund, Germany

Abstract

We propose a solution to the problem of system-level testing of functionally complex communication systems based on lightweight coordination. The enabling aspect is here the adoption of a coarse-grained approach to test design, which is central to the scalability of the overall testing environment. This induces an understandable modelling paradigm of system-wide test cases which is adequate for the needs and requirements of industrial test engineers. The approach is coarse-grained in the sense that it renounces a detailed model of the system functionality (which would be unfeasible in the considered industrial setting). The coordination is lightweight in the sense that it allows a programming-free definition of system-level behaviours (in this case complex test cases) based on the coarse models of the functionalities. These features enable test engineers to graphically design complex test cases, which, in addition, can even be automatically checked for their intended purposes via model checking.

1 Introduction

System-level testing of communication systems is an intrinsically business-critical issue for all the stakeholders: technology providers, service providers, and customer companies that rely on those systems as basis of their business. It is also a very complex problem, because it involves a variety of technologically heterogeneous subsystems. Although adequate test tools for the unit test

[1] eMail:TMargaria@METAFrame.de
[2] eMail:{Tiziana.Margaria, Bernhard.Steffen}@cs.uni-dortmund.de

of each subsystem are available, an integrated approach is still missing: the system-level tests must be designed and executed almost entirely manually.

We present here a coordination-based integrated test environment which realizes at Siemens the high-level coordination of complex system-level-tests for a scenario of commercial, state-of-the-art Computer-Telephony Integrated systems. Central aspect is here the adoption of a coarse-grained approach to test design, which is central to the scalability of the overall testing environment. This enables an understandable modelling of system-wide test cases, adequate for industrial test engineers, that is based on a *lightweight coordination* model. These features enable test engineers to graphically design complex test cases, which, in addition, can even be automatically checked for their intended purposes via model checking.

In the following, we introduce the concrete application scenario (Sect. 2), describe our coordination-based scalability approach (Sect. 3 to 5), which concerns test case design, test execution, and validation via coarse-grained model checking. Finally we address related work (Sect. 6) and conclude (Sect. 7) with remarks on the benefits and the current perspectives.

2 CTI System-Level Testing in Practice

The application domain concerns *Computer Telephony Integrated* (CTI) systems, i.e. composite (sensitive to platform aspects), embedded (due to hardware/software codesign practices), reactive systems that offer high availability services to a number of clients, and which therefore run on distributed architectures (e.g. client/server architectures). The supported capabilities cover at the moment the collaboration between LAN-enabled midrange telephone switches and a variety of third-party, typically client-server, applications running on PCs. Integration of WAN capabilities and mobile phones is foreseen for the next generation of switches, thus it must be conceptually captured already by today's environment. In any installed scenario, complex subsystems affect each other in a variety of different ways, so mastering today's testing scenarios for telephony systems demands an integrated, open and flexible approach to support the management of the overall test process, from the specification and design of tests to their execution and to the analysis of test results.

As a typical example of an integrated CTI platform, Fig. 1 shows a midrange telephone switch and its environment. The switch is connected to the ISDN telephone network and communicates directly via a LAN or indirectly via an application server with CTI applications that run on PCs. Like the phones, CTI applications are active components: they may stimulate the switch (e.g. initiate calls), and they react to stimuli sent by the switch (e.g. notify incoming calls). System level test investigates the interactions between such subsystems. Typically, each participating subsystem requires an individual test

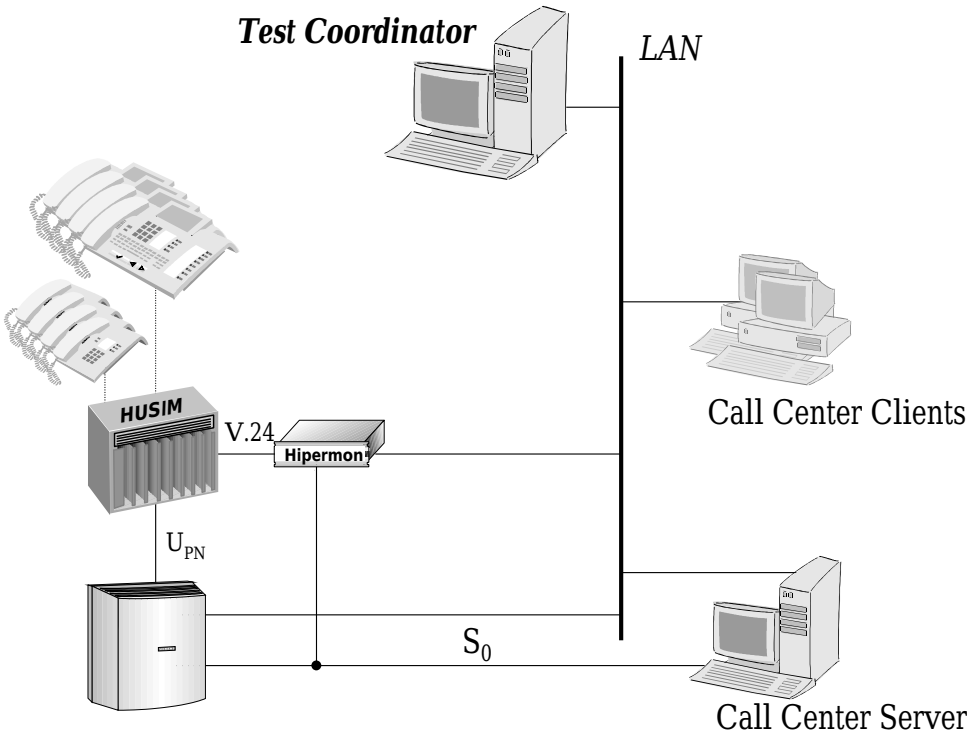


Fig. 1. Example of an Integrated CTI Platform

tool. In the scenario of Fig. 1 three different test tools are needed: *Husim*, an emulator for *UPN Devices* (i.e. telephones), *Hipermon* [6], a telephony and LAN interface tracer, and *Rational Robot* [14], a GUI capture/replay test tool for applications located on the application PCs.

Accordingly, in order to test systems composed of several independent subsystems that intercommunicate, one must be able to coordinate a heterogeneous set of test tools in a context of heterogeneous platforms. This task exceeds the capabilities of today's commercial test management tools, which typically cover only the needs of specific (homogeneous) subsystems and of their immediate periphery.

Traditional formal-methods based test automation approaches fail to enter practice in this scenario because they require a fine granular formal model of the involved systems as a basis. In reality, none of the depicted subsystems has any formal model, but all have a running reference implementation, which is itself a moving target, yet constitutes de facto the basis of all functional and regression testing activities. We thus need an approach capable of developing a formal coordination layer on top of existing black or graybox implementations which rapidly evolve.

Due to the gray/blackbox availability of the systems under test the coordination is necessarily *coarse grained*. Due to the rapid evolution of the systems (with cycles of one week to three months) the coordination must be extremely *lightweight*: there is no hope of having the resources for "reprogramming" new test cases once a subsystem varies. Adaptions and changes have to be easy

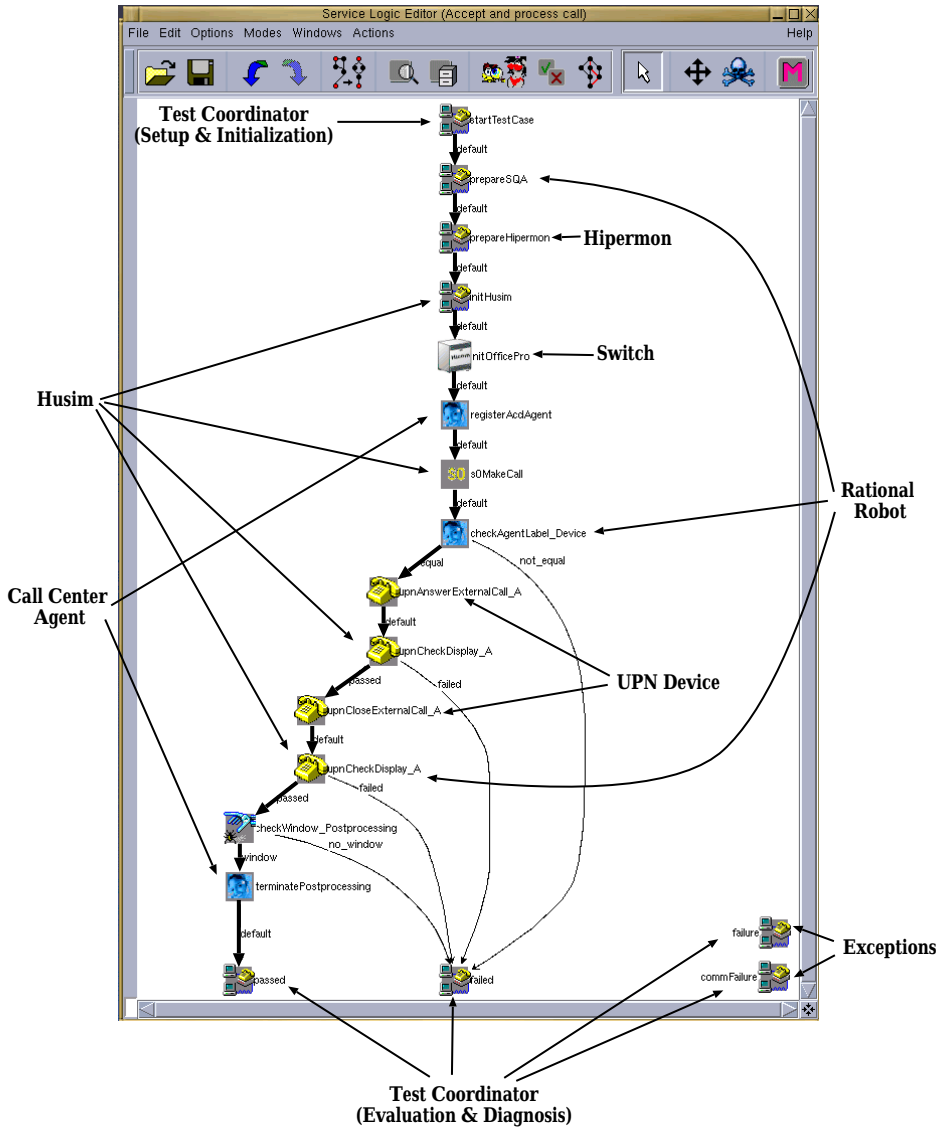


Fig. 2. Test Case in the ITE Environment

and programming-free. Taken together, this defines the 'meta-level' on which

- test engineers are used to think,
- test cases and test suites can be easily composed and maintained,
- test scenarios can be configured and initialized,
- critical test case consistency requirements (including version compatibility and frame conditions for executability) are easily *formulated*, and
- error diagnosis must occur.

2.1 Test Coordination as Superposition

The ITE (Integrated Test Environment) deployed at Siemens contains a dedicated *Test Coordinator* (TC), see Fig. 1 tool which constitutes the system

level test management, organization, and coordination layer of the ITE. The TC is an application-specific specialization for the testing domain of an existing general purpose environment for the management of complex workflows, (METAFrame Technologies' *Agent Building Center* ABC [18]). The ABC offers built-in features for the programming-free coordination and the management of libraries of functional components. This platform also forms the basis of the new release of ETI (Electronic Tool Integration platform) [17,2].

The test coordinator is responsible for the definition and enforcement of complex behaviours which are *superposed* on the system under test. The challenge is precisely how to handle this superposition in an independent, understandable, and manageable way:

- it should be *expressive* enough to capture the coordination tasks, like steering test tools, measuring responses, and taking decisions that direct the control flow within a system-level test case
- it should be *non-intrusive*, i.e. we cannot afford having to change the code of the subsystems, and this both for economical reasons and lack of feasibility: most applications are complete black boxes
- it should be *intuitive and controllable* without requiring programming skills. This implies that we need a lightweight, possibly graphical approach to coordination definition, and that easy and timely validation of coordination models should be available.

In our solution, we have adopted a coarse-grain approach to modelling system behaviour, that accounts for the required simplicity and allows direct animation of the models as well as validation via model checking.

3 Coarse-Grained System Models

Instantiating the TC to cover a tool or a CTI application consists of designing a set of application-specific test blocks that cover the relevant behaviour of the application and are executed during the test runs. The test blocks embody coarse granular functionalities of a subsystem, whose implementation is not further formally described. They constitute the *computational core* of the system, are atomic in the coordination model and are not subject to modifications.

These test blocks are used by test designers to graphically construct test cases by drag-and-drop on the TC canvas. The resulting test graphs are directly executable on a system in the field, and, at the same time, they constitute the formal models for verification by means of model checking.

Fig. 2 shows a typical test graph, which illustrates the complexity of the scenarios. Each test block is marked with the name of the subsystem it controls. Some test blocks control directly subsystems-under-test (e.g. when initializing the switch) while others control the corresponding test tools. It is easy to see that even this relatively simple test case needs to access almost all

participants of the test scenario of Fig. 1 in a varied way, and that even for small configurations (only one PC application) the current practice of manual coordination must require specialized personnel, is tedious, time consuming, by far not exhaustive, and error prone.

The central aim of the ITE Test Coordinator is to relieve test engineers from the manual activities of

- programming test blocks, by largely automating the test block generation, and
- executing the system-level test, by automating the coordinative execution (initialization, execution, analysis, and reporting).

For the design of appropriate system-level test cases it is necessary to know what features the system provides, how to operate the system (and the corresponding test tools) in order to stimulate a feature, and how to determine if features work. This information is gathered by the test experts, and after identification of the system's controllable and observable interfaces it is transformed into a set of stimuli (inputs) and verification actions (inspection of outputs, investigation of components' states). Our coarse-grained approach allows test engineers to capture these user-level test activities directly in terms of coordination-level executables test blocks.

3.1 The ITE Component Model

ITE has a very simple **component model**:

(i) a *name* characterizing the block,

(ii) a *class* characterizing the tool, subsystem, or – for test case-specific blocks – the specific management purpose (e.g. report generation) it relates to,

(iii) a set of *formal parameters* that enable a more general usage of the block (e.g. phone ID),

(iv) a set of *branches* which direct the flow of the test execution in dependence of the results of the test block execution (e.g. equal or unequal for the *CheckAgentLabel* block), and

(v) *execution code* written in the coordination language, typically to wrap the actual code that realizes the functionality.

It is easy to see that the *name, class, formal parameters*, and *branches* of this component model provide a very abstract characterization of the components, which will be used later to check the consistency of coordination graphs. The computational portion is encapsulated in the execution code, which is independent of the coordination level, thus it is written (or, as in this application, generated) once and then reused across the different scenarios (e.g. to test several CTI applications).

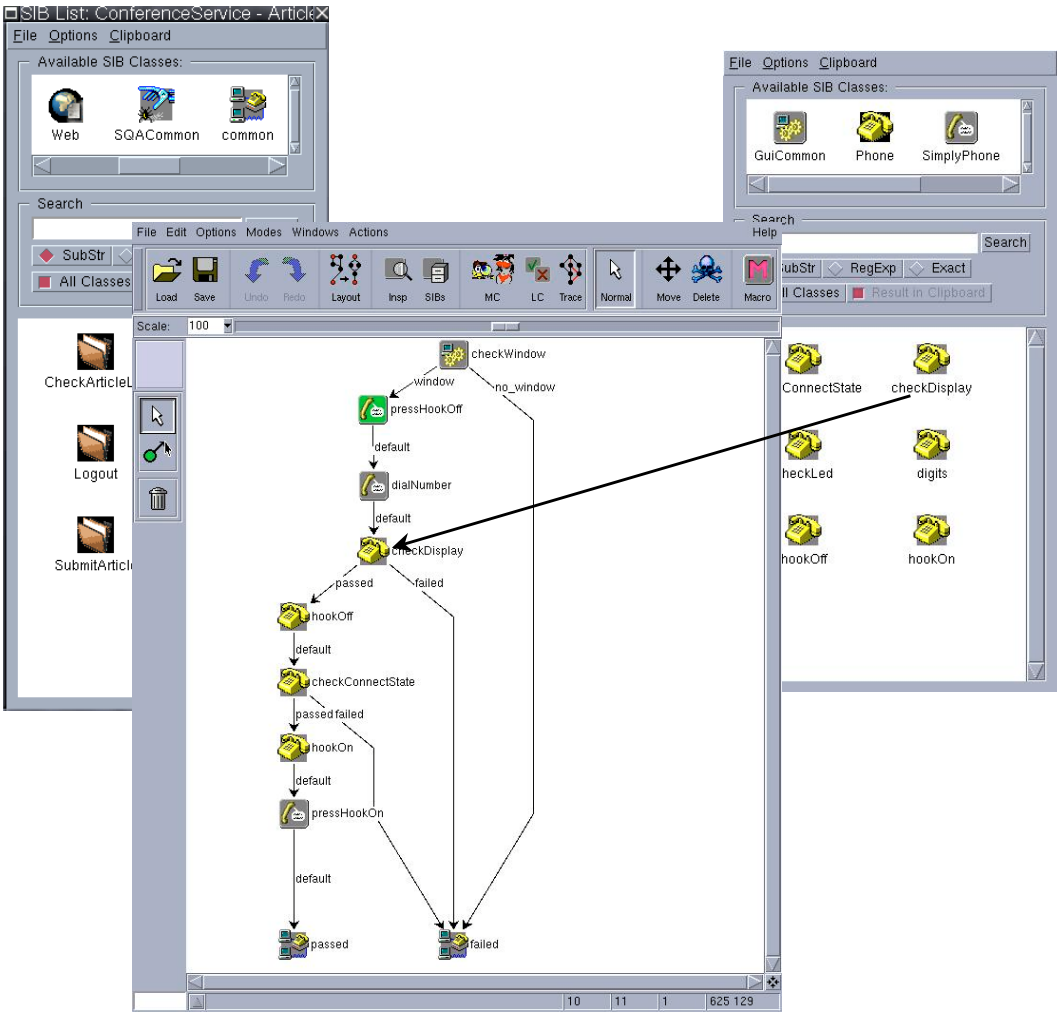


Fig. 3. Fragment of the Taxonomies as SIB Palettes)

3.2 Formal Test Case Models

Test cases are composed of elementary modules, called SIBs (service independent building blocks). The complexity of these SIBs ranges from elementary statements to relatively large procedures steering the routing or application machinery. They are classified in our test design environment in terms of a taxonomy, which reflects the essentials of their profile. A taxonomy is a directed acyclic graph, where sinks represent SIBs, which are atomic entities in the taxonomy, and where intermediate nodes represent groups, that is sets of modules satisfying some basic property (expressed as predicates). Fig. 3 shows a fragment of our taxonomy as it is presented by the ITE test case editor. It shows two snapshots of SIB palettes: on the left we recognize e.g. groups for internet actions (web) and for steering the test tool Rational Robot (SQA-Common), on the right we see groups for telephony activities (GUICommon and Phone).

Test cases are internally modelled as Kripke structures whose nodes represent elementary SIBs and whose edges represent branching conditions:

Definition 3.1
A *test case model* is defined as a triple $(\mathcal{S}, Act, Trans)$ where

- $\mathcal{S}$ is the set of available SIBs
- Act is the set of possible branching condition
- $Trans = \{(s, a, s')\}$ is a set of transitions where $s, s' \in \mathcal{S}$ and $a \in Act$.

Through this non-standard abstraction in our model we obtain a **separation of concerns** between the control-oriented coordination layer, where the test engineer is not troubled with implementation details while designing or evaluating test cases, and the underlying data-oriented communication mechanisms enforced between the participating subsystems, which are hidden in the test block implementation. Our tools support the automatic generation of test blocks according to several communication mechanisms (CORBA [10], RMI [19], and other more application-specific ones), as explained in [11].

3.3 Test Block Libraries

A library of test blocks arose this way at Siemens in a very short time, covering test blocks that represent and implement, e.g. (cf. again Figs. 2 and 3):

Common actions: Initialization of test tools, system components, test cases and general reporting functions,

Switch-specific actions: Initialization of switches with different extensions,

Call-related actions: Initiation and pick up of calls via a PBX-network or a local switch,

CTI application-related actions: Miscellaneous actions to operate a CTI application via its graphical user interface, e.g., log-on/log-off of an agent, establish a conference party, initiate a call via a GUI, or check labels of GUI-elements.

3.4 Generating Test Blocks

This simple and well structured component model enables the automatic generation of coordinable components. In this application domain, only a few components are generated out of directly programmed code (in some script language for some proprietary tools or APIs, e.g. for the communication with the Hipermon). Most components are directly obtained from behaviour recordings during experiments (e.g. the body of the communication "answers" from the Rational Robot). The general structure of most of the test blocks is in fact similar: a parameterized test script written in some typically proprietary language must be transferred to/from the subsystem or its test tool. We provide tools for the automatic generation of tool-specific adapter code that

makes legal test blocks out of such test scripts. Thus the definition of a new test block is rather simple: the tester in the field records a GUI test script which performs some actions, the test engineer defines the abstract component as described above, and the script code is automatically wrapped into a test block that can be directly made available to the test engineers, who graphically construct new test cases.

4 Organization of Coordination in the ITE

All we need to define coordination in the ITE is already provided by the test case model together with the executable code of each test block. The graph structure that we use for the description of test cases also defines a superposition of coordination sequences over the executable code, and it is thus independent of the chosen communication paradigms and of the underlying programming language.

4.1 The Framework

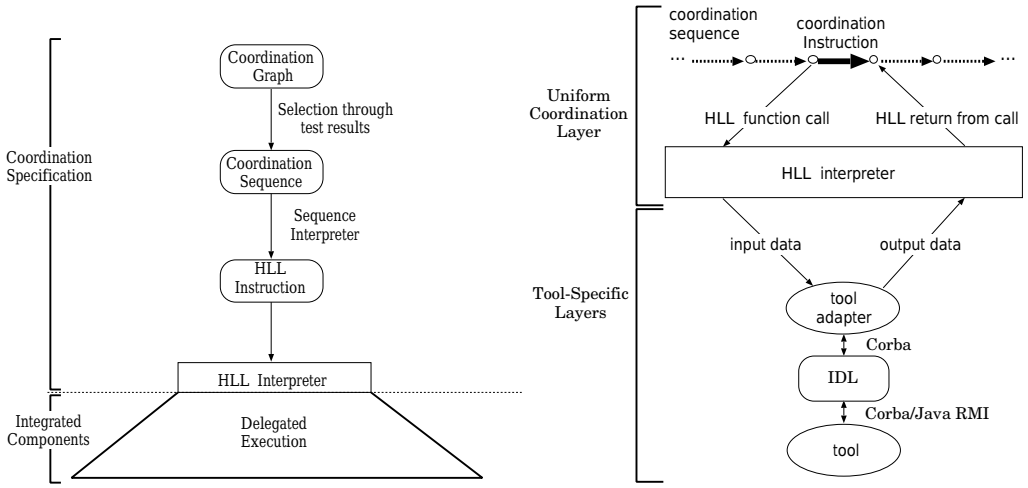
The coordination environment of the ITE bases on the paradigm of application development in the underlying Agent Building Center tool (ABC), which is *coordination-oriented*. In the ABC, application development consists in fact of the behaviour-oriented combination of building blocks on a *coarse* granular level. Building blocks are identified on a functional basis, understandable to application experts, and usually encompass a number of 'classical' programming units (be they procedures, classes, modules, or functions). They are organized in application-specific collections. In contrast to (other) component-based approaches, e.g., for object-oriented program development, the ABC focusses on the dynamic behaviour: (complex) functionalities are graphically stuck together to yield flow graph-like structures embodying the application behaviour in terms of control.

Throughout the behaviour-oriented development process, the ABC offers access to mechanisms for the verification of libraries of constraints by means of model checking (Sect. 5). The model checker individually checks hundreds of typically very small and application- and purpose-specific constraints over the flow graph structure. This allows concise and comprehensible diagnostic information in the case of a constraint violation (see Fig. 5), since the feedback is provided on the coordination graph, i.e. at the application level rather than on the code.

These characteristics are the key towards distributing labour according to the various levels of expertise.

Programming Experts: They are responsible for the software infrastructure, the runtime-environment for the compiled services, as well as for programming the single building blocks.

Domain Modelling Experts: They classify the building blocks, typi-



(A) (B)

Fig. 4. The Coordination Environment in the ITE Scenario

cally according to technical criteria like their version or specific hardware or software requirements, their origin (where they were developed) and, here, most importantly, according to their intent for a given application area. The resulting classification scheme is the basis for the constraint definition in terms of modal formulas.

Application Experts: They develop concrete applications just by defining their coordination structure. This happens without programming: they graphically combine building blocks into coarse-granular flow graphs. These coordination graphs can be immediately executed by means of an interpreter, in order to validate the intended behaviour (rapid prototyping). Model checking guarantees the consistency of the constructed graph with respect to the constraint library.

4.2 The Testing Scenario

In the ITE the test cases play the role that applications play in the ABC and we are able to use all the benefit offered by the development environment: in particular, the design environment provides the capability of designing hierarchical test cases, and the interpreter provides an efficient mechanism for test case execution.

4.3 Test Case Execution

The general principle of the test case execution in the ITE is shown in Fig. 4(A). From the coordination point of view, a test case model is interpreted as a coordination graph, where the actually executed coordination sequence (a path in the graph) is determined at runtime by results of the execution of the actual test block. In the concrete test case of Fig. 2, the branching is determined

at the coordination level through the results of the check points (i.e. *checkAgentLabel*, *upnCheckDisplay*, and *checkWindow*). The coordination sequence is executed by means of a sequence interpreter (*tracer* tool): for each test block it delegates the execution to the corresponding execution code. This reflects our policy of separation between coordination and computation: it embodies the superposition of the coordination on the components'code and it enables a *uniform view* on the tool functionalities, abstracting from any specific technical details like concrete data formats or invocation modalities.

Intertool communication is realized via parameter passing and tool functionality invocation by function calls which, via their arguments, pass abstract data to the adapters encapsulating the underlying functionalities as sketched in Fig. 4(B). The functionalities can be accessed via the *Corba* or *Java RMI* mechanism. In the concrete setting of system level tests the input data for the test tools are test scripts, that can be passed to the subsystems by the corresponding test tools (delegation stack principle).

The practical impact of our coordination based test environment exceeded our expectations: ITE has been already successfully used in industrial system-level testing of advanced CTI applications based on Siemens' HICOM family of switches [11,5]. In such scenarios, we have been able to fully automate the test case execution, and to document an efficiency improvement of factors over 30 during the test execution phase [12].

5 Model Checking as an Aid to Test Case Design

In [13] we presented some pragmatic verification-oriented aspects of our solution: we showed how the component-based test design was introduced on top of a library of elementary (but intuitively understandable) test case fragments (the test blocks), and we showed that the correctness and consistency of the test design is fully automatically enforced in ITE via *model checking*. The impact of this approach on the efficiency of test case design and documentation is dramatic in industrial application scenarios.

The ITE contains an iterative model checker based on the techniques of [16]: it is optimized for dealing with large numbers of constraints, in order to allow verification in real time. Concretely, the algorithm verifies whether a given model satisfies properties expressed in a user friendly, natural language-like macro language. In the CTI setting:

- the *properties* express correctness or consistency constraints the target CTI service or the test case itself are required to respect.
- the *models* are directly the coordination graphs, where building block names correspond to atomic propositions, and branching conditions correspond to action names. Figs 2 and 5 show typical test graphs for illustration.

Classes of constraints are formed according to the application domain, to

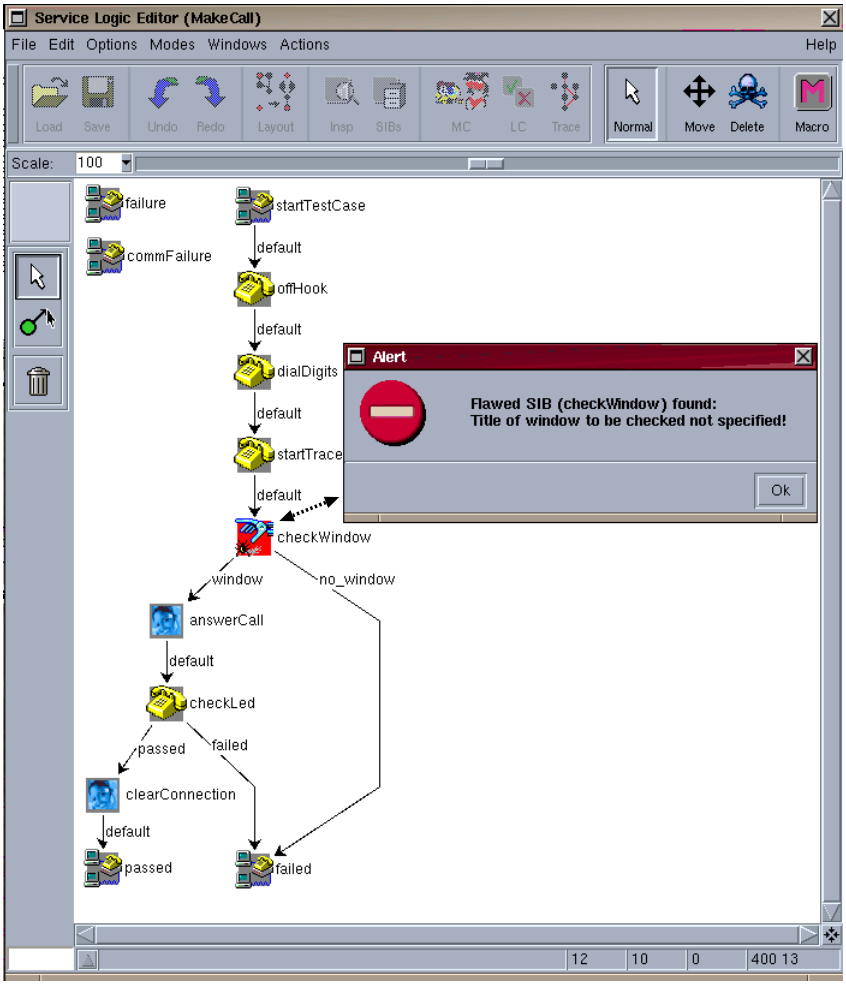


Fig. 5. Test Case Checking in the ITE Environment

the subsystems, and to the purposes they serve. This way it depends on the global test purpose, which constraints are bound to a test case. [3]

5.1 The Logic

Local Constraints.

The overall on-line verification during the design of a new test case captures both local and global constraints. Local constraints specify requirements on single SIBs, as well as their admissible later parameterization.

Whereas the specification of single SIBs is done simply by means of a predicate logic over the predicates expressed in the taxonomy, parametrization conditions are formulated in terms of a library of corresponding predicates. The verification of local constraints is invoked during the verification of the

[3] It was not possible to obtain clearance for publication of confidential material pertaining to the actual implementation of portions of the system, including complex test cases and specific constraints.

global constraints.

Global Constraints: The Temporal Aspect.

Global constraints allow users to specify causality, eventuality and other vital relationships between SIBs, which are necessary in order to guarantee test case well-formedness, executability and other frame conditions.

A test case property is global if it does not only involve the immediate neighbourhood of a SIB in the test case model [4], but also relations between SIBs which may be arbitrarily distant and separated by arbitrarily heterogeneous submodels. The treatment of global properties is required in order to capture the essence of the expertise of designers about do's and don'ts of test case design, e.g. which SIBs are incompatible, or which can or cannot occur before/after some other SIBs. Such properties are rarely straightforward, sometimes they are documented as exceptions in thick user manuals, but more often they are not documented at all, and have been discovered at a hard price as bugs of previously developed test cases. This kind of domain-specific knowledge accumulated by experts over the years is particularly worthwhile to include in the design environment for automatic reuse.

In the ITE, such properties are gathered in a Constraint Library, which can be easily updated and which is automatically accessed by the model checker during the verification.

Global constraints are expressed internally in the *modal mu-calculus* [9]. The following negation-free syntax defines mu-calculus formulas in positive normal form. They are as expressive as the full modal mu-calculus but allow a simpler technical development.

$$\Phi ::= A \mid X \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid [a]\Phi \mid \langle a\rangle\Phi \mid \nu X.\,\Phi \mid \mu X.\,\Phi$$

In the above, $a \in \mathit{Act}$, and $X \in \mathit{Var}$, where A is given by the SIB taxonomy, Act by the library of branching conditions, and Var is a set of variables. The fixpoint operators νX and μX bind the occurrences of X in the formula behind the dot in the usual sense. Properties are specified by *closed* formulas, that is formulas that do not contain any free variable.

Formulas are interpreted with respect to a fixed labeled transition system $\langle \mathcal{S}, \mathit{Act}, \rightarrow\rangle$, and an environment $e : \mathit{Var} \rightarrow 2^{\mathcal{S}}$. Formally, the semantics of the mu-calculus is given by:

$$\begin{aligned}
[\![X]\!]e &= e(X)\\
[\![\Phi_1 \vee \Phi_2]\!]e &= [\![\Phi_1]\!]e \cup [\![\Phi_2]\!]e\\
[\![\Phi_1 \wedge \Phi_2]\!]e &= [\![\Phi_1]\!]e \cap [\![\Phi_2]\!]e\\
[\![[a]\Phi]\!]e &= \{\, s \mid \forall s'.\ s \xrightarrow{a} s' \Rightarrow s' \in [\![\Phi]\!]e \,\}
\end{aligned}$$

[4] I.e., the set of all the predecessors/successors of a SIB along all paths in the model.

$$[\![\langle a\rangle\Phi]\!]e = \{\, s \mid \exists s'.\ s \xrightarrow{a} s' \wedge s' \in [\![\Phi]\!]e \,\}$$
$$[\![\nu X.\Phi]\!]e = \bigcup\{S' \subseteq \mathcal{S} \mid S' \subseteq [\![\Phi]\!]e[X \mapsto S']\}$$
$$[\![\mu X.\Phi]\!]e = \bigcap\{S' \subseteq \mathcal{S} \mid S' \supseteq [\![\Phi]\!]e[X \mapsto S']\}$$

Intuitively, the semantic function maps a formula to the set of states for which the formula is "true". Accordingly, a state s satisfies $A \in \mathcal{A}$ if s is in the valuation of A, while s satisfies X if s is an element of the set bound to X in e. The propositional constructs are interpreted in the usual fashion: s satisfies $\Phi_1 \vee \Phi_2$ if it satisfies one of the Φ_i and $\Phi_1 \wedge \Phi_2$ if it satisfies both of them. The constructs $\langle a\rangle$ and $[a]$ are *modal operators*; s satisfies $\langle a\rangle\Phi$ if it has an a-derivative satisfying Φ, while s satisfies $[a]\Phi$ if each of its a-derivatives satisfies Φ. Note that the semantics of $\nu X.\ \Phi$ (and dually of $\mu X.\Phi$) is based on Tarski's fixpoint theorem [Tars55]: its meaning is defined as the greatest (dually, least) fixpoint of a continuous function over the powerset of the set of states.

For the project we provide a simple 'sugared' version of LTL, which is translated into the modal mu-calculus for model checking. Here it is important to provide a natural language-like feeling for the temporal operators. As indicated by the example below, the standard logical connectors turned out to be unproblematic. We omit the formal definition of **next**, **generally**, **eventually**, and **until** here, as they are standard. In addition, we have implemented a pattern-driven formula editor which further simplifies the extension of the constraint library.

5.2 Expressing Test Case Properties

The library of constraints is also structured according to the main purposes addressed by the constraints.

Legal Test Cases: Constraints in this class define the characteristics of a correct test case, independently of any particular system under test and test purpose. Specifically, testing implies an evaluation of the runs wrt. expected observations done through *verdicts*, represented through the predicates passed and failed. For example, to enable an automated evaluation of results, verdict points should be disposed in a *nonambiguous* and *noncontradictory way* along each path, which is expressed (in a more user-friendly syntax) as follows:

$$(\mathsf{passed} \ \vee \mathsf{failed}) \Rightarrow \ \mathbf{next}(\mathbf{generally}\ \neg(\mathsf{passed} \ \vee \mathsf{failed}\ ))$$

POTS Test Cases: these constraints define the characteristics of correct functioning of Plain Old Telephone Services (POTS), which build the basis of any CTI application behaviour. Specific constraints of this class concern the different signalling and communication channels of a modern phone with an end user: signalling via tones, messaging via display, optic signalling via LEDs, vibration alarm. They must e.g. all convey correct and consistent information.

System Under Test -Specific Test Cases: these constraints define the

correct initialization and functioning of the single units of the system under test (e.g. single CTI applications, or the switch), of the corresponding test tools, and of their interplay. Fig. 5 shows the detection of a mismatch for this class of correctness criteria. Here, a misconfiguration in the test case definition is discovered: the identifier (the "title") of the window expected to appear on the Call Center Agents PC after a call has been started in the switch should have been specified before it can be accessed by Rational Robot. In fact, the Robot (the GUI test tool for the PC Call Center Application) needs this information in order to check that this window appropriately appears on the screen. As we see in this example, also the diagnostic information in case of property violation is provided in a user-friendly way: the violating path is the one that leads to the highlighted (red, instead of gold) Rational Robot test block, and a verbal formulation of the failed property allows the test designer to spot the problem without need to master temporal logics.

6 Related Work

Our work differs both from the usual approach to test definition and generation and from the usual attitude in the coordination community.

Most research on *test automation for telecommunication systems* concentrates on the generation of test cases and test suites on the basis of a formal model of the system: academic tools, like TORX [21], TGV [3,7], Autolink [15], and commercial ones like Telelogic Tau [20] presuppose the existence of fine-granular system models in terms of either automata or SDL descriptions, and aim at supporting the generation of corresponding test cases and test suites. This approach was previously attempted in the scenario we are considering here, but failed to enter practice because it did not fit the current test design practice, in particular because there did not exist any fine granular formal model of the involved systems.

The requirements discussed in this paper exceed the capabilities of todays commercial testing tools. To our knowledge there exist neither commercial nor academic tools providing comprehensive support for the whole system-level test process. We combine commercial test tools (in this case Rational Robot [14], Hipermon [6]) that deal with the subsystems of the considered scenario in order to capture the global test process.

Concerning *coordination approaches*, the closest to ours is in our opinion that of [1], which proposes the use of coordination contracts to promote the separation of the coordination aspects that regulate the way objects interact in a system, from the way objects behave internally. Like for us, their main concern is supporting evolutionary aspects of the whole system. In their work, contracts fulfill a role similar to architectural connectors: they make these coordination features available as first-class citizens, so that it is possible to treat them distinctly from the functionality of the components. Contracts

are based on superposition mechanisms [8] for supporting forms of dynamic reconfiguration of systems. These mechanisms enable contracts to be added or replaced without the need to change the objects to which they apply. [4] describes CDE, an environment for developing coordination contracts in Java. The CDE approach is still programming-oriented: unlike our coordination graphs, contracts must be programmed, they do not (yet) support macros or hierarchy, and no automatic verification for contracts is available. In our application domain, scalability of the approach is a major demand! It must be applicable to a regression testing scenario of hundreds of complex applications with very high regression frequencies. Accordingly, it is important that contracts (for us, test cases/coordination graphs are the global contracts) be

- definable in a programming-free fashion,
- themselves largely reusable, since we coordinate large behaviours,
- hierarchical, and
- subject to the validation of "reusability" of contracts via model checking in different contexts.

Indeed, hierarchy and formal verification of test cases are appreciated and heavily used features of the ITE environment.

7 Conclusion

Coordination graphs provide a useful abstraction mechanism to be used in conceptual modelling because they direct developers to the identification and promotion of interactions as first-class citizens, a pre-condition for taming the complexity of system construction and evolution. Their incarnation for system-level test case application has proven the feasibility of the approach and, more importantly, its adequacy for adoption in an industrial environment. Coarse grained modelling was natural for the test engineers, who are used to a functionality-oriented macro-model of the systems, of the test tools, and of the applications they test. It was also gracefully fitting with their pre-existing practice. Lightweight coordination solved the problem of keeping track of continuously evolving subsystems. The graphical configuration of test cases was perceived as intuitive, easy to manage, and for the first time providing an illustrative means to depict system-wide behaviours. The additional benefit of verification of the test cases wrt. abstract requirements or conventions was also perceived as useful and economically productive: it greatly enhanced reusal of test cases in similar contexts, and it allowed fast debugging of new test cases to take care of changed contexts or settings.

Our partners are confident that the scalability of the approach to the next generation of switches (which will involve widely networked and mobile applications) is within reach.

Acknowledgements

This is joint work with G. Brune, H.-D. Ide, W. Goerigk, B. Hammelmann, and A. Erochok (SIEMENS ICN Witten), A. Hagerer, O. Niese and M. Nagelmann (METAFrame Technologies GmbH Dortmund).

References

[1] L. Andrade, J. Fiadeiro, J. Gouveia, G. Koutsoukos, A. Lopes, M. Wermelinger: *Coordination Technologies For Component-Based Systems*, to appear at Integrated Design and Process Technology, IDPT-2002, Pasadena (CA), June, 2002, Society for Design and Process Science.

[2] V. Braun, T. Margaria, B. Steffen: *The Electronic Tool Integration Platform* appears in the Special Theme Issue on "Internet Based Technology Transfer Services" of the Journal Asia Pacific Tech Monitor.

[3] J.-C. Fernandez, C. Jard, T. Jéron, C. Viho: *An Experiment in Automatic Generation of Test Suites for Protocols with Verification Technology*, Science of Computer Programming, 29, 1997.

[4] J. Gouveia, G. Koutsoukos, L. Andrade, J. Fiadeiro: *Tool Support for Coordination Based Evolution*, Proc. TOOLS 38, W.Pree (ed.), IEEE Computer Society Press 2001, pp. 184-196.

[5] A. Hagerer, T. Margaria, O. Niese, B. Steffen, G. Brune, H.-D. Ide: *An Efficient Regression Testing of CTI Systems: Testing a complex Call-Center Solution*, Accepted for publication in Annual Review of Communic., Vol. 55, Int. Engineering Consortium, Chicago, 2001.

[6] Herakom GmbH, Germany, `http://www.herakom.de`.

[7] C. Jard, T. Jeron: *TGV: Theory, Principles and Algorithms*,Proc. Int.Symposium on Integrated Design and Process Technology 2002, Pasadena, June 2002, (to appear).

[8] S. Katz: *A Superimposition Control Construct for Distributed Systems*, ACM TOPLAS 15(2), 1993, pp. 337-356.

[9] D. Kozen: *Results on the Propositional μ-Calculus*, Theoretical Computer Science, Vol. 27, 1983, pp. 333-354.

[10] Object Management Group: *The Common Object Request Broker: Architecture and Specification*, Revision 2.3, Object Management Group, 1999.

[11] O. Niese, T. Margaria, A. Hagerer, M. Nagelmann, B. Steffen, G. Brune, H.-D. Ide: *An Automated Testing Environment for CTI Systems Using Concepts for Specification and Verification of Workflows*, In Annual Review of Communic., Vol. 54, Int. Engineering Consortium, Chicago, 2000.

[12] O. Niese, T. Margaria, A. Hagerer, B. Steffen, G. Brune, W. Goerigk, H.-D. Ide: *An Automated Regression Testing of CTI Systems*, In Proc. IEEE European Test Workshop 2001, pp. 51-57, Stockholm (S), 2001.

[13] O. Niese, B. Steffen, T. Margaria, A. Hagerer, G. Brune, H.-D. Ide: *Library-based Design and Consistency Checking of System-level Industrial Test Cases*, FASE 2001, Int. Conf. on Fundamental Aspects of Software Engineering, Genova, LNCS 2029, Springer Verlag, 2001, pp. 233-248.

[14] Rational, Inc.: *The Rational Suite description*, `http://www.rational.com/products`.

[15] M. Schmitt, B. Koch, J. Grabowski, D. Hogrefe: *Autolink - A Tool for Automatic and Semi-automatic Test Generation from SDL-Specifications*, Technical Report A-98-05, Medical Univ. of Lübeck, Germany, 1998.

[16] B. Steffen, A. Claßen, M. Klein, J. Knoop, T. Margaria: *The Fixpoint Analysis Machine*, (invited paper) CONCUR'95, Pittsburgh (USA), August 1995, LNCS 962, Springer Verlag.

[17] B. Steffen, T. Margaria, V. Braun: *The Electronic Tool Integration platform: concepts and design*, Int. J. STTT (1997)1, Springer Verlag, 1997, pp. 9-30.

[18] B. Steffen, T. Margaria: *METAFrame in Practice: Intelligent Network Service Design*, In *Correct System Design – Issues, Methods and Perspectives*, LNCS 1710, Springer Verlag, 1999, pp. 390-415.

[19] Sun: *Java Remote Method Invocation*. `http://java.sun.com/products/jdk/rmi`.

[Tars55] A. Tarski: *A Lattice-Theoretical Fixpoint Theorem and its Applications*, Pacific Journal of Mathematics, V. 5, 1955.

[20] Telelogic: *Telelogic Tau*, `http://www.telelogic.com`.

[21] J. Tretmans, A. Belinfante: *Automatic testing with formal methods*, In EuroSTAR'99: 7^{th} European Int. Conference on Software Testing, Analysis & Review - EuroStar Conferences, Galway, Ireland, November 8-12, 1999.