Test Automation of Safety-Critical Reactive Systems

J. Peleska* M. Siegel†

*Department of Mathematics and Computer Science FB3,University of Bremen, 28334 Bremen, Germany, email: jp@Informatik.Uni-Bremen.de†Department of Computer Science, University of Kiel, Preußerstr. 1 – 9, 24105 Kiel, Germany, email: mis@informatik.uni-kiel.de

Abstract

This article focuses on test automation for safety-critical reactive systems. In the first part of the paper we introduce a methodology for specification, design and verification of fault-tolerant systems allowing to combine different methods in a systematic and consistent way, provided that these methods are compositional. The methodology indicates how to "switch" between formal verification and testing during the construction of (possibly large) reactive systems. We introduce the basic notions of testing as far as relevant in the context of reactive systems and relate them to the verification methodology. Part II formally describes our test automation method which is based on Hoare's CSP and takes Hennessy's testing theory as a starting point. It is indicated how this specific method fits into the general approach described in Part I. We introduce CSP test drivers which are trustworthy in the sense that they "approximate" refinement proofs, converging to a full proof with the increasing (possibly infinite) number of tests successfully executed. These drivers have been implemented in the VVT-RT (Verification, Validation and Test for Reactive Real-Time Systems) *tool developed at Bremen University in cooperation with the University of Kiel, JP Software-Consulting and ELPRO LET GmbH. The presentation of this article is based on the lectures given by the first author during the WOFACS '96 workshop at the University of Capetown.*

Keywords: *Test generation, test strategies, dependability, fault-tolerance, reactive systems*

Computing Review Categories: *????*

Received: August 1996, Accepted: ???, Final version: ???.

1 Introduction

Design, execution and evaluation of tests for safety-critical systems require considerable effort and skill and consume a large part of today's development costs. Due to the growing complexity of control systems it has to be expected that their trustworthy test will become unmanageable in the future if only conventional techniques requiring a high degree of human interaction during the test process are applied. For these reasons methods and tools helping to automate the test process gather wide interest both in industry and research communities. *"Serious"* testing – not just playing around with the system in an unsystematic way – always has to be based on some kind of specification describing the desired system behaviour at least for the situations covered by the test cases under consideration. As a consequence, the problem of test automation is connected to formal methods in a natural way, because the computer-based design and evaluation of tests is only possible on the basis of formal specifications with well-defined semantics.

Just as it is impossible to build theorem provers for the fully mechanised proof of arbitrary assertions, the general problem of testing against arbitrary types of specifications cannot be solved in a fully automated way. The situation is much more encouraging, however, if we specialise on well-defined restricted classes of systems and test objectives. This strategy is pursued in the present article, where we will focus on the test of *reactive systems*.

Reactive Systems are characterised by their continuous interaction with the environment and often rely on specific environment behaviour as a premise for their correct operation. As a consequence, the environment behaviour has to be reflected in test configurations for reactive systems. Moreover, tests do not only provide a single set of input data at the beginning of the execution, but continuously interact with the target system by monitoring system outputs and providing new inputs at certain stages of the execution. For *safety-critical* reactive systems another dimension of complexity is introduced by the requirement that the system should guarantee at least a minimum amount of dependability even in presence of certain internal or external faults. This requires *fault-tolerance*, where normal operation and operation in presence of exceptions have to be distinguished, so that the specification, design and verification efforts have to deal with two types of system behaviours instead of one.

In order to cope with such a high degree of complexity we suggest in Part I of the paper to develop and verify safety-critical systems according to a well-defined methodology, regardless of the specific methods applied during the development life cycle. Development and verification according to this methodology is driven by the architectural decomposition of the system. Each development step refining the architecture designed so far leads to a specific verification obligation, so that the architectural design can be used for track-keeping of the verification process. In order to structure developments in a second dimension it is suggested to separate the hierarchy of requirements specifications constructed during the architectural decomposition into the specifications related to normal behaviour and those related to acceptable behaviour in presence of faults. These structuring mechanisms for the development and verification process offer the opportunity to change formalisms at precisely defined development steps, so that different techniques optimised for specific sub-tasks of the whole development and verification process can be applied. For example, the methodology indicates how to combine formal verification of certain system components with the test of others in order to increase the efficiency of the overall verification process. In Part II of the paper we prove that such a combination is really consistent in the case of CSP-based tests and formal verification of refinement properties. Based on Hennessy's testing theory, we specify test classes characterising various CSP refinement notions with a minimum amount of test effort. Moreover, CSP specifications for test drivers are given that indicate how to perform automatic test generation, execution and evaluation of these test classes.

1.1 The VVT-RT Tool

The test classes and associated drivers described in Part II are implementable and have been integrated in the VVT-RT tool *(Validation, Verification and Test for Reactive Real-Time Systems)* offering the following possibilities:

- symbolic execution of CSP specifications
- formal validation and verification of the specification
- automated generation of test cases based on the CSP specification
- automated test execution
- automated test evaluation, including the check of real-time properties
- automated test documentation

VVT-RT has been developed by the first author in cooperation with ELPRO LET GmbH. The main fields of application are *hardware in the loop tests* for reactive (embedded) systems based on PLC, VME or PC hardware. Typically, such applications possess discrete interfaces and focus on possibly complex control functionality, while data transformations play a less important rôle. Examples are railway control systems, control components ensuring fault-tolerance, telephone switching systems and network protocols. At present the VVT-RT system is used for the test of computers controlling components of railway interlocking systems. The first application was the automated test of a PLC system controlling signals, traffic lights and train detection sensors for a tramway crossing. VVT-RT makes use of the model checker FDR developed by Formal Systems Ltd [Ltd94].

1.2 Related Work

The concept for our methodology described in Part I was stimulated by Schepers' development of a formal specification language and associated proof theory explicitly distinguishing between normal and exceptional behaviour of fault-tolerant systems [Sch94]. In contrast to that our main objective is to work out a general paradigm admitting the use of *arbitrary* specification languages, as long as they are compositional. Combining different methods and associated tools for the development and verification of complex and large systems is the objective of the R&D project UniForM (Universal Formal Methods Workbench) funded by the German ministry of research and education (BMBF) and performed by Bremen University in collaboration with the University of Oldenburg and ELPRO LET GmbH, Berlin [KPO96].

Testing and formal verification were originally investigated in different communities (see [Mye79] for the conventional testing approach). Today, however, the link between these fields seems to be quite soundly established, and this has been mainly stimulated by the necessity to automate larger portions of the test process: Formal specification languages are mandatory for automatic test generation and evaluation.

Test automation concepts based on formal methods have been developed for most of the important specification paradigms: Gaudel [Gau95] investigates testing against algebraic specifications, Hörcher and Mikk in collaboration with the author [HP94, Hör95, Hör95, Mik95] focus on the automatic test evaluation against Z specifications and Müllerburg [Mül94] describes test automation in the field of synchronous languages. The idea to apply the theoretical results about testing in process algebras to practical problems was first presented by Brinksma, with the objective to automate testing against LOTOS specifications. His concept has been applied for the automation of OSI conformance tests; see [Bri89] for an overview. Several authors already have addressed the question how to test reactive systems in an efficient way (see, e. g. [ROM89, RAO92, WGS94]), but still this field seems to be less thoroughly explored than the test of non real-time or sequential systems. An introduction of the basic problems and notions related to real-time testing is given in [Sch93].

The CSP-based test automation method described in Part II is an improved extension of the results presented in [PS96a]. It focuses on untimed CSP; results about real-time testing with CSP are documented in [1]. The practical application of our method in an industrial project has been described in [Pel96].

1.3 Overview

This article is structured as follows: Part I focuses on the general methodology for specification, design and verification of fault-tolerant systems. After introducing the basic concepts of dependability in Section 2, the methodology is introduced in Section 3. The fundamentals of testing reactive systems are introduced in Section 4. Part II describes the theoretical results on which our test automation method is based and indicates how these results are implemented in the VVT-RT tool. In Section 5 we introduce CSP notations and conventions used in subsequent sections. Section 6 introduces transition graphs and results from Hennessy's testing theory. Section 7 contains the main results, where we investigate implementable, minimal test classes and trustworthy test drivers. Appendix A contains the proofs for the main results presented in Part II. Section 8 gives the conclusions.

Part I
Testing Methodology

2 Safety-Critical Systems and Dependability

2.1 Terminology

Test methods designed for safety-critical systems often refer to the terminology defined in the context of *dependability*. In this section, we will therefore introduce some of the most important notions.

Systems are called *safety-critical* if their malfunction represents a severe threat to human lives or to the environment. Following Laprie's terminology [L$^+$92], *dependability is the capability of a system to deliver the specified application services during its period of operation.* Laprie identified four attributes which characterise the dependability of a system: (1) A *safe* system cannot assume states that are regarded as "catastrophic" from the point of view of the application. This means that the system will only perform transitions into states satisfying the specified invariants, perform calculations that are correct with respect to the specification and output data fulfilling the desired integrity constraints. Safety does not guarantee that a desired calculation and the corresponding output will always be produced. This aspect is covered by the following two attributes: (2) *Reliability* is a characteristic specifying the probability that a system will deliver its service for a given period of time. (3) *Availability* is a measure reflecting the probability that the system will be available at a certain point in time. (4) Finally, *Security* reflects the capability of the system to protect the application against damage arising from accidental or malicious human interaction.

All dependability attributes refer to the specified application, and this is the very premise for the notion of *fault-tolerance*, the property of a system to preserve a (possibly limited) degree of dependability in presence of "undesired events" like malfunction of hardware or software, security attacks etc.: A dependable system may be subject to various *internal* defects, as long as these problems do not affect the application behaviour. In order to prevent undesired events from leading to catastrophic system behaviour, system designers must anticipate the possibility of their occurrence and incorporate protective mechanisms in their development concepts. The potential occurrence of any undesired event will be called a *threat* or a *fault hypothesis*, its actual occurrence a *fault*. If an exception causes a transition into an undesired system state, this state is called an *error*. If due to the occurrence of exceptions and errors a specified service to be performed by a (sub-)system cannot execute as required, this is called a *failure*.

2.2 Classification of System Behaviour

The concept of fault-tolerance leads to the classification of system behaviours depicted in Figure 1: In absence of any faults, the system shows *normal behaviour*. In presence of faults the system may perform an execution possibly differing from that of the undisturbed system. Now there are two possibilities: In the first case, an undesired event impairs the functionality of the system in a way that can be tolerated or might even be unobservable in the application layer. In this case the system shows *exceptional behaviour*. If the system performs an execution showing normal and/or exceptional behaviour, this is summarised as *acceptable behaviour*. In the second case, the occurrence of the undesired event corrupts the application functionality in a way that cannot be tolerated. This is called *catastrophic behaviour*.

3 A Specification and Verification Methodology for Dependable Systems

3.1 Motivation

As we shall see later on in Part II, testing against a requirements specification – if performed with an appropriate method and strategy – can be regarded as an "approximation" of a correctness proof verifying that the implementation is consistent with the requirements. Therefore testing and formal verification can be used alternatively, where a successful but not exhaustive test suite represents a "weaker form" of the formal proof. The underlying concept of such an alternative or combined use is that of *linking theories*: Different development and verification methods are combined to reach an overall goal (like showing that the implementation complies with the requirements) in a more efficient way than it would be possible with one isolated method. In order to guarantee consistency of the combination of methods it is necessary to link the underlying theories by showing how results achieved with one method (e. g. a refinement proof) can be expressed in another (e. g. a set of tests that should execute successfully if the refinement property holds). In this section we will therefore introduce a specification and verification methodology for dependable systems, which can be "instantiated" with any compositional formal method. Recall that a method is *compositional* if verification only depends on the requirements specifications and architectural decisions, but not on the *implementation* of the components involved, as long as they fulfill their associated requirements (see [Zwi89]). The main features of our methodology are:

- The verification process "is driven by" the architectural decomposition of the system, which corresponds to the associated proof tree structure in a natural way.
- The verification of dependability properties is divided into two separate suites investigating *normal behaviour* and *exceptional behaviour* correctness, and related to each other by the architectural decomposition of the system.
- The techniques to verify exceptional behaviour properties are exactly the same as those to be used for normal behaviour verification. In contrast to that, Schepers [Sch94] proposed a method using proof rules specifically designed to handle exceptional behaviour. The advantage of our approach is that the knowledge about "conventional" system development, verification and associated tools can also be applied in the context of dependability.

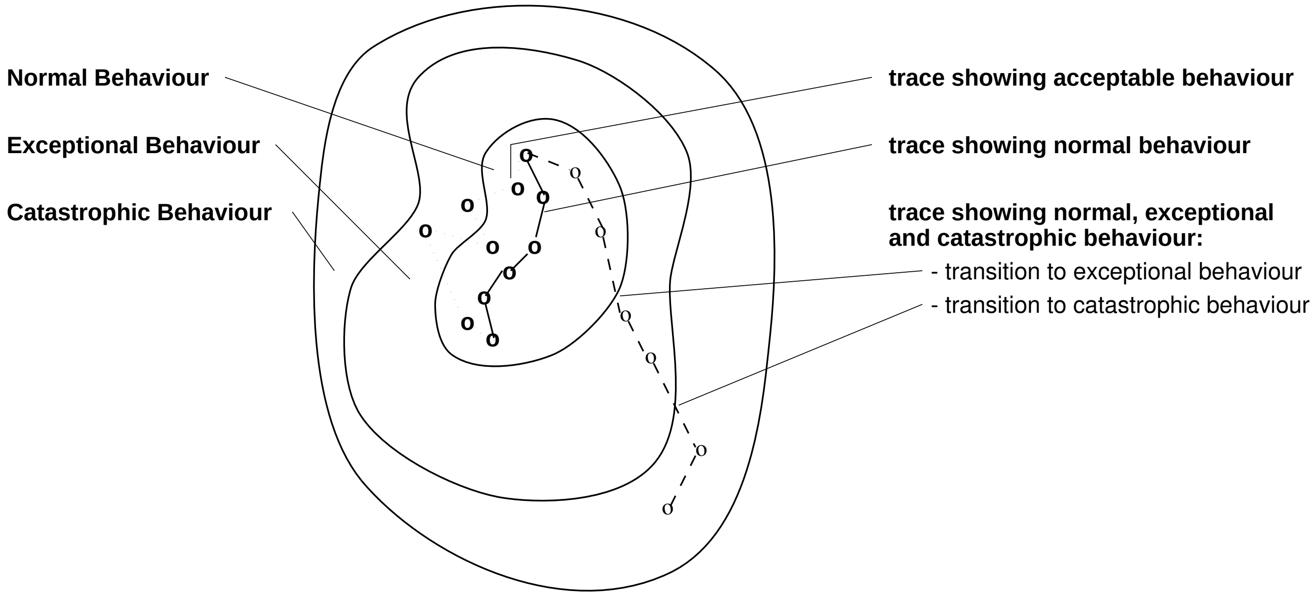

Figure 1. Classification of system behaviour

Before presenting the methodology in Section 3.4 and 3.5, we will first introduce basic concepts of software engineering about system models (Section 3.2) and system decomposition (Section 3.3).

3.2 System Models

The full description of a system can be structured into (a collection of) *conceptual models* and *architectural models*. The former describe *what* the system is supposed to do without necessarily saying how this could be implemented. The latter specify the implementable system structure (layers, processes, modules, processors, interfaces etc.). Architectural models specify *who*, that is, which implementable system component, is going to perform the services to be offered by the system. Documents describing conceptual models are called *requirement specifications*, architectural model descriptions are called *architecture design specifications*. Executable requirements specifications (components, algorithms and data structures) are called *explicit*, non-executable behavioural descriptions are called *implicit*. Observe that the notion of *detailed design* used in software-engineering and denoting the directly implementable algorithms and data structures represents an explicit requirements specification according to this classification. We also consider fragments of an executable programming language describing algorithms and data structures as explicit specifications for the required processor behaviour.

Conceptual models may be further classified according to *views* focusing on specific modelling aspects: The *Data View* specifies the data items to be stored and processed in the system. The *Functional View* specifies the functions operating on the data. (Description methods for object-oriented systems usually combine the data view and the functional view in one specification language according to the OO-paradigm.) The *Behavioural View* describes the causal properties (like sequencing of events), synchronisation and timing.

Conceptual models of large systems also need internal structure to cope with the complexity and the number of aspects to be captured. This internal structure is not necessarily related to the architectural model, since its objective is only to organise the collection of requirements. Some modelling languages explicitly distinguish between conceptual and architectural structures, as, for example, the *activity charts* and *module charts* provided by the Statemate languages [Ha90]. In other languages only a single set of operators introducing structure is provided (as for example in CSP or CCS), so that the distinction between conceptual and architectural structuring has to be drawn by means of additional comments outside the modelling language.

3.3 Hierarchies of Conceptual and Architectural Models

The classical software development approaches suggested to construct only one model of each kind, proceeding according to the phases *requirements analysis*, *architecture design*, *detailed design*, *implementation* and *integration*. Experience has shown that this approach is not suitable for the development of large systems, because there it is usually impossible to capture all the necessary requirements before proceeding with the design phases. The suggested feedback loops from design to requirements specification have turned out to be impractical, often resulting in severe inconsistencies between the conceptual and architectural models. These experiences have led to an approach constructing a *hierarchy of requirements specification and architecture design specifications*, where the hierarchy is defined by the architectural model and each set of documents is associated with a specific level of system decomposition. One representative of this approach is the *V-Model* [dI92] which is a "refinement" of the ISO 9000-3 standard, probably the most widely known software quality standard of today.

The V-Model suggests to organise architectural models in a *layered* fashion, so that each layer may be considered as an architectural model of its own. The layering should follow a top-down decomposition of the full system, where a suitable granularity of the layers is given by the following guidelines:

1. *System Level:* The whole target system without distinctions between software and hardware.
2. *Sub-System Level:* Large systems (e. g. a wide area network) may be decomposed into sub-systems (e. g. a local area network in the wide area network).
3. *Segment Level:* This is a further decomposition step which partitions (sub-)systems into well-defined components (e. g. one computer in a network). On this level the segments consisting of hardware only are separated from those combining hardware and software components.
4. *SW Configuration Item Level:* The software to be allocated in a segment is decomposed into different configuration items, each item representing a software sub-system (e. g. the processes associated with a specific layer of the OSI model which will be allocated in the segment).
5. *Component Level:* A part of a configuration unit, performing a well-defined service (e. g. one task of the configuration unit). Several component layers may be "inserted" to decompose a top-level component (e. g. tasks might be decomposed into threads).
6. *Module Level/Data Level:* This is the lowest level of decomposition defined by the *V-Model* (e. g. a sequential function or a database table with associated attribute definitions).

Now the development phases *alternate* between conceptual and architectural modelling steps: Starting with the complete system regarded as a black box, the *system requirements specification* contains all the details that could be captured in the first development step. Next, an architectural modelling step is performed by describing the decomposition of the system into sub-systems, the sub-system interfaces and their modes of interaction (e. g. parallel or sequential) in the *system architecture design specification*. The sub-system black boxes generated by the system architecture are now considered as "systems in their own right" and therefore associated with their own conceptual models (sub-system requirements specifications).

The collection of conceptual sub-system models will often contain more details than have been described on system level. According to the V-Model approach this does not force us to change the system requirements. (In many cases this would be very impractical, especially when the sub-systems are developed by other parties than the one having developed the system requirements and its architecture.) Instead, we have to discharge the *verification obligation* that

> the sub-system requirements *imply* the system requirements, provided that they cooperate according to the interfaces and modes of interaction specified in the system architecture.

This alternation between conceptual and architectural modelling steps is repeated until the decomposition comes to the module and data level, where the associated requirements specifications should be directly implementable in the target system.

3.4 The Decomposition Tree Paradigm

This decomposition concept described above is complicated in the case of safety-critical embedded systems by the following facts: First, we have to consider both the target system and its operational environment, since the target system behaviour often strongly depends on assertions to be met by the environment. Second, the conceptual models have to address both normal and exceptional behaviour, since the requirements in presence of faults will generally differ from those in absence of exceptions. Finally, for complex systems in general, it may become necessary to change the formalisms when moving from one level of decomposition to the next. These considerations have led to the following representation model we call the *decomposition tree paradigm* (a detailed introduction to the verification approach and its application to CSP is given in [Pel95a]).

The interaction between conceptual modelling of normal and exceptional behaviour, the step-wise decomposition in architectural models and the associated verification obligations can be visualised by the three trees depicted in Figure 2: The left tree depicts the hierarchy of normal behaviour requirements specifications together with their verification obligations. On the right-hand side we have an analogous tree for the acceptable behaviour, describing the "worst" deviations from normal behaviour that can still be tolerated. The tree in the center describes the architectural decomposition of the system. Each tree has two types of nodes, denoted by boxes and bubbles. In the architecture tree, the boxes contain names (*"black boxes"*) representing system components and their interface specifications, while the bubbles contain decomposition specifications (*"How do these black boxes interact?"*). In the requirements specification trees, the boxes contain specifications and the bubbles verification obligations, described by expressions

$$\frac{\mathsf{S(P1)} \quad \mathsf{S(P2)}}{S(P)} \quad [\ \mathsf{P = A(P1, P2)}\ ]$$

Here $\mathsf{S(Pi)}$ denote requirement specification and $\mathsf{P = A(P1, P2)}$ the architectural decomposition of P into $\mathsf{P1}$ and $\mathsf{P2}$. These expressions have to be instantiated with concrete notations and the proof obligation has to be discharged by means of the concrete proof theories when applying the methodology to a specific set of compositional languages. Examples will be given in Part II.

Each of the trees contains information about the target system and its environment in separate sub-trees. While for the target system everything has to be modelled up to a granularity allowing to implement the components described, the environment models are only as detailed as necessary for understanding and verifying their interaction with the target system. Therefore the leaves of the target system architecture sub-tree contain black boxes and

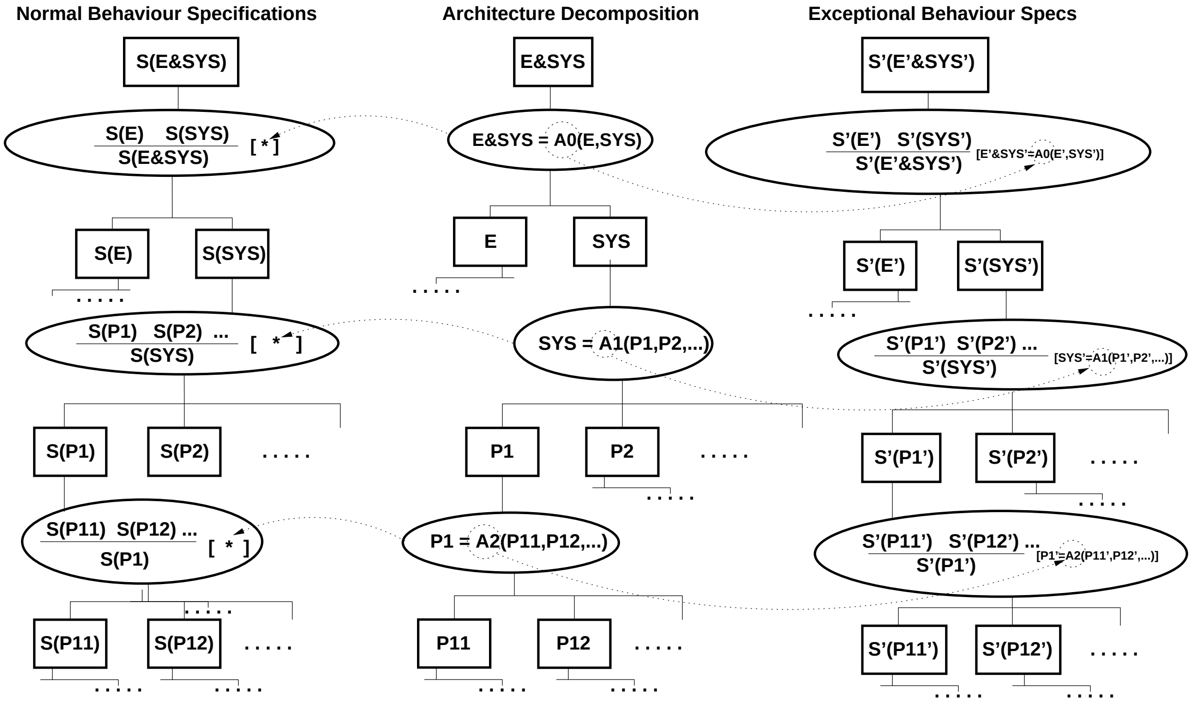

Figure 2. Architecture-driven specification and verification of dependable systems.

interface specifications for modules or data entities in the above terminology. They are associated with leaves of the requirements specification trees containing directly implementable or even executable specifications. The leaves of the environment sub-tree contain the coarsest decomposition level which completely specifies the relevant interactions with the target system.

3.5 Development and Verification Steps according to the Decomposition Tree Paradigm

We will now describe how the Decomposition Tree Paradigm can be applied in a systematic system development effort as a sequence of well-defined specification and verification steps.

Step 1: Specification of the system requirements (normal behaviour) The first step consists in the specification of the required system behaviour in absence of any external or internal exceptions. Since this behaviour generally relies on assumptions about the environment, the root node of the normal behaviour specification tree contains requirements about the target system SYS *when operating in environment* E, which is denoted by E&SYS. We are *not* interested in the behaviour of SYS in any other context.

Step 2: Specification of the system requirements (acceptable behaviour) In analogy to Step 1, the root node of the acceptable behaviour specification tree contains the specification S'(E'&SYS'), where E' denotes the environment showing acceptable behaviour and SYS' the target system in presence of internal faults which we will have to take into account later on because of the design decisions made. S'(E'&SYS') specifies *what can be tolerated as acceptable behaviour* in presence of the faulty environment E' and internal faults SYS'. The possible erroneous behaviours of E and SYS will be specified in further steps described below.

Step 3: Separation of environment and target system The first architectural design step separates the environment from the target system and specifies the system interface and their mode of interaction (for embedded systems this will be in most cases a variant of parallel composition). This is denoted by an architecture function A0(. , .) mapping several components (here environment and target system) onto another (here the full system E&SYS).

Step 4: Specification of the target system requirements (normal behaviour) This specification – denoted by S(SYS) – describes the behaviour of SYS in absence of any internal exceptions. It may also contain behaviours that won't occur in the operational environment.

Step 5: Specification of the environment requirements (normal behaviour) The environment specification S(E) contains all the relevant assertions about the behaviour of E in absence of exceptions, as far as relevant for the target system.

Step 6: Verification of the environment–target system separation (normal behaviour) After having specified the isolated behaviours of environment and target system under normal conditions, we have to verify that under these assumptions E&SYS will really satisfy the original requirements S(E&SYS). In any formalism the full verification obligation will be an "instantiation" of the following one:

Assume that

- The environment E operates according to its normal behaviour specification S(E).
- The target system SYS will operate according to its (postulated) normal behaviour specification S(SYS).

- The environment E and the target system SYS will cooperate according to the architectural decomposition E&SYS= A0(E,SYS). (This is denoted as a side condition since it involves the architecture function, whereas the two other conditions only involve requirements specifications.)

Then show that the full system E&SYS will satisfy the system requirements specification S(E&SYS) for normal behaviour.

If it is not possible to prove this obligation, the specification S(SYS) has to be strengthened or/and the architectural mapping A0 has to be changed. In general it will be not possible to influence the environment behaviour S(E).

Step 7: Specification of the target system requirements (acceptable behaviour) Due to internal design decisions anticipated for the target system it may be the case that internal acceptable behaviour has to be expected for SYS. The acceptable behaviour specification S'(SYS') specifies the *maximum deviation* of SYS from its normal behaviour.

Step 8: Specification of the environment requirements (acceptable behaviour) In analogy to S'(SYS'), the environment specification S'(E') contains the "worst case description" of the behaviour of E in presence of exceptions, as far as these exceptions have an impact for the target system.

Step 9: Verification of the environment–target system separation (acceptable behaviour) After having specified the "worst case" behaviours of environment and target system in the preceding two steps, we have to verify that under these assumptions E&SYS will really show "nothing worse than" its acceptable behaviour S'(E'&SYS'). The verification obligation for the acceptable behaviour verification is analogous to the one performed in Step 6, this time involving the acceptable behaviour specifications of the environment and the target system. The "link" to the normal behaviour verification is represented by the occurrence of the same architecture function A0(. , .) in the side condition of the proof obligation, this time taking the acceptable behaviour components as parameters. At this point the dependency between normal behaviour and acceptable behaviour verification, as expressed in the architecture function, may force us to revise our design and/or the requirements specifications: If the acceptable behaviour verification fails, because S'(E'&SYS') cannot be derived in architecture A0(. , .) from the specifications S'(E') and S'(SYS'), at least one of the following actions will become necessary:

1. If possible, strengthen the acceptable behaviour specification S'(SYS'). If this suffices to prove the verification obligation, the revision is restricted to the acceptable behaviour specification tree, without affecting normal behaviour and architectural decomposition.
2. If 1. does not work, the design step A0(. , .) has to be modified, and this will lead to a revision of both the normal and the acceptable behaviour specifications of SYS, as well as the normal behaviour verification performed in Step 6.

Further decomposition and verification steps After successful completion of Step 9, the analogous procedure is recursively applied to specify and verify the target system decomposition into sub-systems, the sub-system decompositions into segments and so on. The compositionality assumed for the formalism used guarantees that these steps will not necessitate a re-design of the higher decomposition levels, as long as the new decomposition step can guarantee the specifications of the next higher level. For the environment E a further decomposition will only be necessary if a closer analysis of its components is required for the analysis of normal and acceptable behaviour.

3.6 Linking Formalisms

Since the methodology described above is applicable in any compositional method, it is even possible to *change* the specification/design/verification formalism during the development of one system. This is an important consideration because

- when reaching the leaves of the decomposition tree, it will often be necessary to change from a specification language to a programming language,
- for different types of system components (e. g. data transformation components and components performing control tasks) it may be appropriate to use different specification and verification formalisms (e. g. Z and CSP),
- for components of different criticality it is desirable to apply methods of different strength without compromising the overall consistency of the verification approach.

In the architecture tree, the change of formalism is reflected by the architecture function admitting operators that differ from the previous decomposition steps. For example, the use of a parallel operator is allowed in A while working in a formalism describing distributed system components, whereas it might not be used on lower-level decompositions refining sequential components. A change of architecture formalisms is reflected in a bubble of the decomposition tree, possibly accompanied by the introduction of a new syntax for interface descriptions in the black boxes below this bubble.

A change of the specification/verification formalism affects the bubble containing the verification obligation, the higher-level specification box and the lower-level boxes resulting from the decomposition step: Suppose the formalism is changed in the decomposition step P1 = A2(P11,P12, ...): This means that S(P1) is written in specification language SL1, whereas the lower-level components are defined in the context of another language SL2. The verification obligation is an expression containing mixed assertions of SL1 and SL2. The theory link has to provide a mechanism how to discharge these mixed obligations and to prove the soundness of the mechanism.

The most important example for theory links in the context of the present paper will be given in Part II, Theorem 4: There it is shown that assertions about refinement relations can be replaced by assertions about successfully executed tests.

4 Testing: Terminology, Concepts and Strategies

4.1 Terminology

The efforts to ensure the quality and, specifically, the dependability of a system can be divided into *constructive* and *analytic measures*: The former aim at establishing dependability already in the development process (conceptual and architectural modelling and implementation), by means of specific development methods that are suitable to guarantee *a priori* correctness of the system. Analytic methods try to establish dependability by *a posteriori* examinations of completed development objects (specifications, code, system components etc.). In this paper we focus on analytic measures that may again be divided into *Validation, Verification* and *Test* methods. The following definitions have been taken from the international standard [DO93] for the development of software in airborne systems. Verbatim quotations from [DO93] are displayed as *emphasised* text.

Validation denotes the process of *determining that the requirements are the right requirements and that they are complete.* Since the system requirements specification is the "initial" document describing the system behaviour desired, no other document exists which could serve as a reference to *prove* that the specification is complete and really describes the customer's demands. Therefore validation can never be performed by means of formal proofs alone, but must also rely on informal techniques like simulation or – if the customer and the supplier will risk an *a posteriori* validation – system testing.

Verification denotes an evaluation of development objects with the objective to ensure their consistency with respect to the reference documents applicable for the products. So in contrast to validation, the verification process can rely on other documents specifying (at least in theory) completely how the product should look like. If the reference documents are written using a description language with a precisely defined semantics and verification is performed by means of mathematical reasoning, the term *formal verification* will be used. Observe that this definition of *verification* differs from that normally used in the formal methods community, where it is always supposed to be the *formal* verification process. In the context of this article, verification subsumes all formal and informal techniques applicable for *a posteriori* insurance of product system correctness, i. e., reviews, audits, walk throughs, test, formal verification etc.

In our context *testing* means execution of implemented system components providing specific data at their (input) interfaces, while monitoring the component behaviour. The objective of testing is to *verify that the component satisfies the specified requirements for the set of input data applied.* To avoid confusion we will use the term *simulation* for the *symbolic execution* of a specification with specific (abstract) input data, though this is nothing else than testing on an abstract level. Testing in our sense is often called *dynamic testing* to emphasise that the component under inspection is implemented and actually executed and to distinguish this technique from static analysis or simulation.

Assume we are given a system and a requirements specification and want to verify by means of testing that the system implements its specification. Conceptually every test activity starts with the description of *test cases* that have to be passed by the target system to which we also refer as *system under test* (SUT). In case of *requirements-driven testing* the test cases are derived from the specification and consist of sets of test inputs, execution conditions, and expected results developed for a particular objective. A test case describes explicitly or implicitly how to construct the test data, starting with initial inputs and possibly deriving consecutive data values and expected results during the test execution. In the context of reactive systems, a test case is a *process* running in parallel with the target system to be tested and interacting via the system interface.

The description of the test cases is followed by determining a *test procedure* which contains detailed instructions for the set-up and execution of the test cases (e.g. their order of execution), and instructions for the evaluation of results of executing the test cases.

For the latter task *expected results* are defined that provide an unambiguous specification of correct system behaviour as it may be observed during an execution of a test case. Since "correct" behaviour is defined by the specification document, the expected results should be consistent with the specification. In fact, in many approaches the expected results are defined by specification fragments.

With respect to exercising a test case on a system and monitoring the resulting behaviour one distinguishes two complementary kinds of tests. If only the interface behaviour of the SUT is influenced and monitored this is called *black-box testing*. If internal states or internal execution sequences of the SUT are also monitored or even influenced, the test is called a *white-box test*. White-box testing is often necessary in case of reactive systems, since due to nondeterminism the same sequence of inputs may stimulate different responses in different executions of the SUT.

4.2 Concepts

Motivated by the notions introduced above we can identify the following logical building blocks of test-automation systems, as depicted in Fig. 3.

The *test generator* is responsible for the creation of test cases from specifications. The *test driver* interprets the test cases provided by the test generator and controls their execution by writing data on input channels of the target system and collecting system outputs. The *test monitor* observes each test execution in order to (1) decide whether a specific test case has been performed for all relevant executions of the SUT that are possible for this case, and (2) to help to determine the level of test coverage obtained so far. The combination of a test driver and a test monitor implements the test procedure. As explained, in many applications the tasks of a test monitor cannot be performed solely based on black-box observations of the system interface. Instead, additional channels must be created, provid-

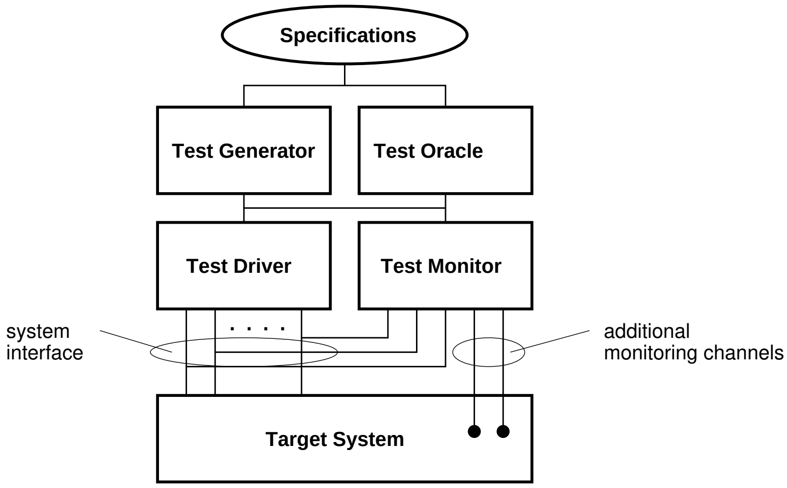

Figure 3. Logical building blocks of a test automation system.

ing internal state information about the SUT for the monitor. Finally, the *test oracle* evaluates the observed test execution against an expected results specification and decides whether the execution was correct.

Note that the specification document used for test evaluation is not necessarily the same as the specification used for test generation. E. g., for test generation we may use an explicit, executable specification. For the test oracle it may be preferable to use an implicit specification.

The two properties characterising a *trustworthy* test system are:

- *Soundness:* Only correct target system behaviour is accepted. This property requires that (1) the test generator creates for each possible implementation error violating the specification at least one test that uncovers the violation, (2) the test oracle rejects test executions violating the specification, and (3) the test monitor detects whether the test executions so far have covered all relevant target system behaviours.
- *Completeness:* Correct target system behaviour is never rejected. For this property it has to be ensured that (1) the test generator only creates tests which should really be successfully executed according to the specification, and (2) the test oracle never rejects test executions that are consistent with the specification.

M.-C. Gaudel used the terms 'validity' and 'unbias' in [Gau95] for these aspects of trustworthiness. However, we prefer 'soundness' and 'completeness' since this makes the close link between requirements-driven testing and formal verification more explicit: The execution of a test suite may be regarded as the claim *"every possible implementation behaviour is consistent with the specification, and the implementation is capable of performing all behaviours required"*. A trustworthy test system is a "mechanised proof theory" capable of falsifying every unjustified claim, so that only correct ones are accepted. In the context of proof theories this property is called soundness. Conversely, every trustworthy test system should be capable of verifying every valid claim, which corresponds to the completeness of a proof theory.

4.3 Test Strategies

Even for systems of moderate complexity exhaustive testing, i.e. performing every test case which potentially uncovers an implementation error, is in most cases infeasible. Thus it is necessary to investigate *test strategies* that select subsets of all relevant test cases and at the same time can be justified to analyse the "most important cases".

In order to support the development of strategies, the methodology introduced in Section 3 gives rise to a classification of different test types that may be derived from the decomposition tree paradigm depicted in Figure 2. Every type of boxes and bubbles shown in these trees gives rise to another sort of tests: The decomposition specifications $\mathsf{P} = \mathsf{A}(\mathsf{P1},\mathsf{P2},\ldots)$ in the architecture tree require tests investigating the consistency between the specified and the implemented system structure. Such tests are called *structure tests*. Tests inspecting the consistency of an arbitrary component in the normal behaviour tree with its conceptual model are called *requirements-driven tests*. Requirements-driven tests inspecting the acceptable behaviour tree are called *robustness tests*. Tests investigating the leaves of the trees in order to check consistency between a module implementation and its specification are called *unit tests*. Finally tests checking verification obligations

$$\frac{\mathsf{S}(\mathsf{P1}) \quad \mathsf{S}(\mathsf{P2})}{\mathsf{S}(\mathsf{P})} \quad [\ \mathsf{P} = \mathsf{A}(\mathsf{P1},\mathsf{P2})\ ]$$

are called integration tests. Below the different test types will be explained in more detail.

Observe that these different test types are already well-known in the software-engineering communities. Our specification and verification methodology offers an *a posteriori* justification that these notions are also relevant and complete from the formal methods point of view.

An important parameter for judging the trustworthiness of a test strategy is the obtained *test coverage*. This is a relative measure based on the specifications used for test generation. Examples are the percentage of functional requirements tested by at least one case (*functional requirements coverage*) or the percentage of statements covered by at least one test case (*code coverage*). From the perspective of test coverage exhaustive testing is the special case where the number of test cases exercised on the target system is sufficient to *prove* correctness of the SUT w.r.t. its specification.

Apart from selecting a subset of relevant test cases, test strategies also define the relative order of test case execution. This order influences the possibility to localise errors which is particularly important in case of nondeterministic systems displaying non-reproducible erroneous behaviour [Sch93].

For test case selection several heuristics exist. Again, they are justified by the considerations about development and verification of dependable systems described in Section 2 and 3 and can be classified as

- heuristics based on the architecture of the SUT,
- heuristics based on the classification of SUT behaviour,
- heuristics based on the different views of the SUT,
- heuristics based on the requirements specification structure.

Heuristics based on architecture

These heuristics identify relevant test cases on the basis of the SUT architecture. Tests may be exercised on the complete system as well as on isolated system components. In practice, combinations of tests on system level and tests on component level are developed in order to cope with the overall complexity of the test process.

Architecture-based tests are usually performed in bottom-up fashion, starting with unit tests. Depending on the granularity used for the test of a specific system, a unit may denote a block, a function, a thread, a task or any other coherent software unit. Unit tests investigate the conformance of the units with their requirements specifications.

Unit testing is followed by integration tests where new components are successively added to those already tested and then a test of the resulting configuration is performed. The basic objective of integration testing is to perform detailed tests for lower-level components Pi, such that the claim *"the* Pi *comply with their specifications* $\mathsf{S(Pi)}$*"* is justified. Then the higher-level component P is tested in order to demonstrate that the validity of specifications $\mathsf{S(Pi)}$ really imply $\mathsf{S(P)}$ in the architecture defined.

Finally tests are defined involving the complete system and its original interface plus auxiliary interfaces revealing internal details. These tests are called *system tests* and execute either in the real operational environment or use an environment simulation. The simulation directly triggers the physical interface of the target system and has additional monitoring channels to observe the resulting behaviour of the SUT.

Heuristics based on the classification of SUT behaviour

Assume that we have prepared a configuration which is to be tested. (According to the previous explanations this configuration may be a unit, the complete system or any other component.) As explained in Part I we distinguish between *normal* and *exceptional* behaviour of a system and its environment. In general it is advisable to start with tests only investigating the system behaviour in presence of proper environment operation. The investigation whether the system will show the required degree of fault-tolerance in presence of abnormal environment behaviour or internal defects is then tackled in a second test phase performing robustness tests. This heuristic is also supported by theoretic considerations about dependable systems [Pel95a], so a test strategy separating normal and exceptional behaviour tests can be formally justified.

Observe that exceptional behaviour often has to be tested on every level of decomposition, since faults do not only occur on system level but also inside the system. Therefore it may be necessary to perform tests w.r.t. normal environment behaviour and tests w.r.t. exceptional environment behaviour from module level all up to system level.

Heuristics based on system views

As introduced in Section 3.2 there are four views on a system used as a reference for test generation. We distinguish between three types of requirements-driven tests and introduce one type of investigating the correctness of architecture decomposition :

- *Functional testing* checks compliance of a system with its functional specification. The term is often used for all types of tests against the conceptual model. We prefer the clearer distinction according to the views introduced in Section 3.2.
- *Data testing* checks whether the correct data structures have been implemented (e. g. in a database system) and if correct data is stored.
- *Behavioural testing* checks the correctness properties related to the behavioural view, such as causality and synchronisation aspects.
- *Structure testing* checks the implemented structure of a system. It derives test cases from the implemented system, like branching of the code, the function call hierarchy, the data flow between processes etc.

The first three test types are *black-box* tests: Both test generation and the definition of expected results are solely derived from the conceptual model of the system. Observe that in case of structural testing in most cases one also wants to check whether the test execution is correct with respect to the conceptual model, therefore the expected results of a structural test can only be defined if such a model is available. Structural tests are typically *white-box* tests, since they need the component architecture or the detailed module design to generate the cases.

In accordance with these views this third kind of heuristic monitors functional, data, behavioural and structural tests in order to guarantee high reliability of the tested system without performing expensive or even impossible exhaustive testing.

Requirements-driven tests and structural tests are complementary: The former in general does not provide sufficient test coverage because it does not consider internal design, the latter will generally not detect missing functionality. Concerning their weighting we suggest to invest the main effort in functional, data, and behavioural testing, with the objective to cover the specified system functionality and behaviour as completely as possible. During functional testing, the internal system structures covered can be recorded. Afterwards structural tests can be defined and executed to cover the remaining system structure not reached by the preceding tests.

This heuristic is motivated by the following facts: (1) The most severe design and implementation errors are caused by erroneous specifications or their incorrect interpretation. Therefore tests comparing the implementation behaviour to the specification are likely to detect the most critical errors. (2) While structural tests cannot be created before the completion of the design phase, functional tests can be derived as soon as the specification is ready. In particular, they can also be used to *validate* the specification and test the design before entering the implementation phase. (3) The specification contains all the requirements relevant for the user and – in case of dependability requirements for a safety-critical system – the certification authorities. Therefore the acceptance test is usually based on the specification. These considerations are consistent with the observations made in [ROM89, RAO92, Hör95].

Heuristics based on the Requirements Specification Structure

As we have already observed in Section 3.2, also conceptual models are internally structured. This may be exploited for requirements-driven testing in order to define *test classes*, i. e., collections of test cases that can be regarded as equivalent, because successful execution of one class representative already indicates that the other test cases will also succeed with high probability. The requirements structure is well-suited to derive such test classes from specifications. This has been demonstrated for example in [HP95] in the case of test class derivation from Z specifications. Test classes are particularly useful for data-strong applications, where it is impossible to provide an exhaustive number of test cases for all the possible value combinations at the SUT interface. In the context of reactive systems with discrete interfaces the construction of test classes is less important, since discrete input values often cannot be regarded as equivalent because they stimulate different control functions in the SUT. Therefore we will not investigate test classes further in this article.

Part II

Test Automation With CSP

In the second part of this paper we will "instantiate" the testing methodology introduced in Part I for CSP.

5 Preliminaries

5.1 CSP Operators, Semantics and Refinement

In this section we introduce some notation and conventions used throughout the paper.

As indicated in previous sections tests and test drivers are specified in the process algebraic framework of *Communicating Sequential Processes (CSP)* [Hoa85]. We use the following set of CSP operators: $STOP$ (deadlock process), $SKIP$ (terminating process), $\rightarrow$ (prefixing), $\sqcap$ (internal choice), $\Box$ (external choice), $\|$ (parallel composition with synchronisation on common events), $\backslash$ (hiding), and $\hat{}$ (interrupt). The operator $(x : \{a_1, \ldots, a_n\} \rightarrow P(x))$ abbreviates the expression $a_1 \rightarrow P(a_1) \Box \ldots \Box a_n \rightarrow P(a_n)$, and $\sqcap_{x:\{a_1,\ldots,a_n\}}(x \rightarrow P(x))$ is an abbreviation for $a_1 \rightarrow P(a_1) \sqcap \ldots \sqcap a_n \rightarrow P(a_n)$. As basic programming operators we use **if then else**, $c?$ (input from channel c) and $g\&B$ (guarded command, g is the guard and B the body). For the specification of recursive processes we use sets of recursive equations rather than an explicit μ–operator. The alphabet of a process P is denoted by $\alpha(P)$. We use the standard semantics for CSP processes: $Traces(P) \subseteq \alpha(P)^*$ (trace semantics), $Fail(P) \subseteq \alpha(P)^* \times \mathbb{P}\,\alpha(P)$ (failure semantics, $\mathbb{P}$ denotes the power set operator), and $Div(P) \subseteq \alpha(P)^*$ (divergences of process P). The elements $s \in Traces(P)$ are the observable traces generated by P. A failure $(s, A) \in Fail(P)$ records the fact that process P may refuse to engage in any action of set A after having performed trace s. The *refusals* of a process P are defined by $Ref(P) =_{df} \{A \mid (\langle\,\rangle, A) \in Fail(P)\}$, where $\langle\,\rangle$ denotes the empty trace. $Ref(P)$ is subset-closed. Due to nondeterminism there may be several $A, A' \subseteq \alpha(P)$ with $A \in Ref(P)$ and $A' \in Ref(P)$, such that neither $A \subseteq A'$ nor $A' \subseteq A$. Due to subset-closure and since $\alpha(P)$ is finite, $Ref(P)$ is uniquely determined by the set

$$maxRef(P) =_{df} \{A : Ref(P) \mid \forall A' : Ref(P) - \{A\} \bullet A \nsubseteq A'\}$$

of maximum refusals. A divergence $s \in Div(P)$ denotes the situation that process P may diverge after having engaged in trace s. Diverging processes show completely unpredictable behaviour, denoted by $CHAOS$ in CSP.

Infinite behaviours are defined as limits of prefix closed sets of finite behaviours [Hoa85, p. 132]. We use the standard fixed point semantics for recursive processes. The set $maxTraces(P)$ denotes the union of the set of all terminated behaviours and the set of infinite behaviours of P.

We consider the following refinement relations:

- trace refinement:
  $P \sqsubseteq_T Q$ iff $Traces(Q) \subseteq Traces(P)$
- failures refinement:
  $P \sqsubseteq_F Q$ iff $Fail(Q) \subseteq Fail(P)$
- failure–divergence refinement:
  $P \sqsubseteq_{FD} Q$ iff $Fail(Q) \subseteq Fail(P) \wedge Div(Q) \subseteq Div(P)$.

For $x \in \{T, F, FD\}$ we define process equivalence $P =_x Q$ by $P \sqsubseteq_x Q \wedge Q \sqsubseteq_x P$. We use the abbreviation $P_I = P \setminus (\alpha(P) \setminus I)$. For $s \in \mathit{Traces}(P)$ the term P/s denotes the process that behaves like process P after having engaged in trace s.

For arbitrary (finite) sequences $s = \langle a_1, \ldots, a_n \rangle$ the function $\mathit{first}(s)$ returns a_1 and $\mathit{last}(s)$ returns a_n. The functions $\mathit{tail}(s), \mathit{front}(s)$ are defined by $s = \langle \mathit{first}(s) \rangle ^\frown \mathit{tail}(s)$ and $s = \mathit{front}(s) ^\frown \langle \mathit{last}(s) \rangle$, where $^\frown$ denotes concatenation on sequences. Function # returns the length of a sequence. The set $[P]^0$ is defined as $[P]^0 =_{df} \{e \in \alpha(P) \mid (\exists\, u \in \mathit{Traces}(P/s) \bullet \mathit{head}(u) = e)\}$. Predicate a **in** $\langle a_1, \ldots, a_n \rangle$ is true iff there exists $i \in \{1, \ldots, n\}$ with $a = a_i$. Relation $s \leq t$ denotes the prefix relation on sequences. Operator $\setminus$ stands for minus on sets.

5.2 Requirements Specifications for Test Automation With CSP

When instantiating our methodology described in Part I with CSP for the purpose of test automation, two types of requirements specifications S(IMP) are interpreted by the VVT-RT tool. The first consists of refinement relations

$$\mathsf{S(IMP)} \equiv_{df} \mathit{SPEC} \sqsubseteq_x \mathit{IMP}$$

where IMP denotes the implemented component to be tested and SPEC is an abstract specification process, explicitly described in the CSP process algebra. These specifications are used by the VVT-RT tool for automatic generation, execution and evaluation of tests, as will be explained below. The second type of requirements specifications takes the implicit form

$$\mathsf{S(IMP)} \equiv_{df} \mathit{IMP} \;\textbf{sat}\; \mathit{pred}(s)$$

where $\mathit{pred}(s)$ is a predicate about the admissible traces s of IMP. These specifications are only used in the VVT-RT test oracle for automatic offline evaluation of observed (timed) traces, using evaluation techniques described in [Hör95, Mik95].

The two specification formalisms can be used in combination during a development, since $\sqsubseteq_x$-refinement preserves implicit trace specifications,

$$\frac{\mathit{SPEC} \;\textbf{sat}\; \mathit{pred}(s) \quad \mathit{SPEC} \sqsubseteq_x \mathit{IMP}}{\mathit{IMP} \;\textbf{sat}\; \mathit{pred}(s)}$$

provided that $\mathit{pred}(s)$ only describes safety properties, i. e. does not make any assertions about possible continuations of s. In this paper we will focus on the first specification type since it is applicable in all stages of the test automation process.

The implementation IMP has to be regarded as a black box (possibly consisting of mixed hardware and software) where only the interface is known, so that an explicit representation as a CSP process is not available. In certain less frequent situations a faithful abstractions of IMP in CSP might be possible, as, for example, in the case of simple OCCAM software components. For such situations it is advisable to apply model checking with FDR [FDR94] instead of testing, since this allows an exhaustive exploration of the state space. For the rest of this paper we are only concerned with implementations that cannot be faithfully represented in CSP, so that testing is the only option to verify S(IMP).

5.3 Alternative Refinement Definitions

The notion of correctness of an implementation IMP w.r.t. an abstract specification process SPEC is given by the different refinement relations introduced above, depending on the semantics which is currently investigated. However, we slightly re-phrase these refinement notions in order to emphasise their relationship to the test classes introduced by Hennessy and de Nicola. (We assume without loss of generality that IMP and SPEC use the same set of visible interface events, while their internal hidden events may differ).

1. **Safety:** The implementation only generates traces allowed by the specification process. This corresponds to trace refinement, we have just introduced a new subscript S to indicate the relation to the safety notion:

   $$\mathit{SPEC} \sqsubseteq_S \mathit{IMP} \text{ iff } \mathit{Traces}(\mathit{IMP}) \subseteq \mathit{Traces}(\mathit{SPEC})$$

2. **Requirements Coverage:** After having engaged in trace s, the implementation never refuses a service which is guaranteed by the specification process.

   $$\mathit{SPEC} \sqsubseteq_C \mathit{IMP} \text{ iff } (\forall\, s : \mathit{Traces}(\mathit{SPEC}) \cap \mathit{Traces}(\mathit{IMP}) \bullet \mathit{Ref}(\mathit{IMP}/s) \subseteq \mathit{Ref}(\mathit{SPEC}/s))$$

   Since $\langle\,\rangle \in \mathit{Traces}(\mathit{SPEC}) \cap \mathit{Traces}(\mathit{IMP})$, this implies that a trace which can never be refused by SPEC will also be guaranteed by IMP.

3. **Non-Divergence:** The implementation may only diverge after engaging in trace s if also the specification process diverges after s.

   $$\mathit{SPEC} \sqsubseteq_D \mathit{IMP} \text{ iff } \mathit{Div}(\mathit{IMP}) \subseteq \mathit{Div}(\mathit{SPEC})$$

4. **Robustness:** An implementation is robust w.r.t. a specification process if every traces that can be performed by the specification process is also a valid trace of the implementation.

   $$\mathit{SPEC} \sqsubseteq_R \mathit{IMP} \text{ iff } \mathit{Traces}(\mathit{SPEC}) \subseteq \mathit{Traces}(\mathit{IMP})$$

The notion of robustness, introduced in [Bri89], can also be expressed as $\mathit{IMP} \sqsubseteq_T \mathit{SPEC}$. This relation has not received much attention in the literature about CSP refinement, though it is a common requirement in practical applications: For example, robustness covers the situation where the specification process contains nondeterminism for exception handling. Failures refinement only requires that every guaranteed behaviour of the specification process will also be performed by the implementation. Robustness additionally requires that exceptional behaviours of the specification process are also covered by the implementation.

The advantage of the new refinement notions is the possibility to give elegant alternative characterisations of these notions by means of mutually distinct test classes. Before introducing these test classes we state the following obvious relations between the standard and the new refinement notions.

Lemma 1

1. $\sqsubseteq_S \; = \; \sqsubseteq_T$
2. $\sqsubseteq_S \; \cap \; \sqsubseteq_C = \sqsubseteq_F$
3. $\sqsubseteq_S \; \cap \; \sqsubseteq_C \; \cap \; \sqsubseteq_D = \sqsubseteq_{FD}$

□

Furthermore we define $\sqsubseteq_{FDR} =_{df} \sqsubseteq_S \; \cap \; \sqsubseteq_C \; \cap \; \sqsubseteq_D \; \cap \; \sqsubseteq_R$ (failure-divergence refinement plus robustness).

5.4 Specification of Acceptable Behaviour With CSP

The nondeterministic operators of CSP allow to specify acceptable behaviour as nondeterministic deviations of normal behaviour, for example as

$$\begin{array}{lll} \mathsf{S}'(\mathsf{IMP}') & \equiv_{df} & SPEC' \sqsubseteq_x IMP' \\ SPEC' & =_{df} & SPEC \sqcap (fault \to REC) \\ SPEC & =_{df} & \ldots \text{normal behaviour} \ldots \\ REC & =_{df} & \ldots \text{recovery process} \ldots \end{array}$$

or

$$\begin{array}{lll} \mathsf{S}'(\mathsf{IMP}') & \equiv_{df} & SPEC' \sqsubseteq_x IMP' \\ SPEC' & =_{df} & SPEC^\frown(fault \to REC) \\ SPEC & =_{df} & \ldots \text{normal behaviour} \ldots \\ REC & =_{df} & \ldots \text{recovery process} \ldots \end{array}$$

Since acceptable behaviour includes normal behaviour, the abstract specification processes $SPEC$ and $SPEC'$ will generally be related by some refinement notion. While the implementation should be an S, F, or FD-refinement of $SPEC$ in most normal behaviour cases, it should be a robustness refinement of $SPEC'$ in most acceptable behaviour cases: $SPEC' \sqsubseteq_R IMP'$ requires that the $fault \to \ldots$-branch is really covered by the implementation. FD-refinement would allow us to ignore this branch completely.

Since we have chosen to use the same specification formalisms for normal and acceptable behaviour, the problem of automatic test generation, execution and evaluation does not require new solutions in the case of robustness tests. An alternative method relating normal and acceptable behaviour by means of implicit specifications is discussed in [Sch94, Pel95a].

6 Transition Graphs and Test Classes

In this section we describe an implementable encoding of the semantics of CSP processes by means of transition graphs. Afterwards we discuss those results of Hennessy's and de Nicola's testing theory [Hen88] that are relevant for the development of implementable test drivers.

6.1 Transition Graphs

Automated test generation will be performed by mechanised analysis of the specification, which results in a choice of traces and possible continuations to be exercised as test cases on the target system. *Automated test evaluation* will be performed by observing traces and their continuations in the target system and checking mechanically, if these behaviours are correct with respect to the specification. Obviously, these tasks are fundamentally connected to the problem of mechanised *simulation* of the specification which is in general based on the following theorem [Hen88, p. 94].

Theorem 1 (Normal Form Theorem) *Let P be a CSP process, interpreted in the failures-divergence model.*

1. *If $\langle\,\rangle \notin Div(P)$, then*
   $P =_{FD} \sqcap_{R:Ref(P)} (x : ([P]^0 \setminus R) \to P/\langle x\rangle)$
2. *If $Div(P) = \emptyset$, then $P/s =_{FD} P(s)$ with $P(s) =_{df}$*
   $\sqcap_{R:Ref(P/s)} (x : ([P/s]^0 \setminus R) \to P(s^\frown\langle x\rangle))$
3. *For arbitrary P, $P \sqsubseteq_{FD} P(\langle\,\rangle)$ holds.*

□

This theorem shows how CSP specifications can be symbolically executed: choose a valid refusal set R of P/s at random, engage into any one of the remaining events $e \in [P/s]^0 \setminus R$ and continue in state $P/s^\frown\langle e\rangle$. Given an implementation of a simulator, the problem of test generation for a given specification can be related to the task of finding executions performable by the simulator. Test evaluation can be performed by determining whether an execution of the real system is also a possible execution of the simulator.

With these general ideas in mind, the first problem to solve is how to retrieve the semantic representation – i. e., the failures and divergences – of a specification written in CSP syntax. This has been solved by Formal Systems Ltd and implemented in the FDR system [Ltd94], for the subset of CSP specifications satisfying:

- The specification only uses a *finite* alphabet. As a consequence, each channel admits only a finite range of values.
- Each sequential process which is part of the full specification can be modelled using a finite number of states.
- The CSP syntax is restricted by a separation of operators into two levels: The *lower-level process language* describes isolated communicating sequential processes by means of the operators $\to$, $\sqcap$, $\Box$, $;$, $X = F(X)$. The *composite process language* uses the operators $\|$, $|||$, $^\frown$, $\setminus$, f^* to construct full systems out of lower-level processes.

Under these conditions the CSP specification may be represented as a *labelled transition system* [Mil89] which can be encoded as a *transition graph* with only a finite number of nodes and edges. Basically, the nodes of this directed graph are constructed from Hennessy's *Acceptance Tree* representation [Hen88] by identifying semantically equivalent nodes of the tree in a single node of the transition graph. The edges of the graph are labelled with events, and the edges leaving one node carry distinct labels. Therefore, since the alphabet is finite, the number of leaving edges is also finite. A distinguished node represents the equivalence class of the initial state of the process P. A directed walk through the graph, starting in the initial state and labelled by the sequence of events $\langle e_1, \ldots, e_n \rangle$ represents the trace $s = \langle e_1, \ldots, e_n \rangle$ which may be performed by P. The uniquely determined node reached by the walk s represents the equivalence class of process state P/s. The labels of the edges leaving this node in the graph correspond to the set $[P/s]^0$ of events that may occur for process P after having engaged in s. The set of internal states reachable in process P after s is encoded in one node of the transition graph as the collection of their refusal sets, one for each internal state. If two directed walks s and u lead to the same node in the transition graph, this means that $P/s = P/u$ holds in the failures model.

The problem of automatic test evaluation now can be re-phrased as follows: A test execution results in a trace performed by the implementation. Evaluating the transition graph, it may be verified whether this execution is correct according to the specification. The problem of test generation is much more complex: Theoretically, the transition graph defines exactly the acceptable behaviours of the implementation. But at least for non-terminating systems, this involves an infinite number of possible executions. Therefore the problem how to find relevant test cases and how to decide whether sufficiently many test executions have been performed on the target system has to be carefully investigated.

6.2 Test Classes

In this section we recall results of Hennessy's and de Nicola's testing theory [Hen88] that are relevant for the construction of the test drivers in Section 7.3.

Hennessy introduced processes U, so-called *experimenters*, with $\alpha(SPEC) = \alpha(U) \setminus \{w\}$, where w is a specific event denoting successful execution of the experiment which consists of U running in parallel with the process to be tested [1]. Experimenters coincide with our notion of *test cases*, so we will only use the latter term. An execution of the test case U for the test of some system P is a trace $s \in \mathit{Traces}(P \parallel U)$. The execution is successful if event w occurs in s. Depending on U and P, two satisfaction relations may be distinguished with respect to the outcome of test executions.

Definition 1 *For a process P and an associated test case U we say:*

$P \ \underline{\mathit{may}}\ U \equiv_{df} (\exists\, s : \mathit{Traces}(P \parallel U) \bullet \langle w \rangle \ \mathbf{in}\ s)$

$P \ \underline{\mathit{must}}\ U \equiv_{df} (\forall\, s : \mathit{maxTraces}(P \parallel U) \bullet \langle w \rangle \ \mathbf{in}\ s)$

□

$P \ \underline{\mathit{may}}\ U$ holds if there exists at least one successful execution of $(P \parallel U)$. Only if every maximum execution of $(P \parallel U)$ leads to success $P \ \underline{\mathit{must}}\ U$ holds.

[1] In [Hen88] also another local experimenter event '1' has been introduced which enables the experimenter to control the course of a test execution. However, for the application of Hennessy's and de Nicola's theory to CSP, this event is not needed.

Note that in general we cannot construct test cases that indicate *failure* in addition to success, because the failure may materialise as a situation where the test execution is blocked or diverges. Even if only non-diverging processes are tested we would need a priority concept for transitions. In the TCCC example expected events always have to occur within certain time bounds, so failures may be detected by means of timeouts.

Based on the introduced refinement notions we classify tests according to their capability to detect certain implementation faults.

Definition 2 *Let U be a test case.*

1. *U detects* safety failure *s iff*
   $(\forall\, P \bullet P \ \underline{\mathit{must}}\ U \Rightarrow s \notin \mathit{Traces}(P))$
2. *U detects* requirements coverage failure *(s, A) iff*
   $(\forall\, P \bullet P \ \underline{\mathit{must}}\ U \Rightarrow (s, A) \notin \mathit{Fail}(P))$
3. *U detects* divergence failure *s iff*
   $(\forall\, P \bullet P \ \underline{\mathit{must}}\ U \Rightarrow s \notin \mathit{Div}(P))$
4. *U detects* robustness failure *s iff*
   $(\forall\, P \bullet P \ \underline{\mathit{may}}\ U \Rightarrow s \in \mathit{Traces}(P))$

□

A main result of [Hen88] is the definition of test classes which detect exactly the failures introduced in the previous definition.

Definition 3 *For a given specification $SPEC$, let $s \in \alpha(SPEC)^*$, $a \in \alpha(SPEC)$, and $A \subseteq \alpha(SPEC)$. The class of* Hennessy Test Cases *is defined by the following collection of test cases:*

1. Safety Tests $U_S(s, a)$:

$$
\begin{array}{l}
U_S(s,a) =_{df} \\
\quad \mathbf{if}\ s = \langle\,\rangle \\
\quad\quad \mathbf{then}\ (w \to \mathit{SKIP} \,\Box\, a \to \mathit{SKIP}) \\
\quad\quad \mathbf{else}\ (w \to \mathit{SKIP} \,\Box\, \mathit{head}(s) \to U_S(\mathit{tail}(s), a))
\end{array}
$$

2. Requirements Coverage Tests $U_C(s, A)$:

$$
\begin{array}{l}
U_C(s,A) =_{df} \\
\quad \mathbf{if}\ s = \langle\,\rangle \\
\quad\quad \mathbf{then}\ (a : A \to w \to \mathit{SKIP}) \\
\quad\quad \mathbf{else}\ (w \to \mathit{SKIP} \,\Box\, \mathit{head}(s) \to U_C(\mathit{tail}(s), A))
\end{array}
$$

3. Divergence Tests $U_D(s)$:

$$
\begin{array}{l}
U_D(s) =_{df} \\
\quad \mathbf{if}\ s = \langle\,\rangle \\
\quad\quad \mathbf{then}\ w \to \mathit{SKIP} \\
\quad\quad \mathbf{else}\ (w \to \mathit{SKIP} \,\Box\, \mathit{head}(s) \to U_D(\mathit{tail}(s)))
\end{array}
$$

4. Robustness Tests $U_R(s)$:

$$
\begin{array}{l}
U_R(s) =_{df} \\
\quad \mathbf{if}\ s = \langle\,\rangle \\
\quad\quad \mathbf{then}\ w \to \mathit{SKIP} \\
\quad\quad \mathbf{else}\ \mathit{head}(s) \to U_R(\mathit{tail}(s))
\end{array}
$$

□

Definition 3 is motivated by the following lemma:

Lemma 2

1. $U_S(s,a)$ *detects safety failure* $s \frown \langle a \rangle$.
2. $U_C(s,A)$ *detects req. coverage failure* (s,A).
3. $U_D(s)$ *detects divergence failure* s.
4. $U_R(s)$ *detects robustness failure* s.

□

Note that the Hennessy test classes even *characterise* the associated failure types: If $s \frown \langle a \rangle \notin \mathit{Traces}(P)$ then P $\underline{must}$ $U_S(s,a)$ follows. Analogous results hold for $U_C(s,A), U_D(s), U_R(s)$.

In our context $s \in \mathit{Div}(P)$ means $P/s = \mathit{CHAOS}$ in the sense of [Hoa85], that is, P/s may both *diverge internally (livelock)* and produce and refuse arbitrary *external* events. The tests $U_D(s)$ have been designed by Hennessy to detect internal divergence only. Conversely, the tests $U_S(s,a)$ and $U_C(s,A)$ can detect external chaotic behaviour but cannot distinguish internal divergence from deadlock. However, using the three test classes together enables us to distinguish deadlock, livelock and external chaotic behaviour. Note that P $\underline{must}$ $U_S(s,a)$ also implies $s \notin \mathit{Div}(P)$, because divergence along s would imply that every continuation of s, in particular $s \frown \langle a \rangle$ would be a trace of P. P $\underline{must}$ $U_C(s,A)$ implies $s \notin \mathit{Div}(P)$, because divergence along s implies the possibility to refuse every subset of $\alpha(P)$ after s.

Hennessy's results about the relation between testing and refinement can be re-phrased for our context as follows:

Theorem 2

1. *If for all* $a \in \alpha(\mathit{SPEC})$, $s \in \alpha(\mathit{SPEC})^*$
   SPEC $\underline{must}$ $U_S(s,a)$ *implies* IMP $\underline{must}$ $U_S(s,a)$
   then $\mathit{SPEC} \sqsubseteq_S \mathit{IMP}$.
2. *If for all* $s \in \mathit{Traces}(\mathit{SPEC})$ *and* $A \subseteq \alpha(\mathit{SPEC})$
   SPEC $\underline{must}$ $U_C(s,A)$ *implies* IMP $\underline{must}$ $U_C(s,A)$
   then $\mathit{SPEC} \sqsubseteq_C \mathit{IMP}$.
3. *If for all* $s \in \alpha(\mathit{SPEC})^*$
   SPEC $\underline{must}$ $U_D(s)$ *implies* IMP $\underline{must}$ $U_D(s)$
   then $\mathit{SPEC} \sqsubseteq_D \mathit{IMP}$.
4. *If for all* $s \in \alpha(\mathit{SPEC})^*$
   SPEC $\underline{may}$ $U_R(s)$ *implies* IMP $\underline{may}$ $U_R(s)$
   then $\mathit{SPEC} \sqsubseteq_R \mathit{IMP}$.

□

If $\mathit{SPEC} \sqsubseteq_D \mathit{IMP}$ holds, the four implications of the theorem become equivalences. Theorem 2 shows that in principle only *requirements-driven* test design is needed: It is only necessary to execute test cases that will succeed for the specification. Due to possible nondeterminism in $\mathit{SPEC}, \mathit{IMP}$ and U the properties covered by Theorem 2 cannot be verified by means of black-box tests alone, because they require the analysis of *every* possible execution of $\mathit{SPEC} \parallel U$ and $\mathit{IMP} \parallel U$. Thus, as indicated in the first part of this paper a *test monitor* collecting information about the executions performed so far is, in general, unavoidable. Note, that this is no disadvantage of the defined classes of tests but inherent in every testing approach that is sensitive to nondeterminism.

7 Minimal Test Classes and Test Drivers

The previous section summarised the relevant *theoretical* aspects of testing for our approach. However, when constructing test drivers one also has to deal with *pragmatical* concerns, such as implementability. This includes the definition of *minimum* test classes to avoid redundancy, characterisation of test strategies that eventually reveal every possible implementation failure, and last but not least the implementation of such strategies by test drivers that simultaneously simulate the operational environment of the process to be tested. These topics will be discussed in this section.

7.1 Minimal Test Classes

When performing a test suite to validate the correctness properties of a system, a crucial objective is to perform a *minimum* number of test cases. The following definition specifies minimal sets of Hennessy tests, which are still trustworthy in the sense that if the implementation passes these tests then it is a refinement of the specification w.r.t. the currently chosen semantics. We use the abbreviation $T_{-D}(\mathit{SPEC}) =_{df} \mathit{Traces}(\mathit{SPEC}) - \mathit{Div}(\mathit{SPEC})$.

Definition 4 *For a given specification* SPEC, *we define the following collections of test cases:*

1. $\mathcal{H}_S(\mathit{SPEC}) =_{df}$
   $\{U_S(s,a) \mid s \in T_{-D}(\mathit{SPEC}) \wedge a \notin [\mathit{SPEC}/s]^0\}$
2. $\mathcal{H}_C(\mathit{SPEC}) =_{df}$
   $\{U_C(s,A) \mid s \in T_{-D}(\mathit{SPEC}) \wedge A \subseteq [\mathit{SPEC}/s]^0$
   $\wedge (\forall R : \mathit{Ref}(\mathit{SPEC}/s) \bullet A \not\subseteq R)$
   $\wedge (\forall X : \mathbb{P}\, A - \{A\} \bullet$
   $(\exists R : \mathit{Ref}(\mathit{SPEC}/s) \bullet X \subseteq R))\}$
3. $\mathcal{H}_D(\mathit{SPEC}) =_{df}$
   $\{U_D(s) \mid s \in T_{-D}(\mathit{SPEC})$
   $\wedge (\forall u : T_{-D}(\mathit{SPEC}) \bullet$
   $s \le u \wedge [\mathit{SPEC}/u]^0 = \emptyset \Rightarrow s = u)\}$
4. $\mathcal{H}_R(\mathit{SPEC}) =_{df}$
   $\{U_R(s) \mid s \in \mathit{Traces}(\mathit{SPEC})$
   $\wedge (\forall u : \mathit{Traces}(\mathit{SPEC}) \bullet$
   $s \le u \wedge [\mathit{SPEC}/u]^0 = \emptyset \Rightarrow s = u)\}$

□

The following theorems state that in order to characterise the refinement notions addressed in Theorem 2, it suffices already to exercise the tests specified in Definition 4 on the implementation. Compared to the full set of Hennessy tests, defined for *all* sequences $s \in \alpha(\mathit{SPEC})^*$ of events and sets $A \subseteq \alpha(\mathit{SPEC})$, this represents a considerable reduction of the test cases to be considered.

Theorem 3 *Given a specification* SPEC.

1. *If* $U \in \mathcal{H}_S(\mathit{SPEC}) \cup \mathcal{H}_C(\mathit{SPEC}) \cup \mathcal{H}_D(\mathit{SPEC})$
   then SPEC $\underline{must}$ U *holds.*
2. *If* $U \in \mathcal{H}_R(\mathit{SPEC})$ *then* SPEC $\underline{may}$ U *holds.*

□

Theorem 4 *Given* SPEC *and the corresponding test classes* $\mathcal{H}_x(\mathit{SPEC})$, $x \in \{S,C,D,R\}$, *the following properties hold:*

1. *If* IMP $\underline{must}$ U *for all* $U \in \mathcal{H}_S(\mathit{SPEC})$,
   then $\mathit{SPEC} \sqsubseteq_S \mathit{IMP}$.

2. *If IMP $\underline{must}$ U for all $U \in \mathcal{H}_C(SPEC)$,*
   then $SPEC \sqsubseteq_C IMP$.
3. *If IMP $\underline{must}$ U for all $U \in \mathcal{H}_D(SPEC)$,*
   then $SPEC \sqsubseteq_D IMP$.
4. *If IMP $\underline{may}$ U for all $U \in \mathcal{H}_R(SPEC)$,*
   then $SPEC \sqsubseteq_R IMP$.

□

Theorem 3 represents the main result about the combination of refinement and testing in the sense of the theory links discussed in Section 3.6. It shows that refinement assertions in our decomposition trees can be consistently replaced by corresponding testing assertions. In addition, the theorem shows that for terminating systems refinement properties can be verified by performing a finite number of tests. (Note, that all processes have only *finite* internal nondeterminism.)

The definitions of $\mathcal{H}_S, \mathcal{H}_C, \mathcal{H}_D$ indicate further that it is not necessary to perform any tests for traces s after which $SPEC$ diverges, since in such a case $SPEC/s$ will allow chaotic behaviour which does not restrict the admissible behaviours of IMP/s. (Of course, it is questionable if specifications allowing divergence will be used in practice at all.) For the test of safety properties, the definition of $\mathcal{H}_S$ states that we only have to use those test cases $U_S(s, a)$, where s is a trace of $SPEC$, but $SPEC/s$ does not admit event a. For the requirements coverage tests $U_C(s, A), \mathcal{H}_C$ indicates that only the smallest sets A, such that $SPEC/s$ can never refuse A completely, have to be tested. As a consequence, it is not necessary to exercise any tests $U_C(s, A)$, if $SPEC/s$ may refuse the full alphabet.

The definitions of $\mathcal{H}_D$ and $\mathcal{H}_R$ are motivated by the fact that for the test of divergence and robustness properties we only have to analyse *maximum* traces: If $SPEC$ terminates or blocks after a trace u, the tests corresponding to proper prefixes of u are covered by $U_D(u)$ and $U_R(u)$, so only the latter are contained in $\mathcal{H}_D$ and $\mathcal{H}_R$ respectively.

The next theorem investigates minimality of the test classes $\mathcal{H}_S$ and $\mathcal{H}_C$ defined above.

Theorem 5 *Given SPEC and the corresponding test classes $\mathcal{H}_S, \mathcal{H}_C$, the following properties hold:*

1. *If $\mathcal{H} \subset \mathcal{H}_S$ there exists a process P satisfying P $\underline{must}$ U for all $U \in \mathcal{H}$ but not refining SPEC in the trace model.*
2. *If $\mathcal{H} \subset \mathcal{H}_C$ there exists a process P satisfying P $\underline{must}$ U for all $U \in \mathcal{H}_S \cup \mathcal{H}$ but not refining SPEC in the failures model.*
3. *If $U_C(s, A) \in \mathcal{H}_C$ and $B \subset A$ then*
   $\neg$ (SPEC $\underline{must}$ $U_C(s, B)$).

□

Theorem 5 shows that $\mathcal{H}_S$ and $\mathcal{H}_C$ are indeed minimal: If one test $U(s, a)$ is removed from $\mathcal{H}_S$, a process with safety failure $s \frown \langle a \rangle$ could be constructed, for which all the remaining tests would succeed. Removing a test $U_C(s, A)$ from $\mathcal{H}_C$ would admit processes P satisfying the remaining tests without refining $SPEC$ in the failures model. Moreover, the set A cannot be reduced in $U_C(s, A)$ in $\mathcal{H}_C$, since otherwise $SPEC$ would no longer pass this test.

The test collections $\mathcal{H}_D$ and $\mathcal{H}_R$, however, cannot be defined as minimal sets, as soon as $SPEC$ describes a nonterminating system: If $s \in maxTraces(SPEC)$ is an infinite computation of $SPEC$, $\mathcal{H}_D$ and $\mathcal{H}_R$ must contain infinitely many tests associated with prefixes $s_1 < s_2 < s_3 < \ldots$ of s, and each infinite subset of these tests would suffice to verify correct behaviour along s. At least we can state that any $\mathcal{H}'_D \subseteq \mathcal{H}_D$ satisfying

$$(\forall\, u : T_{-D}(SPEC) \bullet \exists\, s : \mathcal{H}'_D \bullet u \leq s)$$

is sufficient to detect divergence failures against $SPEC$ and any $\mathcal{H}'_R \subseteq \mathcal{H}_R$ satisfying

$$(\forall\, u : T_{-D}(SPEC) \bullet \exists\, s : \mathcal{H}'_R \bullet u \leq s)$$

is sufficient to detect robustness failures.

7.2 Mixed Verification and Test Strategies

The testing and refinement equivalences described in Theorem 4 allow us to develop mixed verification and test strategies increasing the efficiency of the overall verification process. This will be exemplified by means of a typical verification obligation

$$\frac{\mathsf{S(P1)} \quad \mathsf{S(P2)}}{S(P)} \quad [\, \mathsf{P = A(P1, P2)} \,]$$

with

$$\begin{array}{ll} \mathsf{S(P1)} & =_{df} SPEC1 \sqsubseteq_S P1 \\ \mathsf{S(P2)} & =_{df} SPEC2 \sqsubseteq_S P2 \\ \mathsf{S(P)} & =_{df} SPEC \sqsubseteq_S P \\ \mathsf{A(P1, P2)} & =_{df} P1 \,\|\, P2 \end{array}$$

as it might occur during an integration test suite. If we had to discharge this obligation by means of testing alone, four test series would be necessary:

1. requirements-driven tests $P1\,\underline{must}\,U$ for all $U \in \mathcal{H}_S(SPEC1)$ to show that the premise $\mathsf{S(P1)}$ holds,
2. requirements-driven tests $P2\,\underline{must}\,U$ for all $U \in \mathcal{H}_S(SPEC2)$ to show that the premise $\mathsf{S(P2)}$ holds,
3. requirements-driven tests $P\,\underline{must}\,U$ for all $U \in \mathcal{H}_S(SPEC)$ to show that the conclusion $\mathsf{S(P)}$ holds,
4. structure tests $P\,\underline{must}\,U$ for all $U \in \mathcal{H}_S(P1 \,\|\, P2)$ to show that $P \sqsubseteq_S P1 \,\|\, P2$ holds.

Let us now examine how the amount of testing can be decreased by discharging a part of these points using formal verification: First observe that the above verification obligation might be formally verified under the premise that $\mathsf{S(P1)}$, $\mathsf{S(P2)}$ and $P = P1 \,\|\, P2$ holds. Then the third test series would become superfluous; execution of the other three would just ensure the prerequisites for the formal verification step. The next possibility would be to investigate the premise $P = P1 \,\|\, P2$ by analytical methods. In an implementation language like OCCAM this could be trivially checked since the OCCAM parallel operator has the same semantics as in CSP.

7.3 Test Drivers

As explained in Part I test drivers are hardware and/or software devices controlling the executions of test cases for a target system. To formalise this notion, recall that a *context* in CSP is a term $\mathcal{C}(X)$ with a free identifier X. Apart from the free identifier X, $\mathcal{C}(X)$ may contain other CSP processes as parameters.

Definition 5 *A* Test Driver for the test against *SPEC is a context* $\mathcal{D}(X)$ *using test cases* U_i *satisfying* $\alpha(SPEC) = \alpha(U_i) \setminus \{w\}$ *as parameters.*
□

We will focus on test drivers of the form

$$\begin{aligned}\mathcal{D}(X) =_{df} (i := 0);\ *(U_i \parallel X\hat{\ }(w \to monitor?next \\ \to (\textbf{if}\ next\ \textbf{then}\ i := i + 1;\ SKIP \\ \textbf{else}\ SKIP)));\end{aligned}$$

with test cases U_i. A test driver of this type will execute the test cases in a certain order $U_1, U_2, \ldots$; one test case at a time and with only one copy of the target system $X = IMP$ running. As soon as a test case signals success w, the execution will be interrupted. An input $monitor?next$ will be required from a process monitoring the test coverage achieved so far with the actual test U_i. The implementation of test monitors is not addressed in this paper. If the monitor signals $next = true$, the next test case U_{i+1} will be performed, otherwise U_i will be repeated. If U_i is a $\underline{may}$-test, $next$ is always set to $true$.

The main criterion that test drivers have to satisfy is given in the next definition.

Definition 6 *Let* $\mathcal{D}(X)$ *be a test driver for the test against* $SPEC$*, performing test cases of a collection* $\mathcal{U}$ *in the order* $U_1, U_2, U_3, \ldots$*. Let* $\sqsubseteq \in \{\sqsubseteq_T, \sqsubseteq_F, \sqsubseteq_{FD}, \sqsubseteq_R\}$*. Then* $\mathcal{D}(X)$ *is called* trustworthy for $\sqsubseteq$-test against $SPEC$*, iff the following conditions hold:*

1. *$\mathcal{U}$ contains a subset $\mathcal{U}_{\sqsubseteq}$ which characterises $\sqsubseteq$-refinement against $SPEC$.*
2. *For every (safety-, requirements coverage-, divergence- respectively robustness-) failure violating $\sqsubseteq$, there exists an $n \in \mathbb{N}$ such that $U_n \in \mathcal{U}_{\sqsubseteq}$ can detect this failure in the sense of Definition 2.*

□

Definition 6 covers the intuitive understanding of trustworthiness in a formal way: Condition 1 re-phrases the soundness and completeness requirement for trustworthy test systems described in Part I in the case of CSP refinement specifications. The second condition ensures that any specification violation of IMP can be uncovered by a test case which is guaranteed to be chosen by the driver after a *finite* number of other test cases.

Theorem 6 *The test driver*

$$\begin{aligned}\mathcal{D}(X) =_{df} (i := 0);\ *(U_i \parallel X\hat{\ }(w \to monitor?next \\ \to (\textbf{if}\ next\ \textbf{then}\ i := i + 1;\ SKIP \\ \textbf{else}\ SKIP)));\end{aligned}$$

applying the tests $U \in \mathcal{H}$ according to Definition 4, ordered by the length of the defining traces, is trustworthy for $\sqsubseteq_{FDR}$-refinement.
□

Analogous results hold for the other refinement notions $\sqsubseteq_S$, $\sqsubseteq_C$, $\sqsubseteq_R$, $\sqsubseteq_D$, $\sqsubseteq_F$, $\sqsubseteq_{FD}$.

7.4 Test Drivers for Reactive Systems

The testing methodology presented so far will now be specialised on the development of test drivers for the automated test of *reactive systems*. Following the decomposition tree paradigm introduced in Part I, it is useful to distinguish between the target system and its operational environment in an explicit way. The objective of the test suite is to ensure the correct behaviour of the target system when running in an operational environment satisfying its requirements specification. Therefore test drivers have to *test* the target system behaviour while simultaneously *simulating* the operational environment. This holds for all requirements-driven tests or robustness tests to be exercised on components of the architecture decomposition tree.

In general, the configuration of a reactive system and its environment will be appropriately described by the following definition:

Definition 7 *A* standard configuration *(for reactive systems)* $(E, ASYS, SYS, I)$ *consists of CSP processes* $E, ASYS, SYS$ *and a set I of events such that* $I = \alpha(E) \cap \alpha(ASYS) = \alpha(E) \cap \alpha(SYS)$*. Process E represents the* environment*.* $SPEC = (E \parallel ASYS)_I$ *is called the* specification process*, and* $IMP = (E \parallel SYS)_I$ *the* implementation*. For* $\sqsubseteq \in \{\sqsubseteq_T, \sqsubseteq_F, \sqsubseteq_{FD}, \sqsubseteq_{FDR}, \sqsubseteq_R, \sqsubseteq_C, \sqsubseteq_D\}$*, a standard configuration is called* $\sqsubseteq$-correct*, if* $SPEC \sqsubseteq IMP$ *holds.*
□

Note that in a standard configuration $(E \parallel ASYS)_I = (E_I \parallel ASYS_I)$ and $(E \parallel SYS)_I = (E_I \parallel SYS_I)$ holds, because the hiding operator distributes through $\parallel$, if none of the interface events shared between the parallel components are concealed [Hoa85, p. 112].

Now we are prepared to state the main result of this article, an implementable test driver that is trustworthy for $\sqsubseteq_{FD}$-refinement. The test driver uses test cases derived from the Hennessy Test Cases introduced in Definition 4 and simultaneously simulates the operational environment of the process to be tested. The properties of these test cases are formally expressed by Theorem 7. Their main advantage when compared to the Hennessy Test Cases is that they allow to investigate safety, requirements coverage and non-divergence at the same time, while the Hennessy Cases require to perform different test suites for each correctness feature. Therefore our test cases are more efficient in practical applications.

Theorem 7 *Let* $(E, ASYS, SYS, I)$ *be a standard configuration of a reactive system. Define a collection* $\mathcal{U} = \{U(n) \mid n \in \mathbb{N}\}$ *of test cases by:*

$$U(n) =_{df} U(n, \langle\, \rangle)$$

$$
\begin{array}{l}
U(n,s) =_{df} \\
\quad (\#s = n \vee A(s) = \emptyset) \& (w \rightarrow SKIP) \\
\quad \Box \\
\quad (\#s < n) \& \\
\qquad (e : ([E_I / s]^0 \setminus [ASYS_I / s]^0) \rightarrow \dagger \rightarrow SKIP) \\
\quad \Box \\
\quad (\#s < n - 1 \wedge R(s) \neq \emptyset) \& \\
\qquad (\sqcap_{R:R(s)} U(n, s, [(E \parallel ASYS)_I / s]^0 \setminus R)) \\
\quad \Box \\
\quad (\#s = n - 1 \wedge A(s) \neq \emptyset) \& \\
\qquad (\sqcap_{R:maxRef(E_I/s), A:A(s)} U(n, s, A \setminus R))
\end{array}
$$

$$U(n, s, M) = e : M \rightarrow U(n, s^\frown \langle e \rangle)$$

where

$$
\begin{array}{rl}
A(s) = \{ A : \mathbb{P}\, I \mid & A \subseteq [(E \parallel ASYS)_I / s]^0 \\
& \wedge (\forall R : Ref((E \parallel ASYS)_I / s) \bullet A \not\subseteq R) \wedge \\
& \wedge (\forall X : \mathbb{P}\, A - \{A\} \bullet \\
& \qquad (\exists R : Ref((E \parallel ASYS)_I / s) \bullet X \subseteq R)) \}
\end{array}
$$

and

$$R(s) = \{ R : Ref(E_I / s) \mid [(E \parallel ASYS)_I / s]^0 \setminus R \neq \emptyset \}.$$

Then

1. *If $Traces(E_I) \cap Div(ASYS_I) = \emptyset$ then $ASYS_I$ <u>must</u> $U(n)$ for all test cases $U(n) \in \mathcal{U}$.*
2. *The following statements are equivalent:*
   (a) *SYS_I <u>must</u> $U(n)$ for all test cases $U(n) \in \mathcal{U}$.*
   (b) *$Traces(E_I) \cap Div(SYS_I) = \emptyset$ and $(E \parallel ASYS)_I \sqsubseteq_{FD} (E \parallel SYS)_I$.*

$\Box$

Each test case $U(n)$ explores the behaviour of the target system for traces s of length $\#s \leq n$. The basic idea of the structure of $U(n)$ is to simulate the environment E_I with respect to traces and refusals while exercising a combination of test cases $U_S(s, a)$ and $U_C(s, A)$ on the target system. $U(n)$ uncovers all trace failures up to length $n - 1$ and detects all requirements coverage violations occurring in the next step after having run through a trace of length $n - 1$. This means that the tasks of the test oracle are integrated in the test cases and performed during their execution. This is called *on-the-fly test evaluation*. $U(n, s)$ represents the state of a test execution where trace s has already been successfully performed. At each execution step, $U(n, s)$ will detect any event $e \in ([E_I / s]^0 \setminus [ASYS_I / s]^0)$, which is acceptable according to the environment but corresponds to a trace failure of the target system SYS. This will be indicated by a special event $\dagger$, signalling failure of the test, if the target system does not diverge before indication becomes possible.

As long as $\#s < n - 1$, $U(n, s)$ will behave as E_I / s with respect to the refusal of events: In the third $\Box$-branch an arbitrary refusal $R \in R(s)$ may be selected, and every event outside R that may be performed by $(E \parallel ASYS)_I / s$ is offered to the target system. $R(s)$ is the set of all E_I / s-refusals that do not block further operation of $(E \parallel ASYS)_I$ completely. If the target system SYS_I may "legally" block in environment E_I after trace s because the same holds for $ASYS_I$, this situation is reflected by $A(s) = \emptyset$. Now the first $\Box$-branch offers successful termination because such a deadlock may occur nondeterministically, but does not indicate failure. At the same time the third $\Box$-branch still offers events $[(E \parallel ASYS)_I / s]^0$ for further successful execution, so that full trace coverage can be reached if every possible execution of $(U(n) \parallel SYS_I)$ is carried out. Since $Ref(E_I / s)$ is subset-closed, $R(s)$ is empty iff $[(E \parallel ASYS)_I / s]^0 = \emptyset$, that is iff $(E \parallel ASYS)_I$ always deadlocks after s.

For $\#s = n$, $U(n, s)$ will only admit events contained in a minimum set $A \in A(s)$ that cannot be completely refused by $(E \parallel ASYS)_I / s$. Therefore $U(n)$ can detect requirement coverage failures of SYS occurring after traces of length n, when running in environment E. There is a subtle difference between the third and the fourth ALT-branch: To detect requirement coverage failures in the forth branch it obviously suffices to select the *maximum* refusals R in the expressions $A \setminus R$. If $(E \parallel ASYS)_I / s = P \sqcap STOP$ for some process P, the maximum refusal is the full alphabet I and $A(s)$ is empty, so there is nothing to investigate about non-blocking properties. In contrast to that this it has to be ensured in the third $\Box$-branch that every possible continuation after s is inspected. Therefore also smaller refusals in $R(s)$ have to be selected, so that the possibility to enter the P-branch in process $P \sqcap STOP$ will be provided for the target system.

The internal choice operators ($\sqcap$) used in the definition of $U(n, s)$ show where internal decisions with respect to the control of the test executions may be taken: At each execution step $U(n, s)$ the refusals R or the sets A may be selected according to a test coverage strategy implemented in the test driver. Since there are many possibilities for suitable strategies, these are hidden in the definition of $U(n)$. Any strategy covering all possible executions of $U(n)$ is valid.

Using LTS representations for the CSP specifications of E_I and $ASYS_I$, test $U(n)$ is implementable in a straight forward way: $U(n)$ is determined by the traces and refusals of E_I and $ASYS_I$, and these are contained in the corresponding LTS representations.

Part 1. of Theorem 7 states that the test cases $U(n)$ are complete in the sense that they will always execute successfully when the target system and its abstract specification process $ASYS$ show *identical* behaviour at the system interface. This is only ensured when the abstract specification process $ASYS_I$ of the target system does not diverge in environment E. We do not consider this a severe restriction to the theorem's usability, since explicit incorporation of divergence into abstract specifications occurs very rarely in practice. The implication of part 2., (a) $\Rightarrow$ (b) states the soundness property that successful execution of the tests implies failures-divergence refinement when operating in environment E. Moreover, since the $U(n)$ never diverge by construction, the successful execution ensures that the target system will never diverge in this environment. Conversely, if the refinement relation has been already established and it is ensured that the target system will not diverge in its environment E, we can be certain that all tests $U(n)$ will succeed for SYS_I.

Using the results of Theorem 6 and Theorem 7, now we can state that test drivers using the test cases $U(n)$ have the desired correctness properties:

Theorem 8 *Let $(E, ASYS, SYS, I)$ be a standard configuration of a reactive system, where the associated tests $U(n)$ be defined as above. Then the test driver*

$$\begin{aligned}\mathcal{D}(X) =_{df}\ & (i := 0);\ \ *(U_i \parallel X\hat{\ }(w \rightarrow monitor?next \\ & \rightarrow (\mathbf{if}\ next\ \mathbf{then}\ i := i+1;\ SKIP \\ & \qquad\qquad \mathbf{else}\ SKIP)));\end{aligned}$$

is trustworthy for $\sqsubseteq_{FD}$ -test.
□

8 Conclusion

This article presented a specification, design and verification methodology for dependable systems which can be instantiated with (combinations of) compositional development and verification methods. We have shown how testing safety-critical reactive systems can be regarded as just one specific type of verification technique according to this methodology and therefore be combined with formal verification methods in a consistent way. The article focused on the development of test drivers performing automated generation, execution and evaluation of tests for reactive systems against CSP specifications. Given a correctness relation between abstract specification processes and implementations, a trustworthy test driver should be capable of

- generating test cases for every possible correctness violation,
- exercising test cases on the target system, while simulating at the same time proper environment behaviour,
- detecting every violation of the correctness requirements during test execution.

To obtain test drivers which are *provably correct* with respect to these objectives, we analysed Hennessy's and de Nicola's testing theory in the framework of untimed CSP. Hennessy's test classes are suitable for the detection of safety failures, insufficient requirements coverage, divergence failures and insufficient robustness in an implementation and characterise the corresponding refinement notions. As a result of this analysis we determined minimal subsets of Hennessy's test classes that are capable of detecting safety failures and insufficient requirements coverage. Furthermore we presented the top-level specification of a combined test driver (generator plus driver plus evaluator) as implemented in the VVT-RT system. It was demonstrated that a test driver implementing this specification possesses the three capabilities listed above, with respect to testing safety and requirements coverage.

The work presented in this article reflects a "building block" of a joint enterprise of ELPRO LET GmbH, JP Software-Consulting, Bremen University and Kiel University in the field of test automation for reactive systems. Our objective is to develop a hierarchy of CSP-based test methods, where this article defines the general methodology applicable to each layer in the "stack" of methods and describes the first layer concerned with untimed testing. The higher layers represent the corresponding implementable theories for testing against

- discrete time/discrete values,
- dense time/discrete values,
- dense time/continuous values (i. e.hybrid)

requirements specifications. The complete untimed theory and real-time testing against a restricted version of the CSP timed trace model are already implemented in VVT-RT; further theoretical results will be integrated in future versions of the tool.

Acknowledgements: The first author would like to thank Chris Brink for the invitation to present a series of lectures on testing during the WOFACS '96 workshop. He and his team, the other lecturers (Maarten de Rijke, Jeff Sanders and Holger Schlingloff) and – last but not least! – the audience of the workshop created a stimulating atmosphere and interesting discussions that contributed substantially to the ideas described in this article. Furthermore, the first author has been partially supported by the German Ministry of Education and Research (BMBF), project UniForM.

References

AO91 K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, Berlin Heidelberg New York, 1991.

Bri89 E. Brinksma. A theory for the derivation of tests. In P. H. J. van Eijk, C. A. Vissers, and M. Diaz, editors, *The Formal Description Technique LOTOS*, pages 235–247. Elsevire Science Publishers B. V. (North-Holland), 1989.

Dav93 J. Davies. *Specification and Proof in Real-Time CSP*. Cambridge University Press, 1993.

dI92 Der Bundesminister des Innern. Planung und Durchführung von IT-Vorhaben – Vorgehensmodell. KBSt Koordinierungs- und Beratungsstelle der Bundesregierung für Informationstechnik in der Bundesverwaltung, 1992. English version available from IABG Industrieanlagen Betriebsgesellschaft, Ottobrunn.

DO93 RTCA DO178B. Development considerations in airborne computer systems, 1993.

FDR94 Formal Systems Ltd. Failures Divergence Refinement. User Manual and Tutorial Version 1.4. Formal Systems (Europe) Ltd (1994).

Gau95 M.-C. Gaudel. Testing can be formal, too. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *Proceedings of TAPSOFT '95: Theory and Practice of Software*

Development, Aarhus (Denmark), May 1995. Springer Verlag, 1995.

Gmb94 ELPRO LET GmbH. Programmablaufplan – Bahnübergang, 1994.

Ha90 D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. IEEE Transactions on Software Engineering, (1990) 16: 403-414.

Hen88 M. C. Hennessy. *Algebraic Theory of Processes*. MIT Press, 1988.

Hoa85 C.A.R. Hoare. *Communicating sequential processes*. Prentice-Hall International, Englewood Cliffs NJ, 1985.

Hör95 H. M. Hörcher. Improving software tests using Z specifications. In *ZUM '95: 9th International Conference of Z Users*, LNCS 967. Springer Verlag, 1995.

HP94 H. M. Hörcher and J. Peleska. The role of formal specifications in software test (tutorial). In *Proceedings of FME '94*, 1994.

HP95 H. M. Hörcher and J. Peleska. Using formal specifications to support software testing. *Software Quality Journal* 4, 309 – 327, (1995).

KPO96 B. Krieg-Brückner, J. Peleska, E.-R. Olderog, D. Balzer and A. Baer. Universal Formal Methods Workbench. In U. Grote and G. Wolf, editors, *Statusseminar Softwaretechnologie des BMBF, March 1996, Berlin*, Deutsche Forschungsanstanlt für Luft- und Raumfahrt, Berlin, 1996.

L^+92 J. C. Laprie et al. *Dependability: Basic Concepts and Terminology*. Springer-Verlag, 1992.

Ltd94 Formal Systems (Europe) Ltd. Failures divergence refinement: User manual and tutorial version 1.4, 1994.

Mik95 E. Mikk. Compilation of Z specifications into C for automatic test result evaluation. In J. P. Bowen and M. G. Hinchey, editors, *ZUM '95: 9th International Conference of Z Users*, LNCS 967. Springer, 1995.

Mil89 R. Milner. *Communication and Concurrency*. Prentice-Hall International, Englewood Cliffs NJ, 1989.

Mül94 M. Müllerburg. Systematic testing: a means for validating reactive systems. In *EuroSTAR'94: Proceedings of the 2nd European Intern. Conf. on Software Testing, Analysis&Review*. British Computer Society, 1994.

Mye79 G. J. Myers. *The Art of Software-Testing*. John Wiley & Sons, New York, 1979.

Pel95a J. Peleska. Formal methods and the development of dependable systems. Habilitation Thesis, Kiel University, 1995.

PS96a J. Peleska and M. Siegel. From testing theory to test driver implementation. In *Proceedings of the Formal Methods Europe Conference FME '96*, number 1051 in LNCS. Springer-Verlag, 1996.

Pel96 J. Peleska. Test Automation for Safety-Critical Systems: Industrial Application and Future Developments. In *Proceedings of the Formal Methods Europe Conference FME '96*, number 1051 in LNCS. Springer-Verlag, 1996.

1. J. Peleska: *Trustworthy Tests for Reactive Systems – Automation of Real-Time Testing.* In preparation, JP Software-Consulting (1996).

RAO92 D. J. Richardson, S. Leif Aha, and T. O. O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th International Conference on Software Engineering, Melborne, Australia*, 1992.

ROM89 D. J. Richardson, T. O. O'Malley, and C. Tittle Moore. Approaches to specification-based testing. In *ACM SIGSoft 89: Third Symposium on Software Testing, Analysis and Verification*, 1989.

Sch93 W. Schütz. Fundamental issues in testing distributed real-time systems. Technical Report 170, LAAS-CNRS, Toulouse, PDCS Technical Report Series, 1993.

Sch94 H. Schepers. *Fault Tolerance and Timing of Distributed Systems*. PhD thesis, Eindhoven University of Technology, 1994.

WGS94 E. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from a boolean specification. *IEEE Transactions on Software Engineering*, 20(5), 1994.

Zwi89 J. Zwiers. *Compositionality, Concurrency and Partial Correctness (LNCS 321)*. Springer-Verlag, 1989.

Appendix A Proofs of theorems and lemmas

Proof of Theorem 3.

We prove the case $U_C(s,A) \in \mathcal{H}_C$, the other cases are shown analogously.

Proof Obligation. Show that $\mathit{SPEC}\ \underline{\mathit{must}}\ U_C(s,A)$.

According to the definition of $\mathcal{H}_C$, we have $s \in \mathit{Traces}(\mathit{SPEC}) - \mathit{Div}(\mathit{SPEC})$. Moreover, the definitions of $\mathcal{H}_C$ and $U_C(s,A)$ imply that every execution of $(U_C(s,A) \parallel \mathit{SPEC})$ is a prefix of $s \frown \langle a, w\rangle$, for some $a \in A$.

Case 1. Suppose, an execution of $(U_C(s,A) \parallel \mathit{SPEC})$ does not reach $\mathit{last}(s)$. Since neither $U_C(s,A)$ nor SPEC diverge along s, the same holds for $(U_C(s,A) \parallel$
$\mathit{SPEC})$, therefore SPEC must block before $\mathit{last}(s)$ is reached and, as a consequence, $(U_C(s,A) \parallel \mathit{SPEC})$ is ready to produce any event not contained in $\alpha(\mathit{SPEC})$. Since the only event not contained in $\alpha(\mathit{SPEC})$ and executable in this state is w, this leads to successful execution of $(U_C(s,A) \parallel \mathit{SPEC})$.

Case 2. Suppose $\mathit{last}(s)$ is reached in an execution of $(U_C(s,A) \parallel \mathit{SPEC})$. Since SPEC does not diverge along s, application of the Normal Form Theorem 1 yields

$$\mathit{SPEC}/s = \sqcap_{R:\mathit{Ref}(\mathit{SPEC}/s)}(x : ([\mathit{SPEC}/s]^0 \setminus R) \to \mathit{SPEC}/s \frown \langle x\rangle)$$

Since $A \subseteq [\mathit{SPEC}/s]^0$ and $A \not\subseteq R$ holds for every refusal R of SPEC/s, we get

$$(\forall R : \mathit{Ref}(\mathit{SPEC}/s) \bullet (\exists a : A \bullet a \in ([\mathit{SPEC}/s]^0 \setminus R)))$$

Together with the normal form representation, this shows that SPEC/s cannot refuse all events of A. As a consequence the execution also yields success w. This completes the proof that $\mathit{SPEC}\ \underline{\mathit{must}}\ U_C(s,A)$.
□

Proof of Theorem 4.

Again we only prove the requirements coverage case (2.). Recall that due to the definition of $\sqsubseteq_C$ nothing has to be shown in the case of sequences of events not contained in $\mathit{Traces}(\mathit{SPEC})$. The theorem now follows in case (2.) from the following

Proof Obligation. Show that for all $s \in \mathit{Traces}(\mathit{SPEC})$ and $A \subseteq \alpha(\mathit{SPEC})$:

$$\begin{aligned}&\mathit{SPEC}\ \underline{\mathit{must}}\ U_C(s,A) \wedge \neg\, (\mathit{IMP}\ \underline{\mathit{must}}\ U_C(s,A)) \Rightarrow\\ &\qquad (\exists A' : \mathbb{P}\,\alpha(\mathit{SPEC}) \bullet U_C(s,A') \in \mathcal{H}_C \wedge \neg\, (\mathit{IMP}\ \underline{\mathit{must}}\ U_C(s,A'))))\end{aligned}$$

holds. This means that whenever IMP contains a requirements coverage failure (s,A) with $s \in \mathit{Traces}(\mathit{SPEC})$ and arbitrary A, it will also contain a requirements coverage failure (s,A') which can be detected by a test in $\mathcal{H}_C$.

Assume

$$s \in \mathit{Traces}(\mathit{SPEC}) \wedge \mathit{SPEC}\ \underline{\mathit{must}}\ U_C(s,A) \wedge \qquad U_C(s,A) \notin \mathcal{H}_C \wedge \neg\, (\mathit{IMP}\ \underline{\mathit{must}}\ U_C(s,A))$$

Then, because of the structure of $U(s,A)$, two situations may be the cause for the failure of IMP:

1. $s \in \mathit{Div}(\mathit{IMP})$
2. $s \in \mathit{Traces}(\mathit{IMP}) - \mathit{Div}(\mathit{IMP}) \wedge (\exists R : \mathit{Ref}(\mathit{IMP}/s) \bullet A \subseteq R)$

In both cases we have $s \in \mathit{Traces}(\mathit{IMP})$. Because

$$s \in \mathit{Traces}(\mathit{SPEC}) \wedge \mathit{SPEC}\ \underline{\mathit{must}}\ U_C(s,A)$$

the set $A \cap [\mathit{SPEC}/s]^0$ cannot be completely contained in any of the $R \in \mathit{Ref}(\mathit{SPEC}/s)$. Choose an $A' \subseteq A \cap [\mathit{SPEC}/s]^0$ such that $(\forall R : \mathit{Ref}(\mathit{SPEC}/s) \bullet A' \not\subseteq R)$ and $(\forall X : \mathbb{P}\,A' - \{A'\} \bullet (\exists R : \mathit{Ref}(\mathit{SPEC}/s) \bullet X \subseteq R))$. Then $U(s,A') \in \mathcal{H}_C$ and $\mathit{SPEC}\ \underline{\mathit{must}}\ U_C(s,A')$.

Case 1. If $s \in \mathit{Div}(\mathit{IMP})$ is the cause for $\neg\, (\mathit{IMP}\ \underline{\mathit{must}}\ U_C(s,A))$, then also $\neg\, (\mathit{IMP}\ \underline{\mathit{must}}\ U_C(s,A'))$, because divergence along s will lead to an execution of $(U_C(s,A') \parallel \mathit{IMP})$ where any visible event is refused before success w could be signalled.

Case 2. If $s \in \mathit{Traces}(\mathit{IMP}) - \mathit{Div}(\mathit{IMP}) \wedge (\exists R : \mathit{Ref}(\mathit{IMP}/s) \bullet A \subseteq R)$ is the cause for $\neg\, (\mathit{IMP}\ \underline{\mathit{must}}\ U_C(s,A))$, then also $A' \subseteq A$ will be contained in such an R. This yields once more $\neg\, (\mathit{IMP}\ \underline{\mathit{must}}\ U_C(s,A'))$, because an execution of $(U_C(s,A') \parallel \mathit{IMP})$ will block A'-events after s, and $U_C(s,A')/s$

can signal success w only after having first engaged into some $a \in A'$.
□

Proof of Theorem 5.

Proof of (1). Suppose that $U(s,a) \in \mathcal{H}_S$ and let $\mathcal{H} =_{df} \mathcal{H}_S - \{U_S(s,a)\}$. Define

$$\alpha(P) =_{df} \alpha(\mathit{SPEC})$$

$$\begin{aligned} P/u =_{df} \textbf{ if } u = s \ & \textbf{then } (x : [\mathit{SPEC}/u]^0 \to P/u^\frown\langle x\rangle \, \Box \, a \to \mathit{STOP}) \\ & \textbf{else } (x : [\mathit{SPEC}/u]^0 \to P/u^\frown\langle x\rangle) \end{aligned}$$

Obviously $\mathit{Traces}(P) = \mathit{Traces}(\mathit{SPEC}) \cup \{s^\frown\langle a\rangle\}$. Then $P \, \underline{\mathit{must}} \, U$ holds for all $U \in \mathcal{H}$, but P violates the safety requirement $s^\frown\langle a\rangle \notin \mathit{Traces}(P)$.

Proof of (2). Let $\{U_C(s,A_1),\ldots,U_C(s,A_n)\} \subseteq \mathcal{H}_C$ be an enumeration of the $\underline{\mathit{must}}$-tests in $\mathcal{H}_C$ to be executed for fixed trace s. Suppose $\mathcal{H} = \mathcal{H}_C - \{U_C(s,A_n)\}$. Then define a CSP process P by

$$\alpha(P) =_{df} \alpha(\mathit{SPEC})$$

$$\begin{aligned} P/u =_{df} \textbf{ if } u = s \ & \textbf{then } (x : ([\mathit{SPEC}/u]^0 \setminus A_n) \to P/u^\frown\langle x\rangle) \\ & \qquad \Box (x : A_n \to P/u^\frown\langle x\rangle) \sqcap \mathit{STOP}) \\ & \textbf{else } (x : [\mathit{SPEC}/u]^0 \to P/u^\frown\langle x\rangle) \end{aligned}$$

Since $\mathit{Traces}(P) = \mathit{Traces}(\mathit{SPEC})$ holds by construction, $P \, \underline{\mathit{must}} \, U$ holds for all safety tests in $\mathcal{H}_S$. For traces $u \neq s$, $P \, \underline{\mathit{must}} \, U_C(u,A)$ holds for all tests in $\mathcal{H}_C$, because P/u never refuses an event that might be accepted by SPEC/u. For $u = s$, observe that the sets A_i are minimal in the sense of part (2) of the theorem, and therefore $A_i \setminus A_n \neq \emptyset$ holds for all $i = 1,\ldots,(n-1)$. As a consequence, $[\mathit{SPEC}/u]^0 \setminus A_n$ contains at least one A_i-event for all $i = 1,\ldots,(n-1)$. This implies $P \, \underline{\mathit{must}} \, U_C(s,A_i)$ for $i = 1,\ldots,(n-1)$, so $P \, \underline{\mathit{must}} \, U$ for all tests $U \in \mathcal{H}$. However, P/s may refuse A_n completely, while SPEC/s will always accept at least one A_n-event. As a consequence, P does not refine SPEC in the failures model.

Proof of (3). Suppose $U_C(s,A) \in \mathcal{H}_C$ and $B \subset A$. The definition of $\mathcal{H}_C$ implies the existence of a refusal $R \in \mathit{Ref}(\mathit{SPEC}/s)$ such that $B \subseteq R$. As a consequence there exists an execution $(U(s,B) \parallel \mathit{SPEC})$ blocking and consequently failing after trace s. Therefore, $\neg\,(\mathit{SPEC} \, \underline{\mathit{must}} \, U_C(s,B))$ holds, which proves (3).
□

Proof of Theorem 6.

Since $\alpha(\mathit{SPEC})$ is finite, every $[\mathit{IMP}/s]^0$ is also finite. As a consequence, $\mathit{Traces}(\mathit{IMP})$ contains only a finite number of traces with fixed length $n \in \mathbb{N}$. Since also $\mathit{Ref}(\mathit{IMP}/s)$ is always finite, the number of Hennessy Test $U_i \in \mathcal{H}$ with defining trace of length less or equal n will also be finite.

Since according to Theorem 2 the test cases of $\mathcal{H}$ can detect every type of failure and every failure occurs after a finite trace, the theorem follows.
□

Lemmas for Proof of Theorem 7.

We use the following lemma in the proof:

Lemma 3 *Given* P, Q *with* $\alpha(P) = \alpha(Q)$.

1. $(P \parallel Q) \, \underline{\mathit{must}} \, U_S(s,a) \Leftrightarrow P \, \underline{\mathit{must}} \, U_S(s,a) \vee Q \, \underline{\mathit{must}} \, U_S(s,a)$
2. $(P \parallel Q) \, \underline{\mathit{must}} \, U_C(s,A) \wedge s \in \mathit{Traces}(P \parallel Q) \Rightarrow P \, \underline{\mathit{must}} \, U_C(s,A) \wedge Q \, \underline{\mathit{must}} \, U_C(s,A)$
3. $(P \parallel Q) \, \underline{\mathit{must}} \, U_C(s,A) \Rightarrow P \, \underline{\mathit{must}} \, U_C(s,A) \vee Q \, \underline{\mathit{must}} \, U_C(s,A)$

Proof of Lemma 3.

Proof of (1.)

$(P \parallel Q) \, \underline{\mathit{must}} \, U_S(s,a)$

$\Leftrightarrow s^\frown\langle a\rangle \notin \mathit{Traces}(P \parallel Q) \wedge (P \parallel Q) \, \underline{\mathit{must}} \, U_D(s)$ [Lemma 2]

$\Leftrightarrow (s^\frown\langle a\rangle \notin \mathit{Traces}(P) \vee s^\frown\langle a\rangle \notin \mathit{Traces}(Q)) \wedge P \, \underline{\mathit{must}} \, U_D(s) \wedge Q \, \underline{\mathit{must}} \, U_D(s)$ [$\parallel$-law, $\alpha(P) = \alpha(Q) = I$]

$\Leftrightarrow P \, \underline{\mathit{must}} \, U_S(s,a) \vee Q \, \underline{\mathit{must}} \, U_S(s,a)$ [Lemma 2]

Proof of (2.) Assume $(P \parallel Q) \, \underline{\mathit{must}} \, U_C(s,A) \wedge s \in \mathit{Traces}(P \parallel Q)$, but $\neg\,(P \, \underline{\mathit{must}} \, U_C(s,A))$. There are two possibilities for failure.

Case 1. Assume $s \in \mathit{Div}(P)$. Then, because s is also a trace of Q, the definition of $\parallel$ yields $s \in \mathit{Div}(P \parallel Q)$, which is a contradiction to $(P \parallel Q) \, \underline{\mathit{must}} \, U_C(s,A)$.

Case 2. Assume $(\exists\, R : Ref(P/s) \bullet A \subseteq R)$. Then, because s is also a trace of Q, the definition of $\parallel$ implies that R is also a refusal of $(P \parallel Q)/s$. Once more, this contradicts $(P \parallel Q)\ \underline{must}\ U_C(s, A)$ (Lemma 2). Therefore $P\ \underline{must}\ U_C(s, A)$ holds.

The same argument can be applied for $Q\ \underline{must}\ U_C(s, A)$. This proves (2.).

Proof of (3.) If neither $P\ \underline{must}\ U_C(s, A)$ nor $Q\ \underline{must}\ U_C(s, A)$ holds,

$$\neg\,(P\ \underline{must}\ U_D(s)) \vee \neg\,(Q\ \underline{must}\ U_D(s)) \vee s \in Traces(P) \cap Traces(Q)$$

follows. The first two possibilities both imply $\neg\,((P \parallel Q)\ \underline{must}\ U_D(s))$ which yields $\neg\,((P \parallel Q)\ \underline{must}\ U_C(s, A))$. The last possibility also results in $\neg\,((P \parallel Q)\ \underline{must}\ U_C(s, A))$ because otherwise (2.) would yield a contradiction. This completes the proof of (3.).

□

Proof of Theorem 7.

We will first state and verify four proof obligations that are used to establish the validity of Theorem 7, 2.(a) $\Rightarrow$ 2.(b).

Proof Obligation 1. Show that

$$s \in Traces((E \parallel ASYS)_I) \wedge \#s < n \Rightarrow s \in Traces(U(n)) \wedge U(n)/s = U(n, s)$$

Given $U(n)$ with fixed $n > 0$, we will use induction over the length $m = \#s$ of the traces s, $m = 0, \ldots, n-1$. The assertion holds trivially for $m = 0$, since this implies $s = \langle\,\rangle$ and $U(n)/\langle\,\rangle = U(n) = U(n, \langle\,\rangle)$ by definition. Assume that obligation 1 has been proven for $m \geq 0$ and let $s \in Traces((E \parallel ASYS)_I) \wedge \#s = m + 1 \wedge m + 1 < n$. Define trace $u = front(s)$ and event $a = last(s)$.

Applying the induction hypothesis to u yields $u \in Traces(U(n))$ and $U(n)/u = U(n, u)$. Since $\#s < n$ we have $\#u < n - 1$. Furthermore, $s \in Traces((E \parallel ASYS)_I)$ implies $a \in [(E \parallel ASYS)_I/u]^0$, so $R(u) \neq \emptyset$. Evaluating the guards in the definition of $U(n, s)$ therefore yields

$$\begin{aligned} U(n)/u = U(n, u) = \quad & (A(u) = \emptyset)\&(w \rightarrow SKIP) \\ & \Box \\ & (\#u < n)\&(e : ([E_I/u]^0 \setminus [ASYS_I/u]^0) \rightarrow \dagger \rightarrow SKIP) \\ & \Box \\ & (\sqcap_{\,R : R(u)}\, U(n, u, [(E \parallel ASYS)_I/u]^0 \setminus R)) \end{aligned}$$

$$U(n, u, M) = e : M \rightarrow U(n, u ^\frown \langle e \rangle)$$

Since $\emptyset \in R(u)$, process $U(n, u, [(E \parallel ASYS)_I/u]^0)$ is an alternative of the $(\sqcap_{\,R:R(u)} \ldots)$-branch in $U(n, u)$. As a consequence, $[U(n, u, [(E \parallel ASYS)_I/u]^0)]^0 \subseteq [U(n)/u]^0$ holds. By construction of the processes $U(n, s, M)$ we have $[U(n, u, [(E \parallel ASYS)_I/u]^0)]^0 = [(E \parallel ASYS)_I/u]^0$. Since $a \in [(E \parallel ASYS)_I/u]^0$ we get $a \in [U(n)/u]^0$. This proves $s = u ^\frown \langle a \rangle \in Traces(U(n))$.

Since the first and the second $\Box$-branch of $U(n)/u$ do not accept a, we obtain $U(n)/u ^\frown \langle a \rangle = U(n, u, [(E \parallel ASYS)_I/u]^0 \setminus R)/\langle a \rangle$ for some $R \in R(u)$ with $a \in [(E \parallel ASYS)_I/u]^0 \setminus R$. (Note, that for all such sets R, R' we have by definition of $U(n, s, M)$ that $U(n, u, [(E \parallel ASYS)_I/u]^0 \setminus R)/\langle a \rangle = U(n, u, [(E \parallel ASYS)_I/u]^0 \setminus R')/\langle a \rangle$). Hence, we conclude

$$U(n)/s = U(n)/u ^\frown \langle a \rangle = U(n, u, [(E \parallel ASYS)_I/u]^0 \setminus R)/\langle a \rangle = U(n, u ^\frown \langle a \rangle) = U(n, s)$$

which proves obligation 1.

Proof Obligation 2. For $U_S(s, a) \in \mathcal{H}_S((E \parallel ASYS)_I)$ and $\#s = n - 1$ with $\mathcal{H}_S$ defined as in Definition 4 show that

$$\neg\,((E \parallel SYS)_I\ \underline{must}\ U_S(s, a)) \wedge (E \parallel SYS)_I\ \underline{must}\ U_D(s) \Rightarrow \neg\,(SYS_I\ \underline{must}\ U(n)).$$

By Lemma 3, the premise of Proof Obligation 2 imply $s ^\frown \langle a \rangle \in Traces(E_I)$ and $s ^\frown \langle a \rangle \in Traces(SYS_I)$. Since $U_S(s, a) \in \mathcal{H}_S$, $s \in Traces((E \parallel ASYS)_I) \wedge a \notin [(E \parallel ASYS)_I/s]^0$, so application of Lemma 3 and the definition of $\parallel$ yields $s \in Traces(ASYS_I) \wedge s ^\frown \langle a \rangle \notin Traces(ASYS_I)$. As a consequence, $a \in [E_I/s]^0 \setminus [ASYS_I/s]^0$. From the validity of Proof Obligation 1 we get $U(n)/s = U(n, s)$, so $U(n)/s ^\frown \langle a \rangle = U(n, s)/\langle a \rangle$ will accept the branch $\dagger \rightarrow SKIP$, because $\#s < n$ and $a \in [E_I/s]^0 \setminus [ASYS_I/s]^0$. This means, that at least in one execution $(U(n) \parallel SYS)$ will fail after $s ^\frown \langle a \rangle$, which shows the validity of Proof Obligation 2.

Proof Obligation 3. For $U_C(s, A) \in \mathcal{H}_C((E \parallel ASYS)_I)$ and $\#s = n - 1$ show that

$$\neg\,((E \parallel SYS)_I\ \underline{must}\ U_C(s, A)) \wedge (E \parallel SYS)_I\ \underline{must}\ U_D(s) \Rightarrow \neg\,(SYS_I\ \underline{must}\ U(n))$$

$U_C(s,A) \in \mathcal{H}_C((E \parallel ASYS)_I)$ and part (2.) of Lemma 3 imply $A \in A(s) \wedge s \in \mathit{Traces}((E \parallel ASYS)_I) \wedge E_I \; \underline{must} \; U_C(s,A) \wedge ASYS_I \; \underline{must} \; U_C(s,A)$. If $(E \parallel SYS)_I \; \underline{must} \; U_D(s)$, an execution of $U_C(s,A)$ can only fail for $(E \parallel SYS)_I$ if $s \in \mathit{Traces}((E \parallel SYS)_I)$, so we can assume $s \in \mathit{Traces}(SYS_I)$. Therefore $\neg\,((E \parallel SYS)_I \; \underline{must} \; U_C(s,A))$ implies

$$(\exists\, R : \mathit{maxRef}((E_I \parallel SYS_I)/s) \bullet A \cap [(E_I \parallel SYS_I)/s]^0 \subseteq R).$$

The definition of $\parallel$ implies that such a refusal R is a union of maximum refusals $X_1 \in \mathit{maxRef}(E_I/s)$ and $X_2 \in \mathit{maxRef}(SYS_I/s)$, so $A \cap [E_I/s]^0 \cap [SYS_I/s]^0 \subseteq (X_1 \cup X_2)$. $E_I \; \underline{must} \; U_C(s,A)$ implies $A \setminus X_1 \neq \emptyset$. Moreover, $A \subseteq [E_I/s]^0$ holds according to the definition of $\mathcal{H}_C((E \parallel ASYS)_I)$, so $(A \setminus X_1) \cap [SYS_I/s]^0 \subseteq X_2$. This means that SYS_I/s may refuse every event of $A \setminus X_1$.

From the validity of Proof Obligation 1 we know that $U(n)/s = U(n,s)$. Evaluation of the guards in the definition of $U(n,s)$ results in:

$$\begin{aligned} U(n)/s = U(n,s) = \; & (e : ([E_I/s]^0 \setminus [ASYS_I/s]^0) \to \dagger \to \mathit{SKIP}) \\ & \Box \\ & (\sqcap_{R' : \mathit{maxRef}(E_I/s), A' : A(s)} U(n,s,A' \setminus R')) \end{aligned}$$

$$U(n,s,M) = e : M \to U(n, s \frown \langle e \rangle)$$

Therefore, since $A \in A(s)$ and $X_1 \in \mathit{maxRef}(E_I/s)$, a possible behaviour of $U(n)/s$ is defined by the process

$$\begin{aligned} P = \; & (e : ([E_I/s]^0 \setminus [ASYS_I/s]^0) \to \dagger \to \mathit{SKIP}) \\ & \Box \\ & (e : A \setminus X_1 \to U(n, s \frown \langle e \rangle)). \end{aligned}$$

At least one execution of $((E \parallel SYS)_I/s \parallel P)$ will fail: If SYS_I/s refuses X_2, the second P-branch is blocked, and the first branch leads to trace failure $\dagger$. This proves obligation 3.

Proof Obligation 4. For $U_D(s) \in \mathcal{H}_D((E \parallel ASYS)_I)$ and $\#s = n - 1$ show that

$$\neg\,((E \parallel SYS)_I \; \underline{must} \; U_D(s)) \Rightarrow \neg\,(SYS_I \; \underline{must} \; U(n))$$

The premise $U_D(s) \in \mathcal{H}_D((E \parallel ASYS)_I)$ implies $s \in \mathit{Traces}(E_I \parallel ASYS_I)$ and $s \notin \mathit{Div}(E_I \parallel ASYS_I)$. The semantics of $\parallel$ implies $s \notin \mathit{Div}(E_I) \wedge s \notin \mathit{Div}(ASYS_I)$. Let $u \leq s$ such that $u \in \mathit{Div}((E \parallel SYS)_I)$. Since u is a prefix of s, $u \notin \mathit{Div}(E_I)$ also holds and $u \in \mathit{Div}(SYS_I)$ follows. Since $s \in \mathit{Traces}((E \parallel ASYS)_I)$, the validity of proof obligation 1 shows that $u, s \in \mathit{Traces}(U(n))$. Therefore $u \in \mathit{Traces}(U(n) \parallel SYS_I)$, and the definition of $\parallel$ yields $u \in \mathit{Div}(U(n) \parallel SYS_I)$. As a consequence, $(U(n) \parallel SYS_I)$ may refuse any event of $I \cup \{w\}$ after having engaged into trace u. Since u does not contain w, such an execution will fail. This proves Obligation 4.

Proof of Theorem 7, 2.(a) $\Rightarrow$ 2.(b). Suppose $SYS_I \; \underline{must} \; U(n)$ for all $n \in \mathbb{N}$ and let $U_D(s) \in \mathcal{H}_D((E \parallel ASYS)_I)$. Then $(E \parallel SYS)_I \; \underline{must} \; U_D(s)$, because otherwise $\neg\,(SYS_I \; \underline{must} \; U(\#s+1))$ according to proof obligation 4. Let $U_S(s,a) \in \mathcal{H}_S((E \parallel ASYS)_I)$. Then $(E \parallel SYS)_I \; \underline{must} \; U_S(s,a)$, because otherwise $\neg\,(SYS_I \; \underline{must} \; U(\#s+1))$ according to proof obligation 2, since we already have established that $(E \parallel SYS)_I \; \underline{must} \; U_D(s)$. Let $U_C(s,A) \in \mathcal{H}_C((E \parallel ASYS)_I)$. Then $(E \parallel SYS)_I \; \underline{must} \; U_C(s,A)$, because otherwise $\neg\,(SYS_I \; \underline{must} \; U(\#s+1))$ according to proof obligation 3.

Now we have established that

$$(\forall\, U : \mathcal{H}_S((E \parallel ASYS)_I) \cup \mathcal{H}_C((E \parallel ASYS)_I) \cup \mathcal{H}_D((E \parallel ASYS)_I) \bullet (E \parallel SYS)_I \; \underline{must} \; U)$$

holds. Therefore the application of Theorem 4 results in $(E \parallel ASYS)_I \sqsubseteq_{FD} (E \parallel SYS)_I$. This proves Theorem 7, 2.(a) $\Rightarrow$ 2.(b).

Proof of Theorem 7, 2.(b) $\Rightarrow$ 2.(a). We apply contraposition and prove that the negation of the premise 2.(a) implies the negation of 2.(b). Suppose $\neg\,(SYS_I \; \underline{must} \; U(n))$ for some $n \in \mathbb{N}$. Analysis of the structure of $U(n)$ shows that an execution of $(U(n) \parallel SYS_I)$ can only fail iff at least one of the following conditions are true:

1. An execution diverges, before success could be signalled:
   $(\exists\, s \in \mathit{Traces}(U(n) \parallel SYS_I) \bullet \neg\,\langle w \rangle \; \mathbf{in} \; s \wedge s \in \mathit{Div}(U(n) \parallel SYS_I)$.
2. An execution produces a trace failure:
   $(\exists\, s \in \mathit{Traces}(U(n) \parallel SYS_I) \bullet \#\mathit{front}(s) < n \wedge \mathit{last}(s) \in [E_I/s]^0 \setminus [ASYS_I/s]^0)$.
3. An execution blocks in the third $\Box$-branch of $U(n,s)$:
   $(\exists\, s \in \mathit{Traces}(U(n) \parallel SYS_I);\; R, X : \mathbb{P}\, I \bullet \#s < n - 1 \wedge (s,X) \in \mathit{Fail}(SYS_I) \wedge A(s) \neq \emptyset \wedge R \in R(s) \wedge [(E \parallel ASYS)_I/s]^0 \setminus R \cap [SYS_I/s]^0 \setminus X = \emptyset)$.

4. An execution blocks in the forth $\square$-branch of $U(n,s)$:
$(\exists\, s \in \mathit{Traces}(U(n) \parallel SYS_I);\ A, R, X : \mathbb{P}\, I \bullet \#s = n-1 \wedge (s, X) \in \mathit{Fail}(SYS_I) \wedge A \in A(s) \wedge (s, R) \in \mathit{Fail}(E_I) \wedge A \setminus (R \cup X) = \emptyset)$.

Observe that, due to the specification of $U(n)$, s is also contained in $\mathit{Traces}(E_I)$ in all four cases. Moreover, for cases 3. and 4. s is also contained in $\mathit{Traces}((E \parallel ASYS)_I)$.

Since $\mathit{Div}(U(n)) = \emptyset$ by construction, failure condition 1. implies $s \in \mathit{Div}(SYS_I)$. As a consequence, $\mathit{Traces}(E_I) \cap \mathit{Div}(SYS_I) \neq \emptyset$, so the negation of 2.(b) holds.

For failure condition 2., we have that $\mathit{front}(s) \in \mathit{Traces}((E \parallel ASYS)_I)$, but $\mathit{last}(s) \notin [(E \parallel ASYS)_I / \mathit{front}(s)]^0$. Then the test case $U_S(\mathit{front}(s), \mathit{last}(s))$ is contained in $\mathcal{H}_S((E \parallel ASYS)_I)$, and this test fails for the execution of $(E \parallel SYS)_I$ when trace s is produced. Again this violates 2.(b), because $(E \parallel ASYS)_I \sqsubseteq_{FD} (E \parallel SYS)_I$ implies $(E \parallel SYS)_I$ $\underline{\mathit{must}}$ $U_S(\mathit{front}(s), \mathit{last}(s))$ according to Theorem 4.

For failure condition 3., there exists an $A \in A(s)$ such that $(E \parallel ASYS)_I$ $\underline{\mathit{must}}$ $U_C(s, A)$, so $U_C(s, A)$ is an element of $\mathcal{H}_C((E \parallel ASYS)_I)$. This test will fail for at least one execution of $(U_C(s, A) \parallel (E \parallel SYS)_I)$, since this parallel composition may block completely after s. This implies that $(E \parallel SYS)_I$ does not refine $(E \parallel ASYS)_I$ due to a requirements coverage failure.

For failure condition 4., observe that $(s, R \cup X)$ is a failure of $(E \parallel SYS)_I$, so at least one execution of $(U_C(s, A) \parallel (E \parallel SYS)_I)$ will fail. However, $(E \parallel ASYS)_I$ $\underline{\mathit{must}}$ $U_C(s, A)$ holds since $A \in A(s)$. Therefore $U_C(s, A)$ is contained in $\mathcal{H}_C((E \parallel ASYS)_I)$. Again, this implies that $(E \parallel SYS)_I$ does not refine $(E \parallel ASYS)_I$ due to a requirements coverage failure. Now we have established that the negation of 2.(b) holds for each of the failure conditions listed above. This proves Theorem 7, 2.(b) $\Rightarrow$ 2.(a).

Proof of Theorem 7, 1. Choosing $SYS_I =_{df} ASYS_I$ and noting that $\sqsubseteq_{FD}$ is reflexive, part 1. of the theorem is a trivial consequence of part 2., (b) $\Rightarrow$ (a). This completes the proof of Theorem 7.
□

Proof of Theorem 8.

In analogy to Theorem 6, $\mathcal{D}(X)$ applies test cases ordered by the length of the traces. Since by Theorem 7 $U(n)$ has the same capabilities to detect failures as the Hennessy test cases $U_S(s, a)$, $U_C(s, A)$ with $\#s = n - 1$, Theorem 6 implies that $\mathcal{D}(X)$ is trustworthy for $\sqsubseteq_{FD}$ -test.
□