FTRTFT '98

Testing Reactive Real-Time Systems

Jan Peleska,
(JP Software-Consulting and University of Bremen, Department of Mathematics and Computer Science, jp@informatik.uni-bremen.de)
Peter Amthor,
Sabine Dick,
Oliver Meyer,
(University of Bremen, TZI Center for Computing Technology - Department BISS Bremen Institute of Safe Systems, {amthor,sab,emm}@informatik.uni-bremen.de)
Michael Siegel,
(Institute of Computer Science and Applied Mathematics in the Technical Faculty of Christian-Albrechts-University of Kiel, mis@informatik.uni-kiel.de)
Cornelia Zahlten
(University of Bremen, Department of Mathematics and Computer Science, cmz@informatik.uni-bremen.de)

1. Preface

This contribution to the FTRTFT '98 School focuses on methods for the automated test of (potentially safety-critical) reactive real-time systems. The automation problem is analysed with respect to automated test generation, test execution, test evaluation and test coverage analysis with a focus on automated generation and evaluation. Our presentation will be structured into two parts: Part I introduces the basic notions, objectives and problems to be encountered in the context of testing in general. It is illustrated why different approaches are required for the test of sequential (sub-)systems concerned with data transformations on one hand and reactive real-time systems on the other hand, and our general approach to the trustworthy and yet manageable test of hybrid systems is sketched. Part II presents theoretical results that are relevant for the development of trustworthy test automation tools. These results can be applied to testing against untimed, discrete-time, dense-time/discrete value and dense-time/continuous value (i.e. hybrid) specifications. Our underlying specification framework will be real-time CSP and hybrid CSP variants; readers who are not familiar with CSP might should consult references [6, 13, 27, 30, 33].

The theory and practical concepts described in this report have been implemented in the VVT-RT (Verification, Validation and Test for real-Time Systems) tool which has been developed by JP Software-Consulting in collaboration with the BISS. Until now, VVT-RT has been successfully applied in the following industrial projects:

- Test of a tramway crossing control computer for ELPRO LET GmbH Berlin (1995), see also [20],
- Test of the ABRIXAS Satellite Controller developed by OHB System GmbH Bremen (1997/1998),
- Test of a fail-safe data transceiver for transmission between railway interlocking systems of South African Railways (1997/1998, details will be published in [32]),
- Test of the DMS-R Fault-Tolerant Computer System developed by Daimler-Benz Aerospace Bremen for the International Space Station ISS (1998, see also [23]).

The material presented in this report is the result of a "dedicated group effort": The untimed testing theory presented in Chapter 6 has been developed by Michael and Jan; some very important ideas go back to Ed Brinksma's work on testing against Lotos specifications for OSI conformance tests [3]. The theory based on Timed CSP described in Chapter 7 has been worked out by Jan and Sabine, the hybrid approach described in Chapter 8 is currently investigated by Oliver and Jan with some related work from Peter based on hybrid automata [1]. The main contributions for the development of VVT-RT come from Cornelia, Jan, Oliver and Peter. Related work based on timed transition systems has been developed by Rachel Cardell-Oliver's group at the University of Essex [4].

Contents

Part I

Testing – Basic Concepts and Definitions

2. General Test Configuration and Test Objectives

The typical testing scenario to be investigated in this report is depicted in Figure 2.1. The system to be tested is an embedded controller that will interact with its operational environment through

- digital interfaces and/or
- sensors delivering analogue data (or sampled digital data representing continuous physical values) to the controller and/or
- actuators stimulated by the target system to control analogue devices in the environment.

Our objective is to perform *hardware-in-the-loop tests* to check the behavioural correctness of the controller at its system interface as well as the proper cooperation of the controller hardware and its integrated software components[1]. In most cases, the "real" operational environment will be unsuitable to perform this task, since it will generally not have the capability to stimulate all the test cases in a convenient way (for example, if it requires high physical energy to provide certain input values to the controller) and may suffer severe damage in case of failing test executions (think of a controller for a railway interlocking system). Moreover, the operational environment might even be unavailable at testing time (think of a controller operating in a satellite on a space mission).

As a consequence, the operational environment has to be replaced by a *hardware-in-the-loop test system*. This is a computer or a network of computers that will interact with the *system under test (SUT)* using the same interfaces as the operational environment. These interfaces, however, will not be sufficient to test all relevant aspects of SUT behaviour, as soon as the SUT will behave nondeterministically at its system interface. Nondeterministic behaviour at the system interface will occur quite naturally if the SUT uses internal resource management algorithms (as, for example, a multi tasking scheduler) or if the SUT is composed of several (hardware and/or software) modules whose internal communications cannot be observed from the outside. Therefore it may be necessary to access the SUT via additional *monitoring channels* to inspect "internal SUT decision making" or even to *manipulate* internal SUT states to stimulate specific behaviours that might be too difficult to activate via the system interface (e.g. *fault injections* to test fault-tolerance properties).

In addition to the proper test system it may be necessary to use *environment simulators* modelling specific physical aspects of the operational environment which cannot be imple-

[1]The complete testing theory developed in Part II as well as the test tool described in Part III can also be applied to testing software components – e.g., networks of cooperating processes – running on any computer fulfilling some basic requirements with respect to process communication mechanisms and computing power. However, the typical testing scenario we have in mind are variants of the hardware-in-the-loop test described in this chapter.

mented on the computers of the test system for technical reasons[2]. In such a case the test system will trigger the required behaviours in the environment simulator which will in turn produce input data for the SUT and react to SUT outputs according to the physical laws it simulates.

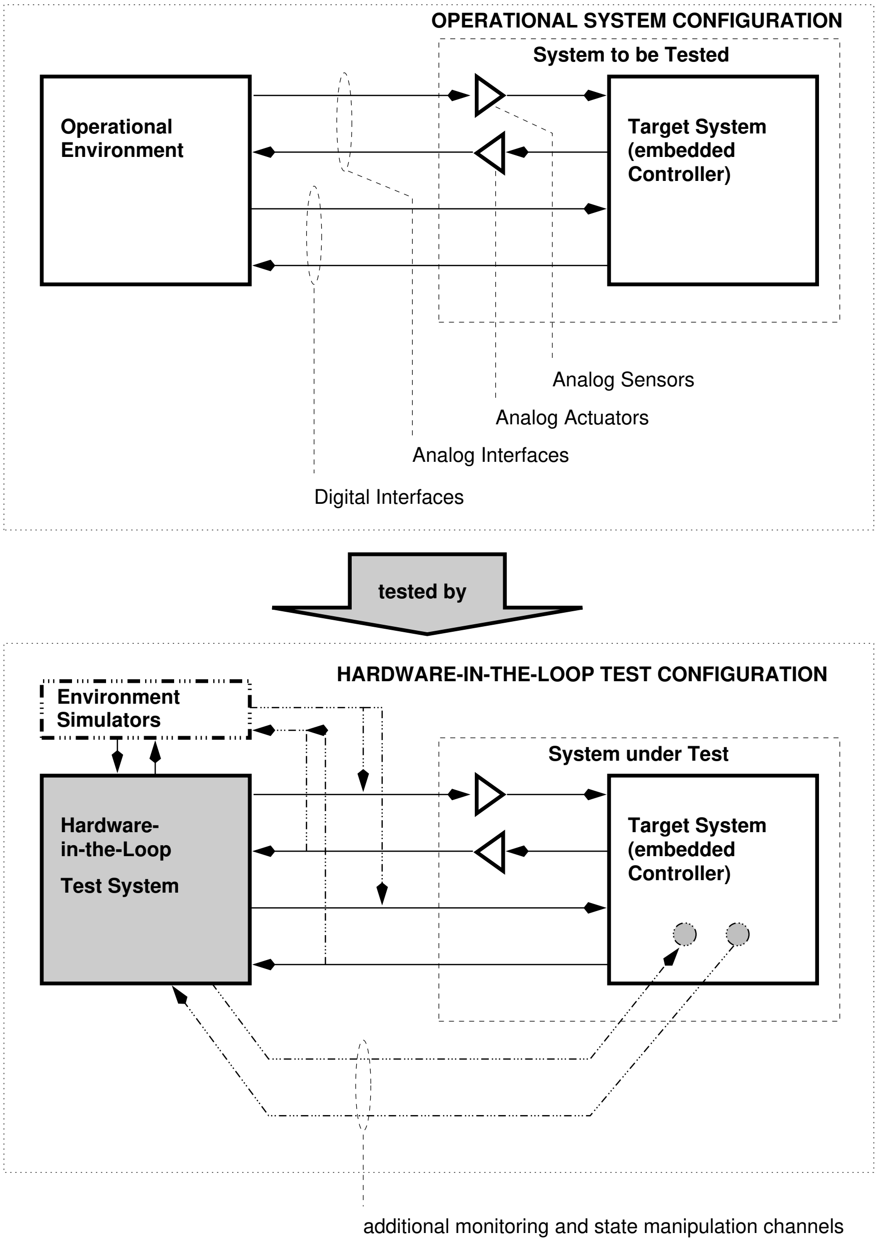

Figure 2.1: General Test Configuration for Hybrid Systems.

The general characterisation of a trustworthy test automation system is informally given as follows:

> Let *SPEC* be the specification of the desired target system behaviour when running in its intended operational environment. Let *IMP* denote the implemen-

[2]For example, BMW Rolls-Royce used original hardware components of an aircraft engine as environment simulators in the test of the engine controller, because the hydro-mechanical behaviour of these hardware components was too complex to be modelled as computer simulations inside the test system.

tation of the target system embedded into the operational environment. Let $\mathcal{R}$ denote a binary relation between specifications and implementations, such that *correctness* of the implementation is characterised by $SPEC\ \mathcal{R}\ IMP$. A test automation system is called *trustworthy* for the implementation relation $\mathcal{R}$ if it satisfies the following conditions:

- *Soundness:* For every implementation IMP' violating the requirement $SPEC\ \mathcal{R}\ IMP'$ the test system will finally generate a test execution with IMP' which fails.
- *Completeness:* For every implementation IMP'' satisfying $SPEC\ \mathcal{R}\ IMP''$ every test execution generated by the test system will succeed.

The mathematical formulation of this statement depends on the formalism used for specification of environment and target system behaviour. In the subsequent chapters we will describe testing theory and test systems which are trustworthy in the above sense for CSP-based specification styles and different variants of CSP refinement relations.

M.-C. Gaudel uses the terms 'validity' and 'unbias' in [10] for these aspects of trustworthiness. We prefer 'soundness' and 'completeness' since this makes the close link between testing and formal verification more explicit: A trustworthy test system can be compared to a proof tool checking claims of the type $SPEC\ \mathcal{R}\ IMP$. The test system should be capable of falsifying every unjustified claim. In the context of proof theories this property is called soundness. On the other hand, the test system should accept every justified claim, which is called completeness in proof theories.

3. Testing Terminology

In this chapter some testing terminology, as presently used in industry is introduced. The definitions have been taken from the international standard DO-178 B [8] for the development of software in airborne systems. *Verbatim* quotations from [8] are displayed as *slanted* text.

The term *Validation* will denote 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 products (specification documents, design documents, code etc.) 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[1].

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 but testing on an abstract level.

A *Test Case* is a *set of test inputs, execution conditions, and expected results developed for a particular objective.* A test case does not necessarily define the complete set of input and output data to be used in an explicit way. Instead, the test case can consist of a specification describing how to construct the explicit *test data*, starting with initial inputs and deriving consecutive data values and expected results during the test execution. The objective of *expected results* is to provide an unambiguous characterisation of correct system behaviour, as it may be observed during an execution of the test case. Since "correct" behaviour is defined by the specification document, the expected results – if developed in the correct way – should be consistent with the specification.

A *Test procedure* is a *detailed instruction for the set-up and execution of a given set of test cases, and instructions for the evaluation of results of executing the test cases.*

[1] 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 chapter, 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.

We will use the term *requirements-driven testing* if the specification serves as the input document to construct the associated test cases. If the test cases are derived from the implementation, e. g. from the software code, the term *implementation-driven testing* will be used. Note that implementation-driven testing also requires a specification document, because otherwise it cannot be decided whether the results are correct.

Tests may be exercised on the complete system as well as on isolated system components. In practice, strategies for a combination of tests on system level and tests on component level are developed, in order to cope with the overall complexity of the test process. The term *Unit Testing* will be used to denote the test of one isolated system component, if no smaller unit of this component will be tested separately. 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. *Integration Testing* applies a strategy of successively adding new components to those already tested and then performing a test of the resulting configuration. *System Testing* denotes the test of the complete target system. For system testing, the environment used for the tests will be the "real" operational context or at least a simulation implemented according to the hypotheses about the operational environment behaviour at the interface level of the target system.

Tests influencing and monitoring only the interface behaviour of the component to be tested will be called *Black-Box Tests*. If internal states or internal execution sequences of the component are also monitored or even influenced, this will be denoted as *White-Box Tests*.

If a component is not tested in the operational environment of the system but within an "artificial" context, the software and hardware implementing this context will be called a *Test Driver*.

As pointed out for example by Hennessy [11], test cases for reactive systems cannot be simply described by means of a pre-state, test inputs and the expected state "after program execution". Since a reactive system interacts continuously with its environment, test cases also have to verify the correct causal relationships between inputs and outputs, the readiness to engage into specific actions, real-time requirements etc. Therefore in the context of reactive systems a *(Black-Box) Test Execution* denotes a *Trace* observable at the interface between the test driver and the component under consideration. In our context a trace is a finite sequence of *Events* (t, c, v). Here c denotes an input- or output interface channel observable by the test environment. We do not impose any restrictions about one-to-one or multi-way communication. Value v is passed over channel c. *Time stamp* t records the point in time, when the communication took place.

A *test case for reactive systems* consists of the following information:

1. *input data and causal execution conditions*: This is a specification referring to the channels of the component to be tested. It describes initial inputs and consecutive inputs depending on the reactions of the target system. The specification may be defined by means of a predicate or by an explicit process characterising the behaviour required of the test driver during test execution.

2. *timed execution conditions*: a specification defining time-dependent conditions when to stimulate a certain input

3. *(timed) expected results*: a specification characterising the traces which are regarded as correct executions of the test case. In the case of deterministic reactive systems

without timing requirements, this specification may explicitly define the sequence of inputs and outputs which is required. However, as soon as timing requirements exist or tolerances for data values are acceptable or the system reaction to a sequence of inputs may be nondeterministic, correct system behaviour can only be specified by more general assertions or processes.

In specific situations, where the systems to be verified can only assume a very limited number of different states and use only restricted ranges for variable values, we may test every possible execution performable by the system. If this complete number of tests is reached, the process will be called *exhaustive testing*.

4. Logical Building Blocks of a Test Automation System

For automated testing, a test system should implement the logical building blocks shown in Figure 4.1[1].

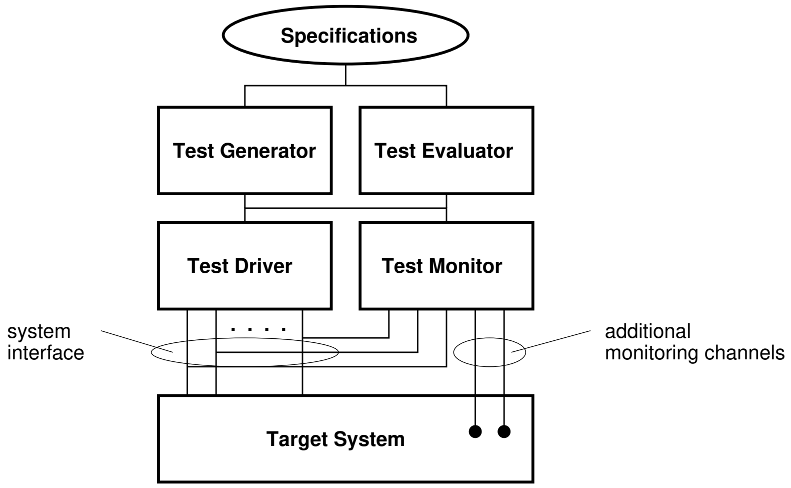

Figure 4.1: Logical building blocks of a test automation system.

Testing will be performed based on Specifications. The different aspects of test automation require different specification types explained with each logical component below.

The Test Generator is responsible for the creation of test cases from specifications. To this end, an *environment specification* E describing the possible behaviours of the operational environment is required: The test system should be capable of exercising every input/output sequence possible in the real environment. Observe, however, that for the test of refinement properties in the failures models of CSP the test generator will also have to interpret *target system specifications* (see Theorem 6 for the formal treatment of this fact): In order to test non-blocking properties of the SUT it is necessary for the test generator to construct minimal sets of events which cannot be completely refused by the SUT. For example, the generator can decide to block all inputs to the SUT if the target system specification guarantees that in the actual state the SUT should be capable of producing outputs without getting additional inputs. The generator is called trustworthy, if for each possible implementation error violating the specified requirements a test case will be created which is capable of detecting this error. As soon as the SUT may legally show nondeterministic behaviour at its interface it is mandatory for the test generator to be capable of *on-the-fly test generation*. This property means that the next test execution step (e. g., the selection of the next input to be sent to the SUT) can be determined in real-time based on the results of the test

[1]Figure 4.1 is not intended to suggest a system architecture for a test system, it only represents logical components that should be present.

execution performed so far. As a consequence the test never has to be aborted due to legal but "unexpected" target system behaviour.

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. In general, the target system is only required to behave according to the specification, as long as the operational environment also behaves correctly. A test driver is therefore called trustworthy, if it exactly simulates the behaviour of the "real" environment during test execution.

The Test Evaluator (also called Test Oracle) verifies the observed test execution against a specification and decides whether the execution was correct. To this end, it has to interpret a target system specification. This specification, say, $SPEC_2$ is not necessarily the same as the specification $SPEC_1$ used for test generation: For example, $SPEC_1$ may be an explicit CSP specification, because the possibility to interpret explicit CSP processes is suitable for test generation. For test evaluation, it may be preferable to use trace assertions in $SPEC_2$. Moreover, it will not always be the case that every behavioural property reflected by $SPEC_1$ will also be checked by $SPEC_2$. Conversely, it may be the case that $SPEC_1$ is an incomplete description of the required behaviour, which is just suitable for the generation of certain test cases, while $SPEC_2$ is a stronger specification, valid for the evaluation of other classes of test cases as well. For the test evaluator trustworthiness means that, given the results of a test execution, it will be detect every violation of the specified requirements. For the test of reactive systems it is necessary for the test evaluator to have the capability of *on-the-fly test evaluation*. This means that (at least a sub-set of) the correctness properties specified for the SUT can be checked in real time before the next execution step is chosen: After erroneous SUT behaviour it may be impossible for the test generator to find a valid execution step to continue the test (think of a situation where the environment specification is only prepared to react on SUT outputs from a set $\{x_1, \ldots, x_n\}$, but the target system produces output x_0). Moreover, it may be necessary to record additional analysis data which would otherwise not be added to the test execution log. Finally, after certain types of erroneous target system behaviour it might be futile to continue the test. This is of special importance for long-term testing of safety-critical real-time systems where automated tests typically run for days or even months.

The Test Monitor observes each test execution in order to decide whether

- a specific test case has been performed for all relevant executions that are possible for this case, when exercised on the target system,
- the full set of test cases executed so far suffices for the required level of test coverage.

In many applications, these tasks of a test monitor cannot be performed based on the black-box observations of the system interface alone. Instead, additional monitoring and state manipulation channels must be created to provide internal state information about the target system for the monitor and to "force" the target system to select specific execution branches that cannot be controlled at the regular system interface.

5. Testing Hybrid Systems – the Global Picture

Testing hybrid systems requires a combination of very different techniques for test generation, execution, evaluation and monitoring since such systems are usually constructed from components of the following types:

Sequential components execute data transformation algorithms without any interleaved communication or synchronisation with other components. Timing requirements only exist with respect to the upper bound of CPU time that may be used by the algorithms. Typically, the correctness of sequential component behaviour is specified by relations between initial and termination states. Test generation based on such relations uses the classical techniques for module testing (see [14, 18, 28]). Test Evaluation is performed by checking the correctness relation by substituting its free variables with the concrete test data selected and obtained in the test execution (see, for example, [15]). For test monitoring, different test coverage measures (statement coverage, branch coverage, path coverage etc.) have been defined and the relation between the coverage achieved and the probability of remaining errors in the code is well understood.

Untimed concurrent components consist of networks of parallel processes that may communicate and synchronise with each other. In addition to the correctness of data exchanged between these components, correctness requirements refer to synchronisation and sequencing of events and can be expressed as refinement relations or assertions about traces and refusals. Test generation and evaluation methods based on untimed CSP are described in detail in Chapter 6 and have also been published in [19, 20, 21, 22]. Test monitoring is now more complex than for sequential components, since in most applications the processes will be specified as non-terminating entities. The implemented system *IMP* may have many more states than its associated specification *SPEC*, therefore it will not suffice to cover every cyclic execution path possible according to *SPEC* just once in a test. Instead, it will be necessary to run several times through cyclic execution paths in *SPEC* until one can be sure that all different internal states of *IMP* have been reached in the tests. For finite state machine-based formalisms, this problem is well understood, and several monitoring algorithms deciding about "sufficient" test coverage have been developed. Their basic concept is to exploit knowledge about the maximum number of internal states in the implementation and to develop criteria to decide how many different implementation states have been reached in the test executions so far (see [5, 28]). While it may be infeasible for complex computer systems to get estimates about the maximum number of internal states, the approach can be practically applied for specialised controllers with low numbers of states as, for example, PLCs.

Discrete-time concurrent components must comply with additional correctness properties referring to the proper timing of events. Time can be modelled as a global counter representing clock ticks (see also [27, pp. 383]), and timing assertions refer to the counter values associated with each event. As we will describe in Chapter 7, the test generation and evaluation problem can be solved with the same methods developed for the untimed concurrent case. Observe that the notion of discrete time is of high practical value, since it can often be implemented in a natural way by sampling input data with a fixed frequency or communicating by means of frame protocols allowing input/output only at fixed multiples of a basic frame-time. In fact most hard real-time systems implement a discrete-time model.

Observe that for untimed and discrete-time concurrent programs only discrete values can be passed over interfaces, since data can only be received or transmitted at discrete points in time. As a consequence, analogue data to be passed over system interfaces cannot be captured by these models.

Dense-time/discrete value concurrent components use time models based on a dense set but still only use digital interfaces for discrete data. The dense time/discrete value model is useful if the relative time distances between events cannot be specified by multiples of a "shortest" interval but should be modelled as being taken from intervals over real numbers. Observe that such a model will also be implemented in a discrete-time component, since every digital interface has a minimum latency between two consecutive I/Os. However, this latency can be so short that the dense-time model really is more appropriate for the system than a discrete-time model with an extremely short minimum time interval between clock ticks. The theoretical results obtained in Chapter 7 hold both for the discrete and dense-time models, but their practical applicability is considerably reduced when the admissible system behaviours really admit very large numbers of different duration intervals. With respect to monitoring, the results about untimed finite state-machines sketched above could be extended by R. Cardell-Oliver and T. Glover [4] to timed transition systems. The authors indicate that their approach should also be applicable to timed specifications formalisms with synchronous communication such as Timed CSP.

Dense-time/continuous value concurrent components handle analogue data to be modelled as piecewise continuous (or better piecewise analytic, see [24]) functions over time, using analogue sensors and actuators as output interfaces. Again, sensors and actuators may be digital components in some applications, but their sampling rate or the granularity of actuator values should be so high that the continuous model represents a valid approximation of reality. For dense-time/continuous value components test generation involves the execution of continuous functions over time, sometimes implicitly specified by differential equations. Test evaluation requires numerical analysis of analogue data recorded by the test system from the actuator outputs of the SUT. A theory investigating the trustworthiness of test generation and evaluation strategies has to combine dense-time/discrete value testing theory with results obtained from mathematical calculus and numerical analysis. A heuristic approach for the test of dense-time/continuous value concurrent components that is presently investigated in our group with respect to its theoretical foundations is described in Chapter 8.

The above classification indicates that a testing effort – as well as a formal verification effort! – may become extremely difficult to manage if components of different type are "glued together" arbitrarily in the system design. It is therefore our objective to work out well-organised design patterns indicating the "proper places" for each component type in the architectural system decomposition. The design pattern should guarantee that the different test or verification methods – each one best suited for a specific component type – can be consistently combined on different levels of the architectural decomposition.

A promising design pattern currently investigated in our group is shown in Figure 5.1: We restrict ourselves to hybrid systems that may be specified using variants of *Hybrid CSP* [33]: $\mathcal{HCSP}$ uses discrete-value interfaces between concurrent components but allows the specification of time-continuous changes for local process variables. Standard operators like $\|$, $\Box$ and $\overset{u}{\triangleright}$ are adopted from timed CSP [6, 29, 30, 31], but the $\mathcal{HCSP}$ semantics is defined by a translation of $\mathcal{HCSP}$ terms into duration calculus. This concept does not offer the full expressive power of duration calculus but has the advantage of providing CSP operators suitable to specify system structure.

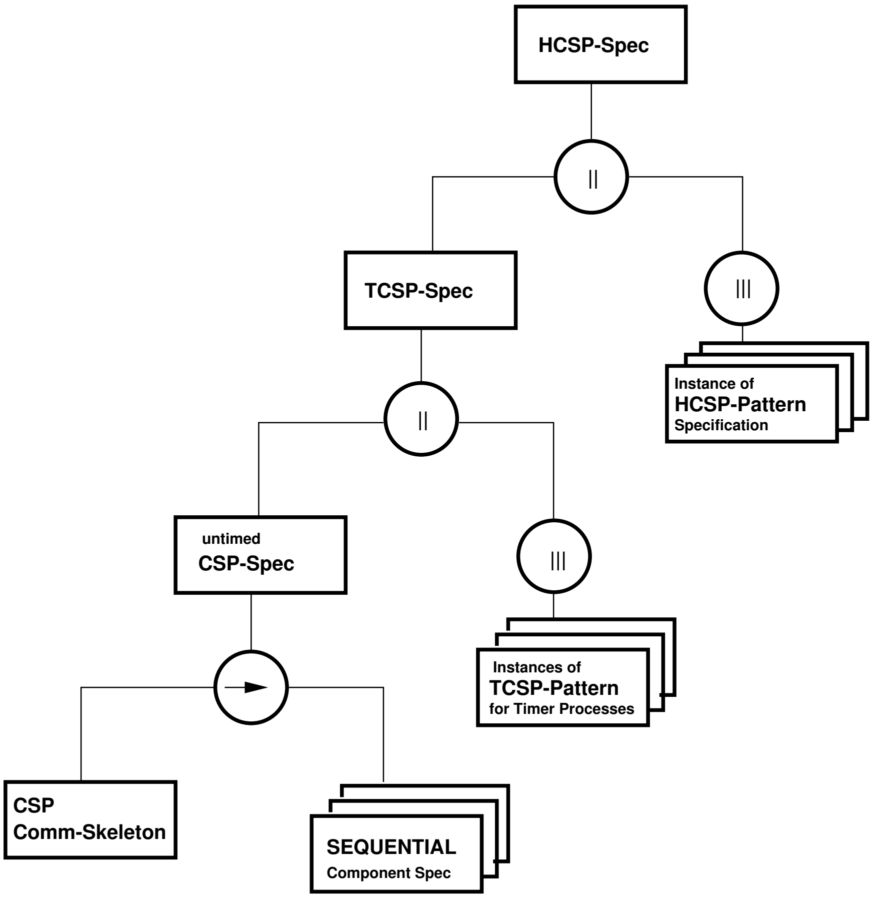

Figure 5.1: Structural decomposition of hybrid system specification.

Suppose a hybrid system has been specified by a $\mathcal{HCSP}$ specification H. Our conjecture is that large classes of specifications H can be decomposed into semantically equivalent specifications $(PRT \,\|\, (\,|||_{\,i} PH_i))$ such that

- PRT is a timed discrete-value component expressible in Timed CSP,
- each PH_i is a $\mathcal{HCSP}$ process containing time-continuous expressions but following a simple pattern structure, such that only a very restricted subset of $\mathcal{HCSP}$ syntax is used to specify the components PH_i,

- each hybrid component PH_i only interacts with PRT via synchronous CSP communication channels.

In Chapter 8 we give examples of the hybrid patterns presently handled in our test system VVT-RT.

The encapsulation of time-continuous expressions in specification components of very restricted syntactical patterns facilitates the development of tests checking the hybrid aspects of a system: The proper manipulation of time-continuous variables can be analysed by means of tests derived from the simplified PH_i specification patterns. The correctness aspects of communication between system processes and their environment can be handled “inside the Timed CSP world” where time-continuous values do not exist. The situation is further simplified if the system architecture follows the same decomposition approach: If the implementation is structured into communicating processes implementing PRT and PH_i the correctness of the overall system can be derived from *component tests* separately analysing the PRT and PH_i implementations and *compositional reasoning* concluding that correct implementation of PRT and PH_i implies correct implementation of $(PRT \parallel (\, |||_i PH_i))$ which in turn is equivalent to the original requirements specification H.

Turning to the problem of testing against Timed CSP (TCSP) specifications PRT, an analogous structural decomposition theorem is already available and will be introduced in Chapter 7: Each PRT using a slightly restricted TCSP syntax can be decomposed into a semantically equivalent specification $UC \parallel (\, |||_i Timer_i)) \setminus TE$ where UC may be expressed using untimed CSP syntax only and each $Timer_i$ is an instance of a process pattern encapsulating the timed syntactical constructs $WAIT\ t$ and $\stackrel{u}{\triangleright}$. Again, the processes $Timer_i$ do not interact with each other, but communicate only with UC using a set of auxiliary *timer events* from a set TE introduced in the transformation process from PRT to $UC \parallel (\, |||_i Timer_i)) \setminus TE$. As a result of this theorem we can lift the results obtained for testing against untimed CSP specifications (this is described in Chapter 6) to TCSP only by augmenting the untimed theory by considering timed refusals of the “elapsed-timer” events used in the processes $Timer_i$.

Since a theory for testing against untimed CSP specifications is also available, we could stop the decomposition at this place. However, it may be desirable for practical reasons to separate the test of sequential data transformations from the test of behavioural aspects such as sequencing of events, synchronisation and non-blocking properties: Tests of behavioural aspects will be based on transition graph representations of the associated untimed CSP specifications. These transition graphs will become unmanageable if their labels have to run through wide ranges of data values and if their states have to encode the values of large internal process data structures. Fortunately, in a system design carefully encapsulating data transformations in sequential functions, such a separation is possible. Recall that sequential CSP processes can be represented in *normal form* (see [2])

$$
\begin{array}{l}
P ::= S_0\,; \\
\qquad \textbf{while}\ (\bigvee_{(c,i):\Gamma_{IN}\cup\Gamma_{OUT}} B_{c,i})\ \textbf{do} \\
\qquad\quad \Box_{(c,i):\Gamma_{IN}} (B_{c,i} \& c?x_{c,i} \rightarrow S_{c,i}\,; SKIP) \\
\qquad\quad \Box \\
\qquad\quad \Box_{(c,i):\Gamma_{OUT}} (B_{c,i} \& c!x_{c,i} \rightarrow S_{c,i}\,; SKIP) \\
\qquad\ \textbf{od} \\
\qquad SKIP
\end{array}
$$

Here the $S_{c,i}$ are sequential components operating on local process variables. Communication only takes place at the beginning of the **while** -loop, and it is guarded by boolean expressions $B_{c,i}$ over local variables. The identifiers $\Gamma_{IN/OUT}$ denote the sets of input and output channels, respectively. Now if the $S_{c,i}$ are implemented as local functions and have been sufficiently tested as separate units or have been formally verified we can abstract from the actual data and use *function call identifiers* instead. These identifiers just record which functions have been called between two consecutive communications. The process network can now be tested against CSP specifications just identifying the correctness of

- sequences of channels used during the communications,
- sequences of function calls applied by each sequential process.

If we do not trust previous unit tests or formal verifications but wish to test that the sequential operations operate correctly on the SUT hardware, we just have to log the actual data produced during test execution and evaluate it *a posteriori*. At least this saves us from having to perform on-the-fly test evaluation of data transformation results during test executions. The approach to test sequencing of events and data transformations in a separate way has been further examined in [28]. The author also considers test approaches where such a separation is impossible due to a lack of design structure encapsulating data transformations in local functions.

Part II

Testing Theory

6. Untimed Testing Theory

6.1 Motivation and Conceptual Background

The objective of this section is to investigate methods for the automated tests of systems against specifications expressible in *untimed CSP* as defined by Hoare [13] and Roscoe [27]. The basic idea for the automation process is:

- Automated test generation is performed by mechanised analysis of the specification, which results in a choice of traces and acceptance sets to be exercised as test cases on the target system.
- Automated test evaluation is performed by observing traces and readiness of the target system and checking mechanically, whether this behaviour is correct according to the specification.

Obviously, these tasks are fundamentally connected to the problem of mechanised *simulation* of the specification. How to find an implementable solution for this problem, is indicated by the following well-known theorem (see, e. g., [11, p. 94]):

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

1. *If $\langle\,\rangle \notin Div(P)$, then*

$$P = \sqcap_{R:Ref(P)}(x : (P^0 \setminus R) \rightarrow P/\langle x \rangle)$$

2. *If $Div(P) = \emptyset$, then $P/s = P(s)$ with*

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

3. *For arbitrary P, $P \sqsubseteq_{FD} P(\langle\,\rangle)$ holds.*

□

Here the notation $[P/s]^0 = \{e : \alpha(P) \mid (\exists\, u \in Traces(P/s) \bullet head(u) = e)\}$ has been used and P/s denotes the process P after having performed a trace s.

Interpreting this result shows how *trustworthy simulators* – i. e., simulators really implementing what is expressed by the specification – could be constructed: At each step P/s of a symbolic execution, choose a valid refusal set R at random and engage into any one of the remaining events $e \in [P/s]^0 \setminus R$. After some e has been chosen, continue in state $P/s \frown \langle e \rangle$. Given an implementation of a simulator, the problem of test generation 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 initial problem to solve is how to retrieve the semantic representation – i. e., the failures (traces plus refusals) and divergences – of a specification written in CSP syntax. This has been solved by Formal Systems Ltd and implemented in the FDR system [9], for the subset of CSP specifications satisfying

- The complete CSP 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 may be modelled by means of a finite number of internal 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 $\rightarrow$, $\sqcap$, $\square$, $;$, $X = F(X)$. The *composite process language* uses the operators $\parallel$, $|||$, $\frown$, $\setminus$, f^* to construct full systems out of lower-level processes.

Under these conditions the CSP specification may be represented as a *Labelled Transition System (LTS)* [16] which may be encoded by means of 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 [11] 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 P of the process. 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, and the uniquely determined node reached by the walk s represents the equivalence class of process state P/s. The labels of the edges leaving P/s define the set $[P/s]^0$ of events that may possibly occur in state P/s. In the *failures model* of CSP a state P/s is characterised by $[P/s]^0$, together with the set of associated refusals $Ref(P/s)$. Since the alphabet is finite and the collection of refusals is subset-closed, $Ref(P/s)$ is uniquely determined by the set $refMax(P/s)$ of *maximum refusals*: $Ref(P/s) = \{X : \mathbb{P}\,\alpha(P) \mid (\exists R : refMax(P/s) \bullet X \subseteq R)\}$.

The LTS representation of P by means of a *finite* transition graph is possible, because any CSP process constructed by communicating finite-state sequential processes using the composite process language is also finite-state. Therefore, though P may be a non-terminating process, only a finite number of processes P/s and P/u will be distinct, and an infinite acceptance tree representation of P may be encoded as a finite graph by identifying the nodes P/s and P/u, if the corresponding processes are identical in the failures model. So, 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.

Encoded in this way, the transition graph therefore allows to recover the complete failures set of a process P, consisting of all pairs (s, X), where s is a trace of P and X a refusal of P/s. Together with the alphabet, this is the representation of P in the (denotational) failures semantics. The possibility of *divergence* for a process $Q = P \setminus H$ may be detected by finding a cycle in the transition graph of P using only events contained in H.

Formal Systems' FDR tool compiles CSP specifications into their corresponding transition graph. As a consequence, any evaluation of the specification semantics may be implemented by means of programs analysing this finite graph.

The problem of automatic test evaluation is now rather trivial: 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 "trustworthy" test generation is much more complex: Theoretically, the transition graph defines completely the acceptable behaviour of the implementation and – by taking traces and refusals *not* occurring in the graph – the incorrect behaviour of an implementation. For non-terminating systems, this involves an infinite variety of possible executions. Therefore the problem how to find "relevant" test cases and how to decide whether "a sufficient amount" of test executions has been exercised on the target system has to be carefully investigated.

6.2 CSP, Refinement and the Relation to Testing Methodology

In this section some well-known properties about the relation between testing and refinement will be summarised and re-phrased for the context of this chapter. Intuitively speaking, a testing procedure should be called "trustworthy", if the associated collection of test cases "converges" to some kind of "correctness proof" with each additional successful test execution. In order to define these informal notions in a precise way, at first we have to decide about an appropriate definition of "correctness".

6.2.1 Correctness Definitions for Implementations

Since we are concerned with the test of an implementation *IMP* against its specification *SPEC*, it is natural to choose a relation between *SPEC* and *IMP* to define the correctness of the implementation. We assume that both *SPEC* and *IMP* may be expressed as untimed CSP processes. In this case the intuitive understanding of implementation correctness in the context of safety-critical systems may be formally expressed by means of different types of CSP *refinement relations*. To this end, it will be required that $\alpha(SPEC) = \alpha(IMP)$, so specification and implementation are both restricted to the set of events that are "relevant" for the decision of implementation correctness. This does not mean, that specification and implementation have to be completely constructed on the basis of the same set of events: In general $SPEC = X \setminus (\alpha(X) - I)$ and $IMP = Y \setminus (\alpha(Y) - I)$, where X and Y contain specification and implementation details using different alphabets. However, we assume that the correctness of *IMP* with respect to *SPEC* may determined by observing I-events only. This will be investigated in the context of tests for *reactive systems* in Section 6.3.2.

The relation between intuitive correctness notions and their formal CSP equivalents can be described as follows:

1. **Safety:** The implementation should only perform executions pre-planned in the specification. This is can be expressed by demanding that the trace space of the implementation should be contained in that of the specification, i. e.

   $$Traces(IMP) \subseteq Traces(SPEC)$$

2. **Requirements Coverage:** After having performed an initial execution s which is possible both for specification and implementation, the implementation should never

refuse a service which is not refused by specification. Formally speaking,

$$\forall s : Traces(SPEC) \cap Traces(IMP) \bullet Ref(IMP/s) \subseteq Ref(SPEC/s)$$

Since $\langle\ \rangle \in Traces(SPEC) \cap Traces(IMP)$, this implies that a trace which can never be refused by $SPEC$ will also be performed by IMP. However, if $Traces(SPEC)$ contains $s \frown \langle e \rangle$ but may refuse e after s, it is not guaranteed that the implementation will be able to perform $s \frown \langle e \rangle$: Sufficient requirements coverage allows to refuse a service in the implementation *completely* if it *may* be refused according to the specification.

3. **Non-Divergence:** Apart from blocking a service, non-availability may be caused by performing an infinite sequence of internal events invisible to the outside world, or by showing completely unpredictable system behaviour. This is denoted by *divergence*, and the implementation should diverge – if at all – only after traces where the specification also diverges.

$$Div(IMP) \subseteq Div(SPEC)$$

4. **Robustness:** If it is required that *every* trace pre-planned in the specification should also be performable in the implementation, we need

$$Traces(SPEC) \subseteq Traces(IMP)$$

This definition of robustness, introduced in [3], 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 nondeterministic exceptional behaviours described by the specification process are also covered by the implementation.

Example 6.2.1 Let $SPEC = a \to b \to STOP \sqcap c \to d \to STOP$ Then $IMP_1 = a \to STOP$ is a safe implementation of $SPEC$, but does not provide requirements coverage. $IMP_2 = a \to b \to STOP \mathbin{[]} e \to STOP$ provides requirements coverage but is neither robust nor safe. $IMP_3 = a \to b \to STOP \mathbin{[]} c \to d \to STOP$ is safe and provides requirements coverage and robustness.
$\square$

These correctness properties will now be related to the notions of *trace (T)-refinement* $\sqsubseteq_T$, *failures (F)-refinement* $\sqsubseteq_F$, *failures-divergence (FD)-refinement* $\sqsubseteq_{FD}$ and *failures-divergence-robustness (FDR)-refinement* $\sqsubseteq_R$. Recall that (see [13, 9, 7])

1. $SPEC \sqsubseteq_T IMP \equiv Traces(IMP) \subseteq Traces(SPEC)$
2. $SPEC \sqsubseteq_F IMP \equiv Fail(IMP) \subseteq Fail(SPEC)$
3. $SPEC \sqsubseteq_{FD} IMP \equiv SPEC \sqsubseteq_F IMP \wedge Div(IMP) \subseteq Div(SPEC)$
4. $SPEC \sqsubseteq_R IMP \equiv SPEC \sqsubseteq_{FD} IMP \wedge IMP \sqsubseteq_T SPEC$

The following table describes the correspondence between the correctness properties introduced above and these refinement notions ('•' means that the property is implied by the refinement relation):

Property	$\sqsubseteq_T$	$\sqsubseteq_F$	$\sqsubseteq_{FD}$	$\sqsubseteq_R$
safety	•	•	•	•
requirements coverage		•	•	•
non-divergence			•	•
robustness				•

Apart from this obvious correspondence to the intuitive correctness notions, there is another reason to focus on these refinement concepts when investigating methods for test automation: As described in the next section, Hennessy's testing methodology introduces types of test cases uncovering exactly those implementation failures which violate one of the above refinement relations. As a consequence, the compositional proof theory associated with the refinement concepts may be consistently used in parallel with the testing process: Critical components may be formally verified to fulfill the refinement relation desired. At the same time tests may be performed for less critical components, and these tests may be regarded as "approximations" of refinement proofs or "incomplete formal verifications". In specific cases to be investigated during the next sections, it is even possible to perform *exhaustive testing* with a finite number of test executions, so that the test activities represent a valid substitute for the corresponding refinement proof.

In some applications one might prefer to specify the correct behaviour of an implementation by means of an assertion $\mathit{IMP} \overset{!}{\textbf{sat}} S(h, R)$ where $S(h, R)$ is a predicate with trace h and refusal R as free variables. However, this concept of correctness is – at least theoretically – covered by our definition: If predicate $S(h, R)$ can be satisfied by a CSP process at all, then there exists a most nondeterministic process *SPEC* such that $\mathit{SPEC}\,\textbf{sat}\,S(h, R)$ [26]. As a consequence, verifying whether *IMP* satisfies the assertion can be viewed as the proof obligation $\mathit{SPEC} \sqsubseteq_{FD} \mathit{IMP}$. We prefer to work with the $\sqsubseteq$ correctness concept only, because in order to allow automatic generation of test cases in the VVT-RT system, *SPEC* will have to be represented as an explicit CSP process.

6.2.2 Hennessy's Testing Methodology

As shown by Hennessy and de Nicola [11, 7], trace *(T)*-, failures *(F)*-, failures-divergence *(FD)*- and failures-divergence-robustness *(FDR)*-refinement are closely related to testing. The results summarised below hold in an even wider context as it is needed for the purpose of this chapter, see [12]. Extending Hennessy's original concepts, several terms and definitions to be used for the classification of tests in the subsequent sections are introduced. These definitions are not essential if the testing methodology is applied only for the characterisation of process algebra semantics, but they are useful for the practical applications of the testing methodology in the context of reactive systems.

Hennessy introduced *experimenters* as processes U with $\alpha(\mathit{SPEC}) = \alpha(\mathit{IMP}) \subset \alpha(U)$ and a specific event $w \in \alpha(U) - \alpha(\mathit{SPEC})$ denoting successful execution of the experiment which consists of U running in parallel with a process possessing alphabet $\alpha(\mathit{SPEC})$. Experimenters coincide with our notion of *Test Cases*, so we will only use the latter term. As pointed out in Chapter 3 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 $s \frown \langle w \rangle \in \mathit{Traces}(P \parallel U)$. Depending on the specification of U and P, three cases may be distinguished with respect to the outcome of test executions:

1. There exists at least one successful execution of $(P \parallel U)$.
2. Every execution of $(P \parallel U)$ must lead to success.
3. No execution of $(P \parallel U)$ is successful.

This observation leads to the following formal definition:

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

1. $P \; \underline{may} \; U \equiv (\exists \, s : \mathit{Traces}(P) \bullet s \frown \langle w \rangle \in \mathit{Traces}(P \parallel U))$
2. $P \; \underline{must} \; U \equiv (\exists \, Q \bullet (P \parallel U) \setminus (\alpha(U) - \{w\}) = w \rightarrow Q)$

□

The definition of $P \; \underline{must} \; U$ requires some explanation: If every execution of $(P \parallel U)$ leads to success, this is equivalent to the following conditions:

- $(P \parallel U)$ may never block before having produced a success indication w.
- $(P \parallel U)$ may never engage into an unbounded sequence of events not containing w.

After success has been indicated, we do not care about the further behaviour of $(P \parallel U)$. This situation is precisely reflected by $(P \parallel U) \setminus (\alpha(U) - \{w\}) = w \rightarrow Q$, where "=" denotes equivalence in the failures-divergence model. This algebraic definition is equivalent to Hennessy's definition [12] of *must*-tests, requiring that every *computation* of $(P \parallel U)$ should indicate success.

Note that we cannot demand that test cases should indicate *failure* in addition to success, because the failure may materialise as a situation where the test execution is blocked or diverges, so that it can only be "communicated" by *refusing to indicate success*. This situation is simplified when studying real-time testing for safety-critical systems: Here expected events should always occur within a bounded amount of time, so deadlock or livelock can be detected by means of timeouts.

If a test execution refuses to indicate success, this information will only be of practical value, if it can be determined what actually went wrong. The correctness notions defined by *T,F,FD,FDR*-refinement allow a precise classification of the failures in an implementation *IMP* when compared with its specification *SPEC*. This classification leads to the following definition:

Definition 2 *Let U be a test case.*

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

□

For practical reasons it is useful to restrict the class of admissible test cases: If the execution is successful we want to be sure to be informed about this "at once". Moreover, the indication of success – if possible – should not allow an unbounded trace of other events to be executed before the "success situation" arises. Since test cases will be executed by test drivers which must know when a test execution has been successfully completed, we require a well-defined termination event after having signalled success, and it is not necessary to indicate success more than once. Finally, we restrict ourselves to test cases which are successful as $\underline{must}$-tests for at least one process. This leads to the following definition:

Definition 3 *An* admissible test case *for the test against SPEC is a CSP process U satisfying*

1. $\alpha(U) = \alpha(\mathit{SPEC}) \cup \{w, \checkmark\}$, $w \notin \alpha(\mathit{SPEC})$
2. $U \,\mathbf{sat}\, S_U(s, R)$ *with*

$$\begin{aligned} &S_U(s, R) \equiv \\ &\qquad w \in [U/s]^0 \implies w \notin R \wedge \#s \leq n \wedge \neg\, (\langle w \rangle \;\mathbf{in}\; s) \wedge U/s^\frown\langle w \rangle = \mathit{SKIP} \end{aligned}$$

   where $n \in \mathbb{N}$ is a constant not depending on s or R.
3. *There exists a process P such that $P \; \underline{must} \; U$.*

□

The following examples illustrate the intuition standing behind the above definition by presenting test cases that are *not* admissible.

Example 6.2.2 The test case $U = a \to \mathit{SKIP} \;\square\; b \to (w \to \mathit{SKIP} \sqcap U)$ would not be admissible in the sense of Definition 3, because it is uncertain whether success will be indicated after event b.
□

Example 6.2.3 The test case $U = a \to w \to \mathit{SKIP} \sqcap \mathit{STOP}$ would not be admissible in the sense of Definition 3, because no process can satisfy U as a $\underline{must}$-test.
□

Example 6.2.4 The test case

$$\begin{aligned} &U = \sqcap_{n:\mathbb{N}} U(n) \\ &U(n) = (n > 0) \& a \to U(n-1) \;\square\; (n = 0) \& w \to \mathit{SKIP} \end{aligned}$$

would be well-defined in the infinite traces model of Roscoe and Barret [25], and $P \; \underline{must} \; U$ holds for process $P = a \to P$. Moreover, if success w is possible after U/s it will never be refused. However, U would not be admissible in the sense of Definition 3, because no global upper bound exists after which every execution of $(P \parallel U)$ would show success.
□

Hennessy defines four types of test cases which are essential for the verification of *T,F,FD,FDR*-refinement properties:

Definition 4 *The class of* Hennessy Test Cases *is defined by the following collection of admissible test cases:*

1. *For* $s \in \alpha(SPEC)^*, a \in \alpha(SPEC)$, *define* Safety Tests $U_S(s,a)$ *by*

$$
\begin{aligned}
U_S(s,a) = \textbf{if } & s = \langle\ \rangle \\
\textbf{then } & (w \to SKIP \;\square\; a \to SKIP) \\
\textbf{else } & (w \to SKIP \;\square\; (head(s) \to U_S(tail(s),a))
\end{aligned}
$$

2. *For* $s \in \alpha(SPEC)^*, A \subseteq \alpha(SPEC)$, *define* Requirements Coverage Tests $U_C(s,A)$ *by*

$$
\begin{aligned}
U_C(s,A) = \textbf{if } & s = \langle\ \rangle \\
\textbf{then } & (a : A \to w \to SKIP) \\
\textbf{else } & (w \to SKIP \;\square\; head(s) \to U_C(tail(s),A))
\end{aligned}
$$

3. *For* $s \in \alpha(SPEC)^*$, *define* Divergence Tests $U_D(s)$ *by*

$$
\begin{aligned}
U_D(s) = \textbf{if } & s = \langle\ \rangle \\
\textbf{then } & w \to SKIP \\
\textbf{else } & (w \to SKIP \;\square\; head(s) \to U_D(tail(s)))
\end{aligned}
$$

4. *For a sequence of events* $s \in \alpha(SPEC)^*$, *define* Robustness Tests $U_R(s)$ *by*

$$
\begin{aligned}
U_R(s) = \textbf{if } & s = \langle\ \rangle \\
\textbf{then } & w \to SKIP \\
\textbf{else } & head(s) \to U_R(tail(s))
\end{aligned}
$$

□

Definition 4 is motivated by the following lemma:

Lemma 2 *In the sense of Definition 2, the Hennessy Test Cases detect the following failures:*

1. $U_S(s,a)$ *detects safety failure* $s ^\frown \langle a \rangle$.
2. $U_C(s,A)$ *detects requirements 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 $s ^\frown \langle a \rangle \notin Traces(P)$ 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 $s \in Div(P)$ means $P/s = CHAOS$ in the sense of [13], 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 $U_C(s,A)$ 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 $s \notin Div(P)$, because divergence along s would imply that every continuation of

s, specifically $s^\frown\langle a\rangle$ would be a trace of P. $P\ \underline{must}\ U_C(s,A)$ implies $s \notin Div(P)$, because divergence along s implies the possibility to refuse every subset of $\alpha(P)$ after s. These observations are summarised in the following lemma:

Lemma 3 *Let P a CSP process, $A \subseteq \alpha(P)$ with $A \neq \emptyset$, $a \in \alpha(P)$, $s \in \alpha(P)^*$. Then*

1. $P\ \underline{must}\ U_S(s,a)$ *iff* $s^\frown\langle a\rangle \notin Traces(P)$.
2. $P\ \underline{must}\ U_C(s,A)$ *iff* $(s,A) \notin Fail(P)$.
3. $P\ \underline{must}\ U_D(s)$ *iff* $s \notin Div(P)$.
4. $P\ \underline{may}\ U_R(s)$ *iff* $s \in Traces(P)$.
5. $P\ \underline{must}\ U_S(s,a)$ *implies* $P\ \underline{must}\ U_D(s)$.
6. $P\ \underline{must}\ U_C(s,A)$ *implies* $P\ \underline{must}\ U_D(s)$.

$\square$

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

Theorem 1 *The classes $U_S(s,a)$, $U_C(s,A)$, $U_D(s)$, $U_R(s)$ of test cases are related to T, F, FD, FDR-refinement as follows:*

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

Proof.
The theorem follows directly from Lemma 3 and the definition of T, F, FD, FDR-refinement.
$\square$

The implications in this theorem become equivalences if additionally $Div(IMP) \subseteq Div(SPEC)$ holds. Theorem 1 shows that "in principle" only requirements-driven test design is needed: It is only necessary to execute test cases that will succeed for $SPEC$. On the other hand, the properties covered by Theorem 1 cannot be verified by means of black-box tests alone, because they require the analysis of *every* possible execution. If IMP is nondeterministic, it may possibly refuse in every test execution to engage into the trace leading to the detection of an error which could be uncovered by the test cases $U_S(s,a)$, $U_C(s,A)$, $U_D(s)$. Conversely, for the test of robustness, the refusal to engage into trace s in a test execution does not prove that $s \notin Traces(IMP)$. Therefore in general, a *test monitor* collecting internal IMP-information about the executions performed will be mandatory to determine whether "sufficient" test executions have been performed. Note that this is no disadvantage of the defined classes of tests but inherent in every testing approach that is sensitive to nondeterminism.

6.2.3 Trustworthy Collections of Test Cases

When performing a test suite to investigate the correctness properties of a system, a crucial objective is to exercise "as few test cases as possible". Therefore the main result (Theorem 2) of this section is devoted to the characterisation of subsets of Hennessy test cases which still suffice to establish the refinement properties implied by Theorem 1. Theorem 3 shows that the sets defined for $\sqsubseteq_T$- and $\sqsubseteq_F$-test in Theorem 2 are even minimal. For non-terminating processes, minimal sets of test cases investigating divergence and robustness properties cannot be defined. However, Theorem 2 shows that they can at least be chosen considerably smaller than the collection used in Theorem 1.

Before presenting the theorems, two additional definitions for the classification of test cases are introduced:

- *Meaningful* collections of test cases against a given specification *SPEC* are those which succeed when executed in parallel with *SPEC* itself. Since in most applications exhaustive testing will be impossible, it would be misleading if test cases U not fulfilled by *SPEC* were executed, because they would add nothing to the question of implementation correctness while giving users the impression that – in case of successful execution – "some progress" had been made.
- *Trustworthy* collections of test cases have the additional property to establish a refinement relationship between *SPEC* and its implementation *IMP*, when exercised completely and successfully on *IMP*. Conversely speaking, trustworthy collections contain for each implementation error a test case which is able to uncover this error.

Definition 5 *A collection $\mathcal{U}$ of admissible test cases is called*

1. meaningful for the $\underline{may}$-test against $SPEC$, *iff*

$$(\forall\, U : \mathcal{U} \bullet s^\frown\langle w\rangle \in \mathit{Traces}(U) \implies s \in \mathit{Traces}(\mathit{SPEC}))$$

2. meaningful for the $\underline{must}$-test against $SPEC$, *iff it is meaningful for the* $\underline{may}$ *-test and*

$$(\forall\, U : \mathcal{U} \bullet \mathit{SPEC}\ \underline{must}\ U)$$

□

Observe that *"meaningful for the* $\underline{may}$ *-test against SPEC"* requires more than just $(\forall\, u : \mathcal{U} \bullet \mathit{SPEC}\ \underline{may}\ U)$: The tests of a meaningful collection may only show success after executions that can also be performed by $SPEC$.

Example 6.2.5 Let

$$\mathit{SPEC} = a \rightarrow \mathit{STOP}$$
$$\mathit{IMP} = b \rightarrow \mathit{STOP}$$
$$U = a \rightarrow w \rightarrow \mathit{SKIP} \,\square\, b \rightarrow w \rightarrow \mathit{SKIP}$$

Then $\mathit{SPEC}\ \underline{must}\ U$, but U is not meaningful for the test against $SPEC$, since it also signals success when exercised on implementations *IMP* violating the safety requirement

$SPEC \sqsubseteq_T IMP$.
□

The notion of *Trustworthiness* is now defined by

Definition 6 *Let $\mathcal{U}$ be a set of admissible test cases for the test against SPEC with $\mathcal{U} = \mathcal{U}_T \cup \mathcal{U}_F \cup \mathcal{U}_{FD} \cup \mathcal{U}_R$, such that $\mathcal{U}_T, \mathcal{U}_F, \mathcal{U}_{FD}$ contain meaningful $\underline{must}$ -tests and $\mathcal{U}_R$ contains meaningful $\underline{may}$ -tests.*

1. *$\mathcal{U}$ is called* trustworthy for $\sqsubseteq_T$-tests *iff*

$$(\forall\, U : \mathcal{U}_T \bullet IMP\ \underline{must}\ U) \implies SPEC \sqsubseteq_T IMP$$

2. *$\mathcal{U}$ is called* trustworthy for $\sqsubseteq_F$-tests *iff it is trustworthy for $\sqsubseteq_T$-tests and*

$$(\forall\, U : \mathcal{U}_F \bullet IMP\ \underline{must}\ U) \implies SPEC \sqsubseteq_F IMP$$

3. *$\mathcal{U}$ is called* trustworthy for $\sqsubseteq_{FD}$-tests *iff it is trustworthy for $\sqsubseteq_F$-tests and*

$$(\forall\, U : \mathcal{U}_{FD} \bullet IMP\ \underline{must}\ U) \implies SPEC \sqsubseteq_{FD} IMP$$

4. *$\mathcal{U}$ is called* trustworthy for $\sqsubseteq_R$-tests *iff it is trustworthy for $\sqsubseteq_{FD}$-tests and*

$$(\forall\, U : \mathcal{U}_R \bullet IMP\ \underline{may}\ U) \implies SPEC \sqsubseteq_R IMP$$

□

Definition 5 defines a considerable reduction of the test classes, when compared to the preconditions of Theorem 1: In the theorem, the Hennessy Test Cases were selected for *every* sequence $s \in \alpha(SPEC)^*$ and *every* set $A \subseteq \alpha(SPEC)$ of events. However, the following theorem characterises the subset of Hennessy Test Cases introduced above which is "relevant" for the verification of refinement properties. It shows that the substantial restrictions of the test cases to be executed according to Definition 5 are really sensible. A more detailed interpretation of the theorem is given at the end of this section.

Theorem 2 *For a given specification SPEC, define collections of test cases by*

1. $\mathcal{H}_S(SPEC) = \{U_S(s,a) \mid s \in Traces(SPEC) - Div(SPEC) \wedge a \notin [SPEC/s]^0\}$
2. $$\begin{aligned}\mathcal{H}_C(SPEC) = \{U_C(s,A) \mid\ & s \in Traces(SPEC) - Div(SPEC) \wedge \\ & A \subseteq [SPEC/s]^0 \wedge \\ & (\forall\, R : Ref(SPEC/s) \bullet A \not\subseteq R) \wedge \\ & (\forall\, X : \mathbb{P}\, A - \{A\} \bullet (\exists\, R : Ref(SPEC/s) \bullet X \subseteq R))\}\end{aligned}$$
3. $$\begin{aligned}\mathcal{H}_D(SPEC) = \{U_D(s) \mid\ & s \in Traces(SPEC) - Div(SPEC) \wedge \\ & (\forall\, u : Traces(SPEC) - Div(SPEC) \bullet \\ & \quad s \leq u \wedge [SPEC/u]^0 = \emptyset \implies s = u)\}\end{aligned}$$
4. $$\begin{aligned}\mathcal{H}_R(SPEC) = \{U_R(s) \mid\ & s \in Traces(SPEC) \wedge \\ & (\forall\, u : Traces(SPEC) \bullet \\ & \quad s \leq u \wedge [SPEC/u]^0 = \emptyset \implies s = u)\}\end{aligned}$$

Then

1. $\mathcal{H}_S(SPEC)$ *is trustworthy for* $\sqsubseteq_T$ *-tests.*
2. $\mathcal{H}_S(SPEC) \cup \mathcal{H}_C(SPEC)$ *is trustworthy for* $\sqsubseteq_F$ *-tests.*
3. $\mathcal{H}_S(SPEC) \cup \mathcal{H}_C(SPEC) \cup \mathcal{H}_D(SPEC)$ *is trustworthy for* $\sqsubseteq_{FD}$ *-tests.*
4. $\mathcal{H}_S(SPEC) \cup \mathcal{H}_C(SPEC) \cup \mathcal{H}_D(SPEC) \cup \mathcal{H}_R(SPEC)$ *is trustworthy for* $\sqsubseteq_R$ *-tests.*

□

The following Theorem shows that the collections $\mathcal{H}_S$ and $\mathcal{H}_C$ are minimal:

Theorem 3 *The collections* $\mathcal{H}_S, \mathcal{H}_C$ *of test cases defined in Theorem 2 are minimal in the following sense:*

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* $U_C(s, A) \in \mathcal{H}_C$ *and* $B \subset A$ *then* $\neg\,(SPEC\ \underline{must}\ U_C(s, B))$.
3. *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.*

□

Interpretation of Theorem 2 and Theorem 3 Let us first state the obvious: For terminating systems, refinement properties can be verified by means of a finite number of tests, because the trace space is finite and we always assume finite alphabets, hence the sets $\mathcal{H}_S, \mathcal{H}_C, \mathcal{H}_D, \mathcal{H}_R$ will also be finite.

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[1], since in such a case $SPEC/s$ will allow chaotic behaviour which does not restrict the admissible behaviour of IMP/s. 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 unnecessary to exercise any tests $U_C(s, A)$, if $SPEC/s$ may refuse the full alphabet, e. g., if $SPEC/s = P \sqcap STOP$. For implementation purposes it is useful to observe that since $Ref(SPEC/s)$ is always finite and subset-closed, we only need to evaluate the *maximal* refusals $R \in refMax(SPEC/s)$ to compute the sets A satisfying $(\forall R : Ref(SPEC/s) \bullet A \not\subseteq R)$.

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 *maximal* 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.

Observe that the collections of tests introduced in Theorem 2 completely characterise the refinement relations between $SPEC$ and IMP: Any IMP refining $SPEC$ must satisfy all corresponding Hennessy tests U_S, U_C, U_D, U_R, and the collections $\mathcal{H}_S, \mathcal{H}_C, \mathcal{H}_D, \mathcal{H}_R$ are subsets of these Hennessy tests.

[1] Of course, it is questionable if specifications allowing divergence will be used in practice at all.

Theorem 3 shows that $\mathcal{H}_S$ and $\mathcal{H}_C$ are indeed minimal sets for T- and F-tests against $SPEC$: If one test $U_S(s, a)$ 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. For each $U_C(s, A)$ in $\mathcal{H}_C$, the set A cannot be reduced, otherwise the test would no longer succeed for $SPEC$. Moreover, 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. $\mathcal{H}_D$ and $\mathcal{H}_R$, however, cannot be defined as minimal sets, as soon as $SPEC$ describes a non-terminating system: If $comp = \langle s(1), s(2), s(3), \ldots\rangle$ 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 $comp$, and each infinite subset of these tests would suffice to verify correct behaviour along $comp$. At least we can state that any $\mathcal{H}_D^0 \subseteq \mathcal{H}_D$ satisfying

$$(\forall\, u : Traces(SPEC) - Div(SPEC) \bullet \exists\, s : \mathcal{H}_D^0 \bullet u \leq s)$$

is sufficient to detect divergence failures against $SPEC$ and any $\mathcal{H}_R^0 \subseteq \mathcal{H}_R$ satisfying

$$(\forall\, u : Traces(SPEC) - Div(SPEC) \bullet \exists\, s : \mathcal{H}_R^0 \bullet u \leq s)$$

is sufficient to detect robustness failures.

6.3 Trustworthy Test Drivers

6.3.1 The Concept of Test Drivers

Test Drivers are hardware and software devices controlling the executions of test cases for a target system. To formalise this notion, recall that a *context* is a CSP term $\mathcal{C}(X)$ with one free identifier X, so every context can be regarded as a continuous function mapping CSP processes X to CSP processes $\mathcal{C}(X)$ (see [17, p.160]). Apart from the free identifier X, $\mathcal{C}(X)$ may contain other CSP processes as parameters.

Definition 7 *A* Test Driver *for the test against $SPEC$ is a context $\mathcal{D}(X)$ using admissible test cases U_i satisfying $\alpha(SPEC) = \alpha(U_i) \setminus \{w\}$ as parameters.*
□

Example 6.3.6 Every admissible test case U can be regarded as a test driver $X \mapsto (U \parallel X)$ executing only a single test case.
□

Example 6.3.7 We will focus on test drivers of the form

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

with admissible test cases U_i. A test driver of this type will execute the test cases in a certain ordering $U_1, U_2, \ldots$; 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. If the monitor signals $next = true$, the next test case U_{i+1} will be performed, otherwise U_i will be repreated. If U_i is a $\underline{may}$-test, $next$ is always set to $true$. The problem how to determine test coverage can be separated from the

task of test generation, execution and evaluation. In principle, the monitor could also be a person, manually interacting with the test driver.
$\Box$

The minimum requirement for a *trustworthy* test driver is given by the following definition:

Definition 8 *A test driver $\mathcal{D}(X)$ is* meaningful for the test against *SPEC, iff the test cases U_i appearing as parameters of $\mathcal{D}(X)$ are taken from a collection $\mathcal{U}$ of meaningful $\underline{may}$ - and $\underline{must}$ -tests against SPEC in the sense of Definition 5.*
$\Box$

Definition 8 requires that the test driver should only perform executions of test cases which may succeed for *SPEC*. This will ensure that each successful test execution performed by $\mathcal{D}(IMP)$ will at least indicate that the implementation "can perform something which is correct according to the specification". However, this minimum requirement does not ensure that any kind of "progress" is made during the test: $\mathcal{D}(IMP)$ might perform always the same test execution, so that an additional success indication would not contribute any new insight about the implementation. Therefore we introduce

Definition 9 *Let $\mathcal{D}(X)$ be a meaningful 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:*

- *$\mathcal{U}$ contains a subset $\mathcal{U}_\sqsubseteq$ which is trustworthy for $\sqsubseteq$-test against SPEC according to Definition 6.*
- *For every safety-, requirements coverage-, divergence- or robustness-failure violating $\sqsubseteq$, there exists an $n \in \mathbb{N}$ such that $U_n \in \mathcal{U}_\sqsubseteq$ detects this failure in the sense of Definition 2.*

$\Box$

Definition 9 covers the intuitive understanding of trustworthiness in a formal way: whenever *IMP* may perform a failure, this can be uncovered by a test case which is guaranteed to be chosen by the driver after having selected a finite number of other test cases.

Example 6.3.8 A test driver performing Hennessy Test Cases $U_x(s, \ldots)$ according to a depth-first ordering of the traces s is *not* trustworthy in the sense of Definition 9: For recursive implementations the test case selection may follow an infinite sequence of events, so that an infinite number of test cases may precede a test case which could detect a failure by following a "neighbouring" trace.
$\Box$

Theorem 4 *Every test driver*

$$\mathcal{D}(X) = (i := 0);\ *(U_i \,\|\, X\,\hat{}\,(w \rightarrow monitor?next \rightarrow (\textbf{if}\ next\ \textbf{then}\ i := i + 1;\ SKIP\ \textbf{else}\ SKIP)));$$

applying the Hennessy Tests

$$U_i \in \mathcal{H}_S(SPEC) \cup \mathcal{H}_C(SPEC) \cup \mathcal{H}_D(SPEC) \cup \mathcal{H}_R(SPEC)$$

according to Theorem 2, ordered by the length of the defining traces, is trustworthy for $\sqsubseteq_R$-test against SPEC. $\Box$

Analogous results hold for $\sqsubseteq_T$, $\sqsubseteq_F$, $\sqsubseteq_{FD}$ -test.

6.3.2 Test Drivers for Reactive Systems

The testing methodology presented so far in this chapter will now be specialised on the development of test drivers for the automated test of *Reactive Systems*.

Separation of Environment and Target System In the context of reactive systems it is useful to distinguish between the target system and its operational environment in an explicit way, when investigating properties of a specification *SPEC* and implementation *IMP*: The very purpose of reactive systems is to interact continuously with their environment. In many applications certain hypotheses are made about the environment behaviour. This means that the target system is not expected to act properly in *every* context. Indeed, the objective of the test suite is to *ensure the correct behaviour of the target system when running in an operational environment satisfying these hypotheses.* Therefore test drivers should

- *simulate* the environment behaviour
- *test* the target system behaviour in the simulated environment

To formalise the notion of an operational environment we will consider expressions of the type

$$SPEC = \mathcal{E}(ASYS) \setminus (\alpha(\mathcal{E}(ASYS)) - I)$$

with the following interpretation: $\mathcal{E}(X)$ is a context and $ASYS$ is the abstract specification of the target system to be developed. The processes appearing as parameters in $\mathcal{E}$ represent the operational environment. The correctness of a reactive system implementation will only be decided with respect to a subset I of observables considered as relevant from the point of view of the application. Therefore the specification consists of $\mathcal{E}(ASYS)$ with all events apart from I concealed. The implementation will be denoted by

$$IMP = \mathcal{E}(SYS) \setminus (\alpha(\mathcal{E}(SYS)) - I)$$

where SYS is the target system "plugged into" environment $\mathcal{E}$. It is natural to demand that $I \subseteq \alpha(\mathcal{E}(ASYS)) \cap \alpha(\mathcal{E}(SYS))$.

Note that the alphabets of the abstract specification $ASYS$ and the implementation SYS of the target system may be distinct, though – as required according to the preceding sections – $\alpha(SPEC) = \alpha(IMP)$. Of course, the verification obligations for *IMP* with respect to *SPEC* remain as introduced above, using *T,F,FD,FDR*-refinement.

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

Definition 10 *A* standard configuration $(E, ASYS, SYS, I)$ (for reactive systems) *consists of CSP processes* $E, ASYS, SYS$ *and a set of events* I *such that*

$$I = \alpha(E) \cap \alpha(ASYS) = \alpha(E) \cap \alpha(SYS)$$

For a given standard configuration, the context $\mathcal{E}_0(X) = (E \parallel X)$ *is called the* environment.

$$SPEC = \mathcal{E}_0(ASYS) \setminus (\alpha(\mathcal{E}_0(ASYS)) - I)$$

is called the specification*,*

$$IMP = \mathcal{E}_0(SYS) \setminus (\alpha(\mathcal{E}_0(SYS)) - I)$$

the implementation*. For* $\sqsubseteq \in \{\sqsubseteq_T, \sqsubseteq_F, \sqsubseteq_{FD}, \sqsubseteq_R\}$*, a standard configuration is called* $\sqsubseteq$-correct*, if* $SPEC \sqsubseteq IMP$ *holds.*
□

Note that it is not a severe restriction to require that the set I of observables should be identical to the set of events shared between environment and target system: Since in the context of reactive systems the tester or any other observer is a part of the environment, any observable event generated by the target system must be shared with the environment. Moreover, if we wish to observe environment events e during test execution that do not occur at the interface to the target system, but "inside" the environment, the standard configuration $(E, ASYS, SYS, I)$ can be re-arranged, such that the new target system is extended by auxiliary system components RUN_e, running in interleaving mode with the "real" target system and sharing e with the environment without ever refusing its occurrence: The new configuration would be defined by

$$(E', ASYS', SYS', I') = (E, (ASYS \mid\mid\mid RUN_e), (SYS \mid\mid\mid RUN_e), I \cup \{e\})$$

We will use the abbreviation $P_I = P \setminus (\alpha(P) \setminus I)$. 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 [13, p.112].

A Characterisation of Meaningful Test Drivers for Reactive Systems The following theorem characterises the trace behaviour of tests used by meaningful test drivers for reactive systems. An interpretation of its intuitive meaning is presented after its proof.

Theorem 5 *Let* $\mathcal{D}(X)$ *be a test driver for the test against SPEC in a standard configuration* $(E, ASYS, SYS, I)$ *of reactive systems. Define*

$$D_0 = D_0(\langle\ \rangle)$$

$$D_0(s) = \sqcap_{R:Ref(E_I/s)}(x : ([E_I/s]^0 \setminus R) \to D_0(s,x))$$

$$\begin{aligned} D_0(s,x) = {} & \textbf{if } x \in [ASYS_I/s]^0 \\ & \quad \textbf{then } (D_0(s \frown \langle x \rangle) \sqcap w \to SKIP) \\ & \quad \textbf{else } E/s \frown \langle x \rangle \end{aligned}$$

Then

1. *If* $\mathcal{D}(X)$ *is meaningful for the test against SPEC in the sense of Definition 8, then every test case* U *appearing as a parameter of* $\mathcal{D}(X)$ *satisfies* $D_0 \sqsubseteq_T (U \parallel E_I)$.
2. *If* $D_0 \sqsubseteq_T (U \parallel E_I)$*,* U *is admissible and* $s \frown \langle w \rangle \in Traces(U)$ *implies* $s \in Traces(E_I)$ *for every test case* U*, then* $\mathcal{D}(X)$ *ist meaningful for the* may*-test against SPEC.*

□

Interpretation of Theorem 5 Since any test case U in the reactive system configuration runs in parallel with E_I, it is ensured that only traces of E_I may be performed. Therefore there is no need for D_0 to admit other traces than those contained in $Traces(E_I)$. A meaningful test case will only indicate success, if the trace performed so far is also a valid trace according to the specification $ASYS$ of the target system. The length of the trace performed before the indication of success depends on the test case U. Due to these initial considerations, process D_0 is designed as a simulation of the environment E as far as it is relevant for the investigation of correctness properties of the target system, i. e., E projected onto the events contained in I. D_0 performs a trace s of I-events, such that s is also a trace of E_I. At each execution step, D_0 chooses a refusal of E_I/s at random and then engages into any event that would not be refused by E_I in this situation. If the next event x is also valid according to the specification of the target system, i. e., $x \in [ASYS_I/s]^0$, D_0 will nondeterministically either continue with the next execution step or indicate success w. If x is not contained in $[ASYS_I/s]^0$, this represents a safety failure. After having indicated success, D_0 does not produce further events, in accordance with the fact that admissible test cases terminates after w. After a failure D_0 will admit any trace that might be performed by E_I. This is necessary to allow the test cases U compared against D_0 to terminate at an arbitrary (finite) trace after failure. Observe that D_0 itself is *not* an admissible test case, since an occurrence of w may be refused. As a consequence, the refinement relation $D_0 \sqsubseteq_T (U \parallel E_I)$ assumed for the second part of the theorem does not allow to conclude the admissibility of the test case U.

A Trustworthy $\sqsubseteq_{FD}$-Test Driver for Reactive Systems After the general characterisation of trustworthy test drivers presented above, our objective is now to introduce a specific test driver with the following properties:

- The test driver is trustworthy for $\sqsubseteq_{FD}$-test, against associated standard configurations $(E, ASYS, SYS, I)$.
- The test driver is implementable in a "straightforward" way, if the specifications of the abstract target system $ASYS$ and of the environment process E are available in the transition system representation described in 6.1.
- The test driver simulates the environment process E, so that only the target system SYS must be provided for a test execution.

The test driver will use test cases derived from the Hennessy Test Cases according to Theorem 2.

Given a fixed standard configuration $(E, ASYS, SYS, I)$ of a reactive system, we will now introduce a test driver which is trustworthy for $\sqsubseteq_{FD}$-test.

Theorem 6 *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:*

$$U(n) = U(n, \langle\, \rangle)$$

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

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

where

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

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

□

Interpretation of Theorem 6 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 †, 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 $\mathit{Ref}(E_I/s)$ 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 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 6 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) $\Longrightarrow$ (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.

Before presenting the proof of the Theorem 6, we will introduce a lemma about Hennessy Tests of type $U_C(s, A)$ in the context of reactive systems.

Lemma 4 *Given P, Q with $\alpha(P) = \alpha(Q)$, $s \in \alpha(P)^*$ and $A \subseteq \alpha(P)$ with $A \neq \emptyset$. Then*

1. $(P \parallel Q)\ \underline{must}\ U_C(s, A) \wedge s \in Traces(P \parallel Q) \implies P\ \underline{must}\ U_C(s, A) \wedge Q\ \underline{must}\ U_C(s, A)$
2. $(P \parallel Q)\ \underline{must}\ U_C(s, A) \implies P\ \underline{must}\ U_C(s, A) \vee Q\ \underline{must}\ U_C(s, A)$

$\square$

Example 6.3.9 Part 1 of Lemma 4 is really as strong as possible: Define $P = a \to b \to STOP$ and $Q = a \to c \to CHAOS$ with $\alpha(P) = \alpha(Q) = \{a, b, c\}$. Then, for arbitrary A, $P\ \underline{must}\ U_C(\langle a, c\rangle, A)$ and $\neg\ (Q\ \underline{must}\ U_C(\langle a, c\rangle, A))$, but $(P \parallel Q)\ \underline{must}\ U_C(\langle a, c\rangle, A)$.
$\square$

Example 6.3.10 The converse of Part 2 of Lemma 4 is not true: Define $P = a \to b \to STOP$ and $Q = a \to c \to STOP$ with $\alpha(P) = \alpha(Q) = \{a, b, c\}$. Then $P\ \underline{must}\ U_C(\langle a\rangle, \{b, c\})$ and $Q\ \underline{must}\ U_C(\langle a\rangle, \{b, c\})$, but $\neg\ ((P \parallel Q)\ \underline{must}\ U_C(\langle a\rangle, \{b, c\}))$.
$\square$

Using the results of Theorem 4 and Theorem 6, now we can state that test drivers using the test cases $U(n)$ have the desired correctness properties:

Theorem 7 *For a given standard configuration $(E, ASYS, SYS, I)$ of a reactive system, let the associated tests $U(n)$ be defined as above. Then the test driver*

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

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

7. Timed Testing Theory – Discrete Values

The objective of this chapter is to show how the untimed testing theory developed above can be lifted to Timed CSP (TCSP), interpreted in the operational semantics of Schneider [30]. As indicated in Chapter 5, this will be achieved by means of a theorem about the structural decomposition of Timed CSP specifications.

7.1 Timed CSP Syntax

The TCSP syntax $\mathcal{L}_1 = \mathcal{L}(P_{L1})$ admitted for our purposes is given in BNF notation as follows:

$$\begin{aligned} P_{L1} \; ::= \; & P_{seq1} \;\mid\; P_{L1} \underset{A}{\|} P_{L1} \;\mid\; P_{L1} \;{}_{A}\|_{A}\; P_{L1} \\ & \mid\; P_{L1} \;|||\; P_{L1} \;\mid\; P_{L1} \setminus A \;\mid\; f(P_{L1}) \mid f^{-1}(P_{L1}) \end{aligned}$$

$$\begin{aligned} P_{seq1} \; ::= \; & \mathrm{STOP} \\ & \mid\; a \rightarrow P_{seq1} \mid\; c\; !\; a \rightarrow P_{seq1} \mid\; c\; ?\; a \rightarrow P_{seq1} \\ & \mid\; a :\; A \;\rightarrow\; P_{seq1} \mid\; a \;\xrightarrow{t}\; P_{seq1} \mid \\ & \mid\; \mathrm{WAIT}\; t;\;\; P_{seq1} \;\mid\; P_{seq1} \;\overset{t}{\triangleright}\; P_{seq1} \\ & \mid\; P_{seq1} \;\Box\; P_{seq1} \;\mid\; \Box_{i \in I}\; P_{i\;\{seq1\}} \\ & \mid\; P_{seq1} \sqcap\; P_{seq1} \;\mid\; \bigsqcap_{i \in I}\; P_{i\;\{seq1\}} \\ & \mid\; X \;\mid\; \mu\; X \;\bullet\; P_{seq1} \end{aligned}$$

This syntax covers the full set of TCSP constructs defined in [30], but we require a separation into a *higher-level* and *lower-level process language*. The former is defined by P_{L1} and meant to compose new processes from existing ones by means of the parallel operator, interleaving, hiding and renaming. The latter – defined by P_{seq1} – allows to specify communicating sequential processes using prefixing, internal and external choice and recursion[1]; furthermore you can make use of the three timing operators *WAIT* t (delay by time t), $a \xrightarrow{t} P$ ($a \rightarrow \mathrm{WAIT}\; t;\;\; P$) and $P \overset{t}{\triangleright} Q$ (if P has not been activated until timeout t occurs, the system will behave as Q). Observe that we allow sequential composition only after the

[1]This syntactic restriction is similar to the one made in the early versions of the FDR tool [9]

WAIT t-operator and do not consider termination by SKIP, only blocking by STOP. The reason for this decision is that currently the treatment of sequential composition ";" and termination "✓" is still inconsistent between different CSP variants, and since we are mostly concerned with non-stop systems that only cease to operate in case of a system failure it does not seem worth the effort to work out the semantic subtleties connected to ";" and "✓".

7.2 A Structural Decomposition Theorem for Timed CSP

Theorem 8 *There exists a transformation $\Omega : \mathcal{L}_1 \rightarrow TCSP$ mapping each process P defined in syntax $\mathcal{L}_1$ to a new process*

$$\Omega(P) = (\Omega_U(P) \underset{TE(P)}{\|} (\|\|_{\,i}\, Timer_i)) \setminus TE(P)$$

such that

1. *$\Omega(P)$ is semantically equivalent to P in the timed failures semantics of TCSP as defined by Schneider in [30].*
2. *$\Omega_U(P)$ does not contain any $WAIT\ t$, timed prefix $a \stackrel{t}{\longrightarrow} P$ or $\stackrel{t}{\rhd}$ construct.*
3. *$TE(P)$ is a set of auxiliary timer events $\{set_i, elapsed_i \mid i : 1 \mathinner{..} n\}$ where n is less or equal to the number of sequential process components in P.*
4. *Each process $Timer_i$ is defined according to the pattern*

$$Timer_i = \mu\, T_i \bullet (set_i?t \rightarrow ((WAIT\ t;\ elapsed_i\ \rightarrow T_i) \;\Box\; T_i))$$

□

The transformation Ω can be explicitly constructed and is currently implemented in the VVT-RT test system. The proof of the theorem is quite lengthy, since it involves three transformations $\Omega = split \circ \Psi \circ \Theta$ where

- Θ is a purely syntactic transformation mapping the abbreviations $a?x, \Box_{i\in I}, \sqcap_{i\in I}, \underset{A}{\|}$ to choice operators without indexing and the $\underset{B\ C}{\|}$ operator.
- Ψ is a semantic equivalence in the timed failures model mapping ran Θ to a further reduced syntactic domain $\mathcal{L}_3$ where (1) recursion operators $\mu X \bullet P(X)$ have been replaced by recursive equations $X = P(X)$, (2) external choice and timeout operators do only have explicit process terms, but no process variables X as operands, (3) consecutive timeout $(P \stackrel{t}{\rhd} Q) \stackrel{u}{\rhd} R$ and $WAIT\ t$ constructs have been replaced by single ones and (4) all external choice operands and first timeout operands P are represented in head-normal form.
- *split* operates on $Q \in$ ran Ψ to separate Q into an untimed part enriched by auxiliary timer events and associated timer components.

Example. TCSP process

$$P = ((\mathit{WAIT}\ t;\ a \to b \to \mathrm{STOP}) \,\square\, c \to \mathit{WAIT}\ v;\ P) \overset{u}{\triangleright} (d \to \mathrm{STOP})$$

is transformed by Ω into

$$\Omega(P) = P_1 \underset{\{|set, elapsed, reset|\}}{\|} \mathit{Timer}$$

$$
\begin{array}{l}
P_1 = \\
\quad \mathbf{if}\ (u < t) \\
\quad \mathbf{then}\ (set!u \to (c \to reset \to set!v \to elapsed \to P_1 \,\square\, elapsed \to d \to STOP)) \\
\quad \mathbf{else}\ (set!t \to (c \to reset \to set!v \to elapsed \to P_1 \\
\qquad\qquad \square \\
\qquad\qquad elapsed \to set!(u-t) \to (a \to reset \to b \to STOP \\
\qquad\qquad\qquad\qquad \square \\
\qquad\qquad\qquad\qquad c \to reset \to set!v \to elapsed \to P_1 \\
\qquad\qquad\qquad\qquad \square \\
\qquad\qquad\qquad\qquad elapsed \to d \to STOP)))
\end{array}
$$

with *Timer* as defined in the theorem above.
□

7.3 Application of the Structural Decomposition Theorem to Timed Testing

The main implication of Theorem 8 is the "recipe" how to augment untimed transition graph interpretation techniques to interpret timed specifications: Suppose that the transition graph TG for $\Omega_U(SPEC)$ has been generated based on the untimed CSP model. As in the preceding chapter, $SPEC$ is the specification of the SUT environment E cooperating with the target system SYS over interface I. Then the timed interpretation of TG is performed according to the following rules:

- Every $SPEC$-event occurs without delay as soon as it is available.
- The auxiliary timer events $set_i.t$ and $reset_i$ occur as soon as they are available for $\Omega_U(SPEC)$ (observe that Ω_U and the $Timer_i$ processes have been defined in such a way that a set_i or $elapsed_i$-event offered by $\Omega_U(SPEC)$ will never be refused by $Timer_i$).
- After having executed trace s each event $e \in [SPEC/s]^0$ with $e \neq elapsed_i$ is either refused in state $SPEC/s$ for an infinite time or immediately accepted.
- Each event $set_i!t$ generates a timed refusal $[0, t] \times elapsed_i$.
- If the associated timed refusal of an event $elapsed_i$ has elapsed, the event will occur immediately (again, the definition of Ω ensures that $elapsed$-events can never be blocked by $\Omega_U(SPEC)$ or the timer processes).

The implementation of a transition graph interpreter observing these timing rules is quite straightforward: Each event $elapsed_i$ is associated with a variable e_i. A $set_i!t$ event sets v_i

to value t. The passing of time by the amount of δ will decrement v_i by δ. As soon as $v_i = 0$ holds $elapsed_i$ becomes available and will occur immediately.

Suppose we are working with a discrete-time model where only a finite number of different time distances t_i can occur and each t_i is a multiple of a minimal time interval δ. In such a case the timer processes can be simply implemented by counters decremented by a clock ticking in δ-time:

Theorem 9 *Suppose the timing values referenced in P are given by $t_j = n_j \cdot \delta$ with $j : 1 \mathrel{..} k, n_j \in \mathbb{N}$. Then $TIMER = (\, |||_{\,i}\, Timer_i)$ can be implemented by*

$$\begin{aligned}
T &= (T_1 \underset{tock}{\|} \ldots \underset{tock}{\|} T_m) \underset{tock}{\|} CLOCK \\
CLOCK &= tock \rightarrow WAIT\ \delta;\ CLOCK \\
T_i &= T_i(0) \\
T_i(u) &= set_i?t \rightarrow T_i(t/\delta) \\
&\quad \Box \\
&\quad tock \rightarrow \textbf{if } (t > 0) \textbf{ then } T_i(u-1) \textbf{ else } T_i(0) \\
&\quad \Box \\
&\quad (u = 0)\&elapsed_i \rightarrow T_i(0) \\
&\quad \Box \\
&\quad reset_i \rightarrow T_i(0)
\end{aligned}$$

$\Box$

This theorem indicates that for discrete-time applications, the test generation strategy is exactly the same as in the untimed case; it only has to be extended to the auxiliary timer events *set*, *elapsed*, *reset* that give rise to a finite number of different clock counter values. The test generation algorithm has to ensure that branches starting with $set_i!t$-events are executed for every possible t value. Since the number of such values is finite, the generation mechanism operates again on a finitely branching transition graph. Test evaluation and calculation of test coverage can also be performed with the methods developed in the untimed case, but applied to the transition graph augmented by timer values.

8. Timed Testing Approach – Continuous Values

Many complex systems have to handle not only discrete events but also continuous values. In general there are two different classes of functionalities such hybrid systems may have to implement:

- Digital outputs have to be generated according to the input values of the analogue interfaces.
- Outputs of analogue interfaces have to fulfill certain conditions, e.g. the values have to be in a specific interval or have to be in an ϵ-environment of a particular function. These conditions may depend on previous target system inputs.

8.1 Test Specifications

In order to use VVT-RT to test the correctness of the behaviour of hybrid systems, the formal test specification of a system module SYS can be decomposed into two parts as follows:

$SYS = PRT \,\|\, PH$

The process PRT describes the discrete value/dense time part of the specification while the process PH handles the continuous value parts. According to the classes of functionalities mentioned above, various patterns for the structure of PH have to be distinguished (see Figure 8.1). In the following we use an extended Hybrid-CSP ($\mathcal{HCSP}$) notation based on [33] to formalize them. According to our experience, real-world systems usually require the test of the following three situations:

1. The test system generates analogue inputs for the target system and checks if it triggers off the correct events at the right time.

   The PRT process has to activate the generation of continuous values, in general according to a differential equation. The PH process listens to this trigger event and executes a continuous statement until a particular condition holds. This condition could be a timeout, an interrupt event generated by the PRT process or it could be a hybrid interrupt based on the values of the function. In extended $\mathcal{HCSP}$ notation these variations can be modelled as follows:

   $$PH = c? \rightarrow < \mathcal{F}(\dot{s}, s) = 0 > \overset{t}{\rhd} PH$$
   $$PH = c? \rightarrow < \mathcal{F}(\dot{s}, s) = 0 > {}^\frown d? \rightarrow PH$$
   $$PH = c? \rightarrow < \mathcal{F}(\dot{s}, s) = 0 > \underset{cond(s)}{\nabla} PH$$

   c is the discrete event which has to be triggered by PRT to start the generation of continuous values according to the differential equation $\mathcal{F}(\dot{s}, s) = 0$. d is a discrete interupt event that can be used within PRT to stop the generation of analogue inputs for the target system or to select a different function afterwards, respectively.

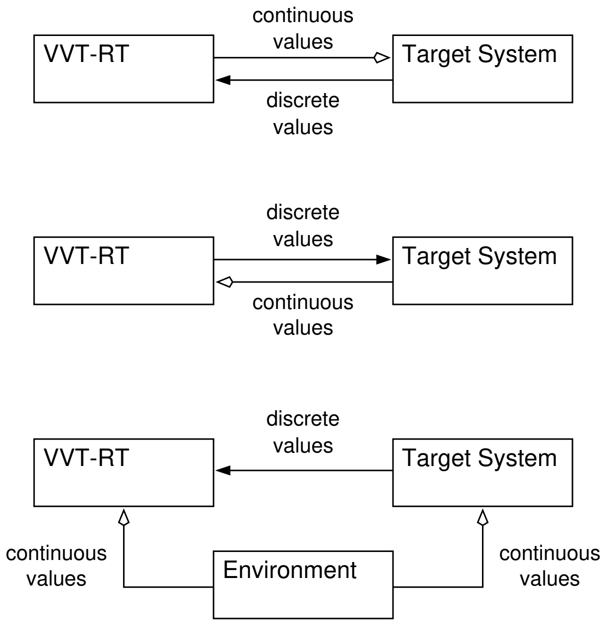

Figure 8.1: Possible structures for the test of hybrid modules

2. The test system generates discrete input events for the target system which trigger off changes on the analogue interfaces. It analyses the corresponding analogue outputs over time which can be done either in real time during the test execution or, if the conditions to be checked are too complex for an on-the-fly evaluation, afterwards by a numerical analysis of the logfiles.

   In general, the PH process has to collect the analogue output values of the target system and evaluate certain conditions on the represented function. Since the test system runs on a digital computer, the analogue values have to be sampled so that the curve is actually described by a list of (value,time)-tuples. For a real time evaluation of the analogue values the following model is necessary to signal the violation of a condition to the PRT process:

   $$\begin{aligned} PH(vs) = h?(v,t) \rightarrow\ & \text{if (not cond}(vs^\wedge(v,t))) \\ & \text{then } c! \rightarrow PH(vs^\wedge(v,t)) \\ & \text{else } PH(vs^\wedge(v,t)) \end{aligned}$$

   The channel h is used to communicate the (value,time)-tuples which are concatenated to the list vs. Whenever the elements in this list violate a static condition $cond()$ this is signalled to the discrete part PRT by the event c. Usually the conditions depend only on a bounded number of latest values. For example, if the specification defines that a certain threshold may never be exceeded, only the last output value has to be checked and a list to store the previous values is unnecessary. In this case, the process PH can be simplified accordingly.

3. A component of the environment generates analogue outputs which serve as inputs for both the target system and the test system. Here the test system has to check the correct behaviour of the target system with respect to the given analogue values. Note that it can only supervise the responses without the possibility to increase the test coverage actively by selecting different curves. This configuration should be used

if the functions to be generated are too complex to be handled by the test system in real time without special hardware or if the environment component is part of the real environment and therefore part of the hardware-in-the-loop-test configuration.

The hybrid parts PH for this kind of test specifications are identical to the specifications for the previous point. Only the PRT process is different since it only has to evaluate the combined (timed) output trace of the target system and the PH process instead of generating inputs to stimulate the system under test.

8.2 Application of the Framework

The framework for testing hybrid systems has been applied to check the correct behaviour of a power and thermal controller (PTC) for a satellite. One of the most important tasks the PTC has to accomplish is the charge and discharge control of the satellite battery. The specification presented in this section describes a simplified version of charge control specification since it should only demonstrate the approach applied. Actually the charge current depends additionally on various conditions of battery pressure and temperature as well as the bus voltage. Also, the following informal specification of the simplified charge control handles only the main charge phase:

> During the sunphase the battery is charged with a current in the interval `I_Charge`. This starts at least `t` seconds after the sunrise and lasts either until the discharge amount taken in the previous shadephase is recharged and a battery pressure of at least `P_Minabs` is reached or until the sunphase ends.

This can be translated directly into the corresponding discrete value CSP test specification:

$$
\begin{array}{l}
CHARGECTRL = Tau\_SUN\_ON \rightarrow \\
\quad (Evt\_I\_Charge \rightarrow CHARGE(false, false)) \overset{t}{\rhd} (error \rightarrow STOP)
\end{array}
$$

$$
\begin{array}{l}
\mathrm{CHARGE(press\_ok,\ recharged)} = \\
\quad Evt\_NOT\_I\_Charge \rightarrow error \rightarrow CHARGE(press\_ok, recharged) \\
\quad \Box \\
\quad Evt\_Press\_Ok \rightarrow \ \mathrm{if}\,(recharged) \\
\qquad\qquad\qquad\quad \mathrm{then}\ SUPPLEMENTARYCHARGE \\
\qquad\qquad\qquad\quad \mathrm{else}\ CHARGE(true, recharged) \\
\quad \Box \\
\quad Evt\_Recharged \rightarrow \ \mathrm{if}\,(press\_ok) \\
\qquad\qquad\qquad\quad \mathrm{then}\ SUPPLEMENTARYCHARGE \\
\qquad\qquad\qquad\quad \mathrm{else}\ CHARGE(press\_ok, true) \\
\quad \Box \\
\quad Tau\_SUN\_OFF \rightarrow DISCHARGE
\end{array}
$$

The hybrid parts analyse the battery charge current and the battery pressure. In order to perform the test of the PTC the following $\mathcal{HCSP}$ specifications were implemented as C programs and ran in parallel with the VVT-RT tool.

The process CCC (charge current control) checks if the charge current communicated over the channel cc exceeds the `I_Charge` thresholds. Therefore, it needs to store only the previous value (in l). Whenever the step from the previous to the current value crosses the `I_Charge` boundaries the corresponding event for the discrete part is triggered:

$$\begin{aligned} CCC(l) = cc?(v,t) \rightarrow\ & \text{if}\,(v \in \texttt{I\_Charge} \wedge l \notin \texttt{I\_Charge}) \\ & \text{then}\ Evt\_I\_Charge \rightarrow CCC(v) \\ & \text{else}\ \text{if}\,(v \notin \texttt{I\_Charge} \wedge l \in \texttt{I\_Charge}) \\ & \qquad \text{then}\ Evt\_Not\_I\_Charge \rightarrow CCC(v) \\ & \qquad \text{else}\ CCC(v) \end{aligned}$$

The channel cc, like all other continuous value channels in this section, is connected to the target system as depicted in Figure 8.1. The target system transmits the corresponding values over these channels in regular time intervals.

In addition, the discharge and charge amounts have to be compared. These amounts are represented by the integrals over the battery charge current during the last shadephase and the current sunphase, respectively:

$$AC = Tau\_SUN\_OFF \rightarrow CAC(<>)$$

$$\begin{aligned} CAC(vs) = cc?(v,t) \rightarrow\ & \text{if}\,(\int_{x \in shadephase} vs \leq \int_{x \in sunphase} (vs^\wedge(v,t))) \\ & \text{then}\ Evt\_Recharged \rightarrow AC \\ & \text{else}\ CAC(vs^\wedge(v,t)) \\ \square & \\ Tau\_SUN\_OFF & \rightarrow CAC(<>) \end{aligned}$$

Finally, the current battery pressure, which is communicated over the channel bp, has to be evaluated. If it exceeds the `P_Minabs` value the event *Evt_Press_Ok* is generated. Due to the fact that the charge current has to be greater than zero during the main charge phase, the pressure increases monotonously and will not fall below the threshold again as long as the sun does not set:

$$\begin{aligned} PC(l) = bp?(v,t) \rightarrow\ & \text{if}\,(v \geq \texttt{P\_Minabs} \wedge l < \texttt{P\_Minabs}) \\ & \text{then}\ Evt\_Press\_Ok \rightarrow PC(v) \\ & \text{else}\ PC(v) \end{aligned}$$

The whole charge control specification comprises all four introduced processes. The hybrid processes are interleaved and synchronized with the discrete part via all events with the prefix *Evt_*:

$$CCSYS = CHARGECTRL \underset{\{Evt\_xxx\}}{\|} (CCC(0)\ |||\ AC\ |||\ PC(0))$$

The *Tau_SUN* events which occur in these specifications are generated by a sun simulation process. This also comprises a discrete and a hybrid part. The discrete process *SUN* chooses between different possible lengths p for the sunphase and between different durations d of the sunrise and the sunset. One cycle in the orbit takes R minutes so that the shadephase always lasts $R - d$ minutes. Furthermore, the process triggers off the hybrid part *SC* by the events with the prefix *Evt_* to control a hardware solar array simulator according to the selected duration. The *Tau_SUN* events are internal events to signal the current sun state to other discrete parts of the test specifications.

$$\begin{aligned} SUN = & \sqcap\, p : \{p_{min}, \ldots, p_{max}\} \\ & \rightarrow \sqcap\, d : \{d_{min}, \ldots, d_{max}\} \end{aligned}$$

$\rightarrow Evt\_SUN\_ON.d$
$\rightarrow Tau\_SUN\_ON$
$\rightarrow$ WAIT d;
$\rightarrow \sqcap\, d : \{d_{min}, \ldots, d_{max}\}$
$\rightarrow Evt\_SUN\_OFF.d$
$\rightarrow Tau\_SUN\_OFF$
$\rightarrow$ WAIT $(R - d)$;
$\rightarrow SUN$

In order to control the solar array simulator which has been used for the PTC hardware-in-the-loop test it is necessary to transmit the percentage of the sun that is currently visible for the solar strings. In the abstract hybrid specification of the sun, this percentage is calculated by the functions r and s for the sunrise and sunset, respectively. Their results depend on the time, the last percentage and the selected duration for the sunrise/sunset:

$$\begin{aligned} SC(x) = \; & Evt\_SUN\_ON?d \rightarrow\; < x = r(t,x,d) > \nabla_{x=100} SC(100) \\ & \square \\ & Evt\_SUN\_OFF?d \rightarrow\; < x = s(t,x,d) > \nabla_{x=0} SC(0) \end{aligned}$$

The sun control specification $SCSYS$ uses the same synchronization pattern as the charge control process $CCSYS$:

$$SCSYS = SUN \underset{\{Evt\_xxx\}}{\parallel} SC(0)$$

Finally, $SCSYS$ and $CCSYS$ have to be synchronized via the Tau_SUN events. The sun specification generates sunrises and sunsets at various times and the charge control part is only allowed to refuse these events during the phase changes. This cannot cause problems as long as the sun timers will never let the sun go down again before it did not rise completely and the other way round:

$$SYS = SCSYS \underset{\{Tau\_SUN\_ON,\, Tau\_SUN\_OFF\}}{\parallel} CCSYS$$

Bibliography

[1] P. Amthor: HySC – A Development Environement for Hybrid Systems Components, to appear in proceedings of the Tool'98 – Tool Support for System Specifications, Development and Verification, Malente, July 2-4, 1998.

[2] K. R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs.* Springer-Verlag, Berlin Heidelberg New York (1991).

[3] E. Brinksma: A theory for the deriviation of tests. In P. H. J. van Eijk, C. A. Vissers and M. Diaz (Eds.): *The Formal Description Technique LOTOS.* Elsevire Science Publishers B. V. (North-Holland), (1989) 235-247.

[4] Rachel Cardell-Oliver and Tim Glover: A Practical and Complete Algorithm for Testing Real-Time Systems. To appear in FTRTFTS98, Sep 98, Springer Verlag LNCS.

[5] T. S. Chow: Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178-187, 1978.

[6] J. Davies: *Specification and Proof in Real-Time CSP.* Cambridge University Press (1993).

[7] R. de Nicola and M. C. Hennessy: Testing Equivalence for Processes. *Journal of Theoretical Computer Science* (1983)34: 83-133

[8] RTCA DO178B: *Development considerations in airborne computer systems.* (1993).

[9] Formal Systems (Europe) Ltd: *FDR2 User Manual.* 24 October 1997.

[10] M.-C. Gaudel: Testing can be formal, too. In P. D. Mosses, M. Nielsen and M. I. Schwartzbach (Eds.): *Proceedings of TAPSOFT '95: Theory and Practice of Software Development.* Aarhus, Denmark, May 1995, Springer-Verlag, Berlin Heidelberg New York (1995).

[11] M. C. Hennessy: *Algebraic Theory of Processes.* MIT Press (1988).

[12] M. C. Hennessy and A. Ingólfsdóttir: Communicating Processes with Value-passing and Assignments. *Formal Aspects of Computing* (1993) 5: 432-466

[13] C.A.R. Hoare: *Communicating sequential processes.* Prentice-Hall International, Englewood Cliffs (1985).

[14] G. J. Meyers: *The Art of Software-Testing.* John Wiley&Sons, New York (1979).

[15] E. Mikk: Compilation of Z Specifications into C for Automatic Test Result Evaluation. In J. P. Bowen and M. G. Hinchey (Eds.): *ZUM '95: 9th International Conference of Z Users*, LNCS, Springer-Verlag, Berlin Heidelberg New York (1995).

[16] R. Milner: *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs NJ (1989).

[17] E.-R. Olderog: *Nets, Terms and Formulas.* Cambridge University Press (1991).

[18] H.-M. Hörcher and J. Peleska: Using formal specifications to support software testing. Software Quality Journal (1995) 4 : 309-327.

[19] J. Peleska and M. Siegel: From Testing Theory to Test Driver Implementation. In M.-C. Gaudel and J. Woodcock (Eds.): FME '96: Industrial Benefit and Advances in Formal Methods. LNCS 1051, Springer-Verlag, Berlin Heidelberg New York (1996) 538-556.

[20] J. Peleska: Test Automation for Safety-Critical Systems: Industrial Application and Future Developments. (invited keynote), In M.-C. Gaudel and J. Woodcock (Eds.): FME '96: Industrial Benefit and Advances in Formal Methods. LNCS 1051, Springer-Verlag, Berlin Heidelberg New York (1996) 39-59.

[21] J. Peleska: Formal Methods and the Development of Dependable Systems. Habilitationsschrift, Bericht Nr. 9612, Dezember 1996, Institut für Informatik und praktische Mathematik, Christian-Albrechts-Universität Kiel (1997).

[22] J. Peleska and M. Siegel: Test Automation of Safety-Critical Reactive Systems. *South African Computer Jounal* (1997)19:53-77.

[23] Gerd Urban, Hans-Joachim Kolinowitz and Jan Peleska: A Survivable Avionics System for Space Applications. Published in Proceedings of the FTCS-28, 28th Annual Symposium on Fault-Tolerant Computing, Munich, June 23-25, 1998, 372-381.

[24] A. P. Ravn: *Design of Embedded Real-time Computing Systems.* Dr. techn. dissertation, Report No. ID-TR 1995-170, ID/DTU, Lyngby, Denmark (1995).

[25] A. W. Roscoe and G. Barret: Unbounded Nondeterminism in CSP. In *MFPS '89*, LNCS 298, Springer-Verlag, Berlin Heidelberg New York (1989).

[26] A. W. Roscoe: Model-Checking CSP. In *A Classical Mind, Essays in Honour of CAR Hoare.* Prentice-Hall International, Englewood Cliffs NJ (1994).

[27] A. W. Roscoe: *The Theory and Practice of Concurrency.* Prentice-Hall (1998).

[28] S. Sadeghipour: *Testing Cyclic Software Components of Reactive Systems on the Basis of Formal Specifications.* Doctoral Thesis, Technical University of Berlin (1998).

[29] J.W. Davies and S.A. Schneider. A Brief History of Timed CSP. Technical report, University of Oxford, 1992.

[30] S. Schneider. An Operational Semantics for Timed CSP. *Information and Computation*, 116:193–213, 1995.

[31] S. Schneider. Timewise Refinement for Communicating Processes. *Science of Computer Programming*, 28:43–90, 1997.

[32] M. B. Schronen: *Methodology for the Development of Micro Processor-Based Safety-Critical Systems.* Doctoral Thesis University of Bremen, Department of Mathematics and Computer Science, to appear in Monographs of the Bremen Institute of Safe Systems, Shaker Verlag (1998).

[33] Zhou, Chaochen and A. P. Ravn and M. R. Hansen: An extended Duration Calculus for Hybrid Real-time systems. In R. L. Grossman and A. Nerode and A. P. Ravn and H. Rischel (eds.): Hybrid Systems. Springer LNCS 763 (1993) 36-59.

Part III

Appendixes

Appendix A. Proofs for the Untimed Testing Theory

In this appendix we present the proofs for lemmas and theorems given in Chapter 6.

Proof of Theorem 2.
We prove 2, the other cases are shown analogously. The proof is structured as follows: Our first objective is to show that the test cases $U_C(s, A) \in \mathcal{H}_C(SPEC)$ are meaningful for the $\underline{must}$-test against $SPEC$ in the sense of Definition 5. By definition of $\mathcal{H}_C(SPEC)$, these test cases only execute traces of $SPEC$ before successful termination, so they are obviously meaningful for the $\overline{may}$-test against $SPEC$. It remains to show that $SPEC\ \underline{must}\ U_C(s, A)$ for these test cases. This is demonstrated in Proof Obligation 2.1. Next, we wish to show that the test cases in $\mathcal{H}_S(SPEC) \cup \mathcal{H}_C(SPEC)$ are powerful enough to detect every possible violation of $\sqsubseteq_F$-refinement. From Part 1 of the theorem we know that the tests of $\mathcal{H}_S(SPEC)$ suffice to detect every violation of $\sqsubseteq_T$-refinement. It remains to show that the tests of $\mathcal{H}_C(SPEC)$ detect missing requirements coverage. Now Theorem 1 implies that the test $U_C(s, A)$ are at least capable for this task if we consider *all* traces $s \in \alpha(SPEC)^*$ and sets of events $A \subseteq \alpha(SPEC)$. We will therefore prove that whenever an *arbitrary* $\underline{must}$-test $U_C(s, A)$ satisfied by $SPEC$ fails for an implementation IMP, there will also be a test *contained in* $\mathcal{H}_C(SPEC)$ which fails for IMP, too. Now the collection of all test cases $U_C(s, A)$ for which $SPEC\ \underline{must}\ U_C(s, A)$ holds can be divided into two groups, depending on whether s is a trace of $SPEC$ or not. The former case is handled by Proof Obligation 2.2, the latter by Proof Obligation 2.3.

Proof Obligation 2.1. For $U_C(s, A) \in \mathcal{H}_C$, show that $SPEC\ \underline{must}\ U_C(s, A)$.

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

Case 2.1.1. Suppose, an execution of $(U_C(s, A) \parallel SPEC)$ does not reach $last(s)$. Since neither $U_C(s, A)$ nor $SPEC$ diverge along s, the same holds for $(U_C(s, A) \parallel SPEC)$, therefore $SPEC$ must block before $last(s)$ is reached, and as a consequence $(U_C(s, A) \parallel SPEC)$ is ready to produce any event not contained in $\alpha(SPEC)$. Since the only event not contained in $\alpha(SPEC)$ and executable in this state is w, this leads to successful execution of $(U_C(s, A) \parallel SPEC)$.

Case 2.1.2. Suppose $last(s)$ is reached in an execution of $(U_C(s, A) \parallel SPEC)$. Since $SPEC$ does not diverge along s, application of the Normal Form Theorem 1 yields

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

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

$$(\forall\, R : Ref(SPEC/s) \bullet (\exists\, a_R : A \bullet a_R \in ([SPEC/s]^0 \setminus R)))$$

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. Case 2.1.1 and 2.1.2 prove that $SPEC \; \underline{must} \; U_C(s, A)$. This shows the correctness of Proof Obligation 2.1.

Proof Obligation 2.2. Show that for all $s_1 \in Traces(SPEC)$

$$\begin{aligned}&(\forall A_1 : \mathbb{P}\,\alpha(SPEC) \bullet \\ &\quad SPEC \; \underline{must} \; U_C(s_1, A_1) \wedge \neg\,(IMP \; \underline{must} \; U_C(s_1, A_1)) \implies \\ &\qquad (\exists A : \mathbb{P}\,\alpha(SPEC) \bullet U_C(s_1, A) \in \mathcal{H}_C \wedge \neg\,(IMP \; \underline{must} \; U_C(s_1, A_1))))\end{aligned}$$

holds. This means that whenever IMP contains a requirements coverage failure (s_1, A_1) with $s_1 \in Traces(SPEC)$ and arbitrary A_1, it will also contain a requirements coverage failure (s_1, A) which can be detected by a test $U_C(s_1, A) \in \mathcal{H}_C$.

Assume

$$s_1 \in Traces(SPEC) \wedge SPEC \; \underline{must} \; U_C(s_1, A_1) \wedge U_C(s_1, A_1) \notin \mathcal{H}_C \wedge \neg\,(IMP \; \underline{must} \; U_C(s_1, A_1))$$

Then, because of the structure of $U_C(s_1, A_1)$, two situations may be the cause for the failure of IMP:

1. $s_1 \in Div(IMP)$
2. $s_1 \in Traces(IMP) - Div(IMP) \wedge (\exists R : Ref(IMP/s_1) \bullet A_1 \subseteq R)$

In both cases we have $s_1 \in Traces(IMP)$. Because

$$s_1 \in Traces(SPEC) \wedge SPEC \; \underline{must} \; U_C(s_1, A_1)$$

$A_1 \cap [SPEC/s]^0$ cannot be completely contained in any of the $R \in Ref(SPEC/s)$. Let $A \subseteq A_1 \cap [SPEC/s]^0$ be one of the smallest sets such that $(\forall R : Ref(SPEC/s) \bullet A \not\subseteq R)$. Then $U_C(s_1, A) \in \mathcal{H}_C$ and $SPEC \; \underline{must} \; U_C(s_1, A)$.

Case 2.2.1. If $s_1 \in Div(IMP)$ is the cause for $\neg\,(IMP \; \underline{must} \; U_C(s_1, A_1))$, then also $\neg\,(IMP \; \underline{must} \; U_C(s_1, A))$, because divergence along s_1 will lead to an execution of $(U_C(s_1, A) \parallel IMP)$ where any visible event is refused before success w could be signalled.

Case 2.2.2. If $s_1 \in Traces(IMP) - Div(IMP) \wedge (\exists R : Ref(IMP/s_1) \bullet A_1 \subseteq R)$ is the cause for $\neg\,(IMP \; \underline{must} \; U_C(s_1, A_1))$, then also $A \subseteq A_1$ will be contained in such an R. This yields once more $\neg\,(IMP \; \underline{must} \; U_C(s_1, A))$, because an execution of $(U_C(s_1, A) \parallel IMP)$ will block A-events after s_1, and $U_C(s_1, A)/s_1$ can signal success w only after having first engaged into some $a \in A$.

Proof Obligation 2.3. Show that for all $s_1 \notin Traces(SPEC)$

$$\begin{aligned}&(\forall A_1 : \mathbb{P}\,\alpha(SPEC) \bullet \\ &\quad \neg\,(IMP \; \underline{must} \; U_C(s_1, A_1)) \implies \\ &\qquad (\exists s : Traces(SPEC);\; a : \alpha(SPEC) \bullet U_S(s, a) \in \mathcal{H}_S \wedge \neg\,(IMP \; \underline{must} \; U_S(s, a))))\end{aligned}$$

holds. To understand the objective of this proof obligation, recall that $SPEC \; \underline{must} \; U_C(s_1, A_1)$ holds for $s_1 \notin Traces(SPEC)$ and any A_1: $SPEC$ cannot diverge after a prefix u of s_1 because otherwise any extension of u would be contained in $Traces(SPEC)$. As a consequence, $SPEC$ will block after a prefix u of s_1 and $U_C(s_1, A_1)/u$ is forced to take the branch $w \rightarrow SKIP$. Conversely, $\neg\,(IMP \; \underline{must} \; U_C(s_1, A_1))$ implies that $s_1 \in Traces(IMP)$.

Since $s_1 \notin Traces(SPEC)$ there exists s_{11}, s_{12} such that $s_1 = s_{11} \frown s_{12} \wedge s_{11} \in Traces(SPEC) \wedge head(s_{12}) \notin [SPEC/s_{11}]^0$. As a consequence, $SPEC \; \underline{must} \; U_S(s_{11}, head(s_{12}))$, but $s_1 \in Traces(IMP)$ implies $\neg \; (IMP \; \underline{must} \; U_S(s_{11}, head(s_{12}))$. Since $U_S(s_{11}, head(s_{12})) \in \mathcal{H}_S$, this proves Obligation 2.3 and completes the proof of 2.
□

Proof of Theorem 3.
Proof of 1. Suppose that $U_S(s, a) \in \mathcal{H}_S$ and $\mathcal{H} = \mathcal{H}_S - \{U_S(s, a)\}$. Define

$$\alpha(P) = \alpha(SPEC)$$

$$Traces(P) = Traces(SPEC) \cup \{s \frown \langle a \rangle\}$$

$$\begin{aligned} P/u = \; & \textbf{if } u = s \\ & \quad \textbf{then } (x : [SPEC/u]^0 \rightarrow P/u \frown \langle x \rangle \; \square \; a \rightarrow STOP) \\ & \quad \textbf{else } (x : [SPEC/u]^0 \rightarrow P/u \frown \langle x \rangle) \end{aligned}$$

The $P \; \underline{must} \; U$ holds for all $U \in \mathcal{H}$, but P violates the safety requirement $s \frown \langle a \rangle \notin Traces(P)$.

Proof of 2. 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 $R \in Ref(SPEC/s)$ such that $B \subseteq R$. As a consequence there exists an execution $(U_C(s, B) \parallel SPEC)$ blocking and consequently failing after trace s. Therefore, $\neg \; (SPEC \; \underline{must} \; U_C(s, B))$ holds, which proves (2).

Proof of 3. Let $\{U_C(s, A_1), \ldots , U_C(s, A_n)\} \subseteq \mathcal{H}_C$ be an enumeration of the $\underline{must}$-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

$$\alpha(P) = \alpha(SPEC)$$

$$Traces(P) = Traces(SPEC)$$

$$\begin{aligned} P/u = \; & \textbf{if } u = s \\ & \quad \textbf{then } (x : ([SPEC/u]^0 \setminus A_n) \rightarrow P/u \frown \langle x \rangle \; \square \; ((x : A_n \rightarrow P/u \frown \langle x \rangle) \sqcap STOP)) \\ & \quad \textbf{else } (x : [SPEC/u]^0 \rightarrow P/u \frown \langle x \rangle) \end{aligned}$$

Since $Traces(P) = Traces(SPEC)$ by construction, $P \; \underline{must} \; U$ holds for all safety tests in $\mathcal{H}_S$. For traces $u \neq s$, $P \; \underline{must} \; U_C(u, A)$ 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, $[SPEC/u]^0 \setminus A_n$ contains at least one A_i-event for all $i = 1, \ldots , (n - 1)$. This implies $P \; \underline{must} \; U_C(s, A_i)$ for $i = 1, \ldots , (n - 1)$, so $P \; \underline{must} \; U$ 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 Theorem 4.
Since $\alpha(SPEC)$ is finite, every $[IMP/s]^0$ is also finite. As a consequence, $Traces(IMP)$

contains only a finite number of traces with fixed length $n \in \mathbb{N}$. Since also $\mathit{refMax}(\mathit{IMP}/s)$ is always finite, the number of Hennessy Tests $U_i \in \mathcal{H}$ with defining trace of length less or equal n will also be finite.

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

Proof of Theorem 5.
Proof Obligation 1: We show that $[D_0/s]^0 \setminus \{w\} = [E_I/s \upharpoonright I]^0$ for all traces s of D_0, so that, as a consequence, $D_0 \sqsubseteq_T E_I$.

The definitions of $D_0(s)$ and $D_0(s,x)$ show that for every s we have either $[D_0/s]^0 = \bigcup_{R:\mathit{Ref}(E_I/s\upharpoonright I)}([E_I/s \upharpoonright I]^0 \setminus R)$ or $[D_0/s]^0 = \{w\} \cup \bigcup_{R:\mathit{Ref}(E_I/s\upharpoonright I)}([E_I/s \upharpoonright I]^0 \setminus R)$. Since $\emptyset \in \mathit{Ref}(E_I/s)$ this yields $[D_0/s]^0 = [E_I/s \upharpoonright I]^0$ or $[D_0/s]^0 = \{w\} \cup [E_I/s \upharpoonright I]^0$, which proves Obligation 1.

Proof of Part 1. Let $\mathcal{D}(X)$ be meaningful for the test against SPEC and U be a test case occurring in $\mathcal{D}(X)$. We show that $D_0 \sqsubseteq_T (U \parallel E_I)$, which is is equivalent to $\mathit{Traces}(U \parallel E_I) \subseteq \mathit{Traces}(D_0)$:

Case 1.1. Let $s \in \mathit{Traces}(U \parallel E_I) \wedge \neg (\langle w \rangle \ \mathbf{in}\ s)$. Then the definition of the parallel operator implies that $s \in \mathit{Traces}(E_I)$. From Obligation 1 $s \in \mathit{Traces}(D_0)$ follows. This proves part 1 in Case 1.1.

Case 1.2. Let $s \frown \langle w \rangle \frown u \in \mathit{Traces}(U \parallel E_I)$. Since U is an admissible test case in the sense of Definition 3, $\neg (\langle w \rangle \ \mathbf{in}\ s \frown u)$ and $u = \langle\ \rangle$ follows, so s contains I-events only, and $s \in \mathit{Traces}(D_0)$ follows as in Case 1.1. Since U is meaningful in the sense of Definition 8, we have $s \in \mathit{Traces}(\mathit{SPEC})$, because U indicates success after s. Inserting the reactive system structure for SPEC implies $s \in \mathit{Traces}(E_I \parallel \mathit{ASYS}_I)$. The $\parallel$-law implies $s \in \mathit{Traces}(\mathit{ASYS}_I)$. Then the definition of $D_0(\mathit{front}(s), \mathit{last}(s))$ implies $w \in [D_0/s]^0$. Therefore we have $s \frown \langle w \rangle \in \mathit{Traces}(D_0)$. This shows Case 1.2 and completes the proof of part 1 of the theorem.

Proof of Part 2. Let $D_0 \sqsubseteq_T (U \parallel E_I)$ and $(\forall s \bullet s \frown \langle w \rangle \in \mathit{Traces}(U) \implies s \in \mathit{Traces}(E_I))$ be valid for every test case U appearing as a parameter in $\mathcal{D}(X)$. We show that $\mathcal{D}(X)$ performs meaningful $\underline{\mathit{may}}$-test executions against SPEC:

Let $s \frown \langle w \rangle \in \mathit{Traces}(U)$. We have to show that $s \in \mathit{Traces}(\mathit{SPEC})$. According to the assumptions, s is a trace both of U and E_I, therefore the law about $\parallel$ yields $s \frown \langle w \rangle \in \mathit{Traces}(U \parallel E_I)$. Since $(U \parallel E_I)$ refines D_0 in the trace model, $s \frown \langle w \rangle \in \mathit{Traces}(D_0)$ follows. According to the definition of $D_0(\mathit{front}(s), \mathit{last}(s))$, this implies $s \in \mathit{Traces}(\mathit{ASYS}_I)$. As a consequence, $s \in \mathit{Traces}(E_I) \cap \mathit{Traces}(\mathit{ASYS}_I)$, so since $\mathit{SPEC} = (E_I \parallel \mathit{ASYS}_I)$, this results in $s \in \mathit{Traces}(\mathit{SPEC})$. This completes the proof of part 2 of the theorem.
□

Proof of Lemma 4.
Proof of 1. Suppose that $(P \parallel Q)\ \underline{\mathit{must}}\ U_C(s,A) \wedge s \in \mathit{Traces}(P \parallel Q)$ holds but $P\ \underline{\mathit{must}}\ U_C(s,A) \wedge Q\ \underline{\mathit{must}}\ U_C(s,A)$ is false. Specifically, assume that $\neg(P\ \underline{\mathit{must}}\ U_C(s,A))$ holds. Then $(s,A) \in \mathit{Fail}(P)$ by Lemma 3. Since $s \in \mathit{Traces}(P \parallel Q)$ and $\alpha(P) = \alpha(Q)$, also

$s \in \mathit{Traces}(Q)$ holds. From to the semantic definition of $\|$, $(s, A) \in \mathit{Fail}(P \| Q)$ follows. Lemma 3 now implies $\neg((P \| Q)\ \underline{\mathit{must}}\ U_C(s, A))$, a contradiction. The same argument can be applied for Q. This proves 1.

Proof of 2. Applying contraposition, we assume that neither $P\ \underline{\mathit{must}}\ U_C(s, A)$ nor $Q\ \underline{\mathit{must}}\ U_C(s, A)$ hold. Then Lemma 3 implies $(s, A) \in \mathit{Fail}(P)$ and $(s, A) \in \mathit{Fail}(Q)$. Again, the semantic definition of $\|$ implies $(s, A) \in \mathit{Fail}(P \| Q)$, so $\neg((P \| Q)\ \underline{\mathit{must}}\ U_C(s, A))$ follows. This completes the proof of the lemma.
□

Proof of Theorem 6: We will first state and verify four proof obligations that are used to establish the validity of Theorem 6, 2.(a) $\implies$ 2.(b).

Proof Obligation 1. Show that

$$s \in \mathit{Traces}((E \| \mathit{ASYS})_I) \wedge \#s < n \implies s \in \mathit{Traces}(U(n)) \wedge U(n)/s = U(n, s)$$

Given $U(n)$ with fixed $n > 0$, we will use induction over the length $m = \#s$ of the traces s, $m = 0, \dots, n-1$. 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 $s \in \mathit{Traces}((E \| \mathit{ASYS})_I) \wedge \#s = m + 1 \wedge m + 1 < n$. Define trace $u = \mathit{front}(s)$ and event $a = \mathit{last}(s)$.

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

$$\begin{aligned} U(n)/u = U(n, u) = \ & (A(u) = \emptyset) \& (w \rightarrow \mathit{SKIP}) \\ & \square \\ & (e : ([E_I/u]^0 \setminus [\mathit{ASYS}_I/u]^0) \rightarrow \dagger \rightarrow \mathit{SKIP}) \\ & \square \\ & (\sqcap_{R:R(u)} U(n, u, [(E \| \mathit{ASYS})_I/u]^0 \setminus R)) \end{aligned}$$

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

Since $\emptyset \in R(u)$, process $U(n, u, [(E \| \mathit{ASYS})_I/u]^0)$ is an alternative of the $(\sqcap_{R:R(u)} \dots)$-branch in $U(n, u)$. As a consequence, $[U(n, u, [(E \| \mathit{ASYS})_I/u]^0)]^0 \subseteq [U(n)/u]^0$ holds. By construction of the processes $U(n, s, M)$ we have

$$[U(n, u, [(E \| \mathit{ASYS})_I/u]^0)]^0 = [(E \| \mathit{ASYS})_I/u]^0$$

Since $a \in [(E \| \mathit{ASYS})_I/u]^0$ we get $a \in [U(n)/u]^0$. This proves $s = u ^\frown \langle a \rangle \in \mathit{Traces}(U(n))$.

Since the first and the second $\square$-branch of $U(n)/u$ do not accept a, we obtain $U(n)/u ^\frown \langle a \rangle = U(n, u, [(E \| \mathit{ASYS})_I/u]^0 \setminus R)/\langle a \rangle$ for some $R \in R(u)$ with $a \in [(E \| \mathit{ASYS})_I/u]^0 \setminus R$. (Note, that for all such sets R, R' we have by definition of $U(n, s, M)$ that $U(n, u, [(E \| \mathit{ASYS})_I/u]^0 \setminus R)/\langle a \rangle = U(n, u, [(E \| \mathit{ASYS})_I/u]^0 \setminus R')/\langle a \rangle$). 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 Theorem 2 show that

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

By Lemma 3, $\neg\ ((E \parallel SYS)_I\ \underline{must}\ U_S(s, a))$ implies $s^\frown\langle a\rangle \in \mathit{Traces}((E \parallel SYS)_I)$. From the semantic definition of $\parallel$, $s^\frown\langle a\rangle \in \mathit{Traces}(E_I)$ and $s^\frown\langle a\rangle \in \mathit{Traces}(SYS_I)$ follows. Since $U_S(s, a) \in \mathcal{H}_S$, $s \in \mathit{Traces}((E \parallel ASYS)_I) \wedge a \notin [(E \parallel ASYS)_I/s]^0$, so the semantics of $\parallel$ yields $s \in \mathit{Traces}(ASYS_I) \wedge s^\frown\langle a\rangle \notin \mathit{Traces}(ASYS_I)$. 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 $\dagger \rightarrow \mathit{SKIP}$, 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

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

$U_C(s, A) \in \mathcal{H}_C((E \parallel ASYS)_I)$ and part 1 of Lemma 4 imply
$A \in A(s) \wedge s \in \mathit{Traces}((E \parallel ASYS)_I) \wedge E_I\ \underline{must}\ U_C(s, A) \wedge ASYS_I\ \underline{must}\ U_C(s, A)$. Since $\neg\ ((E \parallel SYS)_I\ \underline{must}\ U_C(s, A))$, Lemma 3 implies $(s, A) \in \mathit{Fail}((E \parallel SYS)_I)$, that is,

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

The definition of $\parallel$ implies that such a refusal R is a union of maximum refusals $X_1 \in \mathit{refMax}(E_I/s)$ and $X_2 \in \mathit{refMax}(SYS_I/s)$, 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:

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

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

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

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

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

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

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

The premise $U_D(s) \in \mathcal{H}_D((E \,\|\, ASYS)_I)$ implies $s \in Traces(E_I \,\|\, ASYS_I)$ and $s \notin Div(E_I \,\|\, ASYS_I)$. The semantics of $\|$ implies $s \notin Div(E_I) \wedge s \notin Div(ASYS_I)$. Let $u \leq s$ such that $u \in Div((E \,\|\, SYS)_I)$. Since u is a prefix of s, $u \notin Div(E_I)$ also holds and $u \in Div(SYS_I)$ follows. Since $s \in Traces((E \,\|\, ASYS)_I)$, the validity of proof obligation 1 shows that $u, s \in Traces(U(n))$. Therefore $u \in Traces(U(n) \,\|\, SYS_I)$, and the definition of $\|$ yields $u \in Div(U(n) \,\|\, SYS_I)$. As a consequence, $(U(n) \,\|\, SYS_I)$ 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 6, 2.(a) $\implies$ 2.(b). Suppose $SYS_I \;\underline{must}\; U(n)$ for all $n \in \mathbb{N}$ and let $U_D(s) \in \mathcal{H}_D((E \,\|\, ASYS)_I)$. Then $(E \,\|\, SYS)_I \;\underline{must}\; U_D(s)$, because otherwise $\neg\, (SYS_I \;\underline{must}\; U(\#s+1))$ according to proof obligation 4. Let $U_S(s,a) \in \mathcal{H}_S((E \,\|\, ASYS)_I)$. Then $(E \,\|\, SYS)_I \;\underline{must}\; U_S(s,a)$, because otherwise $\neg\, (SYS_I \;\underline{must}\; U(\#s+1))$ according to proof obligation 2. Let $U_C(s,A) \in \mathcal{H}_C((E \,\|\, ASYS)_I)$. Then $(E \,\|\, SYS)_I \;\underline{must}\; U_C(s,A)$, because otherwise $\neg\, (SYS_I \;\underline{must}\; U(\#s+1))$ according to proof obligation 3.

Now we have established that $(E \,\|\, SYS)_I \;\underline{must}\; U$ holds for all $U \in \mathcal{H}_S((E \,\|\, ASYS)_I) \cup \mathcal{H}_C((E \,\|\, ASYS)_I) \cup \mathcal{H}_D((E \,\|\, ASYS)_I)$. Therefore the application of Theorem 2 results in $(E \,\|\, ASYS)_I \sqsubseteq_{FD} (E \,\|\, SYS)_I$. This proves Theorem 6, 2.(a) $\implies$ 2.(b).

Proof of Theorem 6, 2.(b) $\implies$ 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 $(U(n) \,\|\, SYS_I)$ can only fail iff at least one of the following conditions are true:

1. An execution diverges, before success could be signalled:
   $(\exists\, s : Traces(U(n) \,\|\, SYS_I) \bullet \neg\, \langle w \rangle \;\mathbf{in}\; s \wedge s \in Div(U(n) \,\|\, SYS_I)$.

2. An execution produces a trace failure:
   $(\exists\, s : Traces(U(n) \,\|\, SYS_I) \bullet \#front(s) < n \wedge last(s) \in [E_I/s]^0 \setminus [ASYS_I/s]^0)$.

3. An execution blocks in the third $\Box$-branch of $U(n,s)$:
   $(\exists\, s : Traces(U(n) \,\|\, SYS_I);\; R, X : \mathbb{P}\, I \bullet \#s < n-1 \wedge (s,X) \in Fail(SYS_I) \wedge$
   $A(s) \neq \emptyset \wedge R \in R(s) \wedge [(E \,\|\, ASYS)_I/s]^0 \setminus R \cap [SYS_I/s]^0 \setminus X = \emptyset)$.

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

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

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

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

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

For failure condition 4, observe that $(s, R \cup X)$ is a failure of $(E \,\|\, SYS)_I$, so at least one execution of $(U_C(s, A) \,\|\, (E \,\|\, SYS)_I)$ will fail. However, $(E \,\|\, ASYS)_I \; \underline{must} \; U_C(s, A)$ holds since $A \in A(s)$. Therefore $U_C(s, A)$ is contained in $\mathcal{H}_C((E \,\|\, ASYS)_I)$. Again, this implies that $(E \,\|\, SYS)_I$ does not refine $(E \,\|\, ASYS)_I$ 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 6, 2.(b) $\Longrightarrow$ 2.(a).

Proof of Theorem 6, 1. Choosing $SYS_I = ASYS_I$ and noting that $\sqsubseteq_{FD}$ is reflexive, part 1. of the theorem is a trivial consequence of part 2., (b) $\Longrightarrow$ (a). This completes the proof of Theorem 6.
$\square$

Proof of Theorem 7.
In analogy to Theorem 4, $\mathcal{D}(X)$ applies test cases ordered by the length of the traces. Since by Theorem 6 $U(n)$ has the same capabilities to detect failures as the Hennessy test cases $U_S(s, a)$, $U_C(s, A)$ with $\#s \leq n$, Theorem 4 implies that $\mathcal{D}(X)$ is trustworthy for $\sqsubseteq_{FD}$-test.
$\square$