From Testing Theory to Test Driver Implementation

Jan Peleska and Michael Siegel

JP Software-Consulting* and Christian-Albrechts-Universität zu Kiel**

Abstract. In this article we describe the theoretical foundations for the VVT-RT test system *(Verification, Validation and Test for Reactive Real-Time Systems)* which supports automated test generation, test execution and test evaluation for reactive systems. VVT-RT constructs and evaluates tests based on formal CSP specifications [6], making use of their representation as labelled transition systems generated by the CSP model checker FDR [3]. The present article provides a sound formal basis for the development and verification of high-quality test tools: Since, due to the high degree of automation offered by VVT-RT, human interaction becomes superfluous during critical phases of the test process, the trustworthiness of the test tool is an issue of great importance. The VVT-RT system will therefore be formally verified so that it can be certified for testing safety-critical systems. The present article represents the starting point of this verification suite, where the basic strategies for test generation and test evaluation used by the system are formally described and verified. VVT-RT has been designed to support automation of both untimed and real-time tests. The present article describes the underlying theory for the untimed case. Exploiting these results, the concepts and high-level algorithms used for the automation of real-time tests are described in a second report which is currently prepared [14]. At present, VVT-RT is applied for hardware-in-the-loop tests of railway and tramway control computers.

Keywords: CSP — FDR — may tests — must tests — reactive systems — refinement — test evaluation — test generation

1 Introduction

Design, execution and evaluation of trustworthy tests for safety-critical systems require considerable effort and skill and consume a large part of today's development costs for software-based systems. It has to be expected that with conventional techniques, the test coverage to be required for these systems in the near future will become technically unmanageable and lead to unacceptable costs. This hypothesis is supported by the growing complexity of applications

* Goethestraße 26, D-24116 Kiel, e-mail: jap@informatik.uni-kiel.d400.de

** Institut für Informatik und Praktische Mathematik, Preusserstrasse 1-9, 24105 Kiel, Germany, Email: mis@informatik.uni-kiel.d400.de

and the increasingly strict requirements of certification authorities with respect to the verification of safety issues. For these reasons methods and tools helping to automize the test process gather wide interest both in industry and research communities. *"Serious"* testing – not just playing around with the system in an unsystematic way – always has to be based on some kind of specification describing the desired system behaviour at least for the situations covered by the test cases under consideration. As a consequence, the problem of test automation is connected to formal methods in a natural way, because the computer-based design and evaluation of tests is only possible on the basis of formal specifications with well-defined semantics.

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

The idea to apply the theoretical results about testing in process algebras to practical problems was first presented by Brinksma, with the objective to automize testing against LOTOS specifications. His concept has been applied for the automation of *OSI conformance tests*; see [1] for an overview. Today, testing against different types of formal specifications has gained wide interest both for engineers responsible for the quality assurance of safety-critical systems and in the formal methods community: To name a few examples, Gaudel [4] investigates testing against *algebraic specifications*, Hörcher and Mikk in collaboration with the author [7, 8, 9] focus on the automatic test evaluation against *Z specifications* and Müllerburg [11] describes test automation in the field of *synchronous languages.*

Rather than presenting a new testing theory for reactive systems, we investigate how to construct *implementable* and *provably correct test drivers* on the basis of results from testing theory. Our approach is based on the untimed CSP process algebra and uses Hennessy's testing methodology [5] as starting point. To apply the concepts in practice, the VVT-RT tool *(Validation, Verification and Test for Reactive Real-Time Systems)* offers the following possibilities:

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

Typical applications addressed by our approach are systems with discrete interfaces and an emphasis on possibly complex control functionality. Examples are railway control systems, telephone switching systems and network protocols. At present the VVT-RT system is used for the test of computers controlling components of railway interlocking systems. The first application was the automized

test of a PLC system controlling signals, traffic lights and train detection sensors for a tramway crossing, documented in [2, 12, 13]. VVT-RT makes use of the model checker FDR developed by Formal Systems Ltd [3].

This article focuses on theoretic results that are essential for the trustworthy practical application of our testing approach. Examples, industrial applications and a summary of the benefits to be expected from such a test automation concept are described in [16].

The article is structured as follows. In Section 2 we introduce notations and conventions used in subsequent sections. Section 3 introduces transition graphs and results from Hennessy's testing theory. Section 4 contains the main results, where we investigate implementable, minimal test classes and trustworthy test drivers. The full proofs of the theorems discussed in this paper are contained in a technical report [15] which may be obtained from the authors. Section 5 contains conclusions.

2 Preliminaries

2.1 CSP Operators, Semantics and Refinement

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

Tests and test drivers will be specified in the process algebraic framework of *Communicating Sequential Processes (CSP)* [6]. We use the following set of *CSP* operators: *STOP* (deadlock process), *SKIP* (terminating process), $\rightarrow$ (prefixing), $\sqcap$ (internal choice), $\Box$ (external choice), $\|$ (parallel composition with synchronization on common events), $|||$ (interleaving operator without synchronization), $\setminus$ (hiding), and $\frown$ (interrupt). Operator $(x : \{a_1, \ldots, a_n\} \rightarrow P(x))$ abbreviates $a_1 \rightarrow P(a_1) \Box \ldots \Box a_n \rightarrow P(a_n)$, and $\sqcap_{x:\{a_1,\ldots,a_n\}}(x \rightarrow P(x))$ is an abbreviation for $a_1 \rightarrow P(a_1) \sqcap \ldots \sqcap a_n \rightarrow P(a_n)$. As basic programming operators we use **if then else**, $c?$ (input from channel c) and $g\&B$ (guarded command, g is the guard and B the body).

For the specification of recursive processes we use sets of recursive equations rather than an explicit μ–operator. The alphabet of a process P is denoted by $\alpha(P)$.

We use the standard semantics for *CSP* processes: $Traces(P) \subseteq \alpha(P)^*$ (trace semantics), $Fail(P) \subseteq \alpha(P)^* \times \mathbb{P}\,\alpha(P)$ (failure semantics, $\mathbb{P}$ denotes the power set operator), and $Div(P) \subseteq \alpha(P)^*$ (divergences of process P). The elements $s \in Traces(P)$ are the observable traces generated by P. A failure $(s, A) \in Fail(P)$ records the fact that process P may refuse to engage in any action of set A after having performed trace s. Due to nondeterminism there may be several $A, A' \subseteq \alpha(P)$ with $(s, A) \in Fail(P)$ and $(s, A') \in Fail(P)$. The *refusals* of a process P are defined by $Ref(P) =_{df} \{A \mid (\langle\,\rangle, A) \in Fail(P)\}$, where $\langle\,\rangle$ denotes the empty trace. A divergence $s \in Div(P)$ denotes the situation that process P may diverge after having engaged in trace s. Diverging processes show completely unpredictable behaviour, denote by *CHAOS* in *CSP*.

Infinite behaviours are defined as limites of prefix closed sets of finite behaviours [6, p. 132]. We use the standard fixpoint semantics for recursive processes. The set $maxTraces(P)$ denotes the union of the set of all terminated behaviours and the set of infinite behaviours of P.

In this paper we consider the following refinement relations:

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

For $x \in \{T, F, FD\}$ we define process equivalence $P =_x Q$ by
$P \sqsubseteq_x Q \wedge Q \sqsubseteq_x P$.

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

2.2 Alternative Refinement Definitions

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

1. **Safety:** The implementation only generates traces alowed by the specification. This corresponds to the notion of trace refinement:

   $$SPEC \sqsubseteq_S IMP \text{ iff } Traces(IMP) \subseteq Traces(SPEC)^3$$

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

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

[3] We have introduced a new subscript for trace to indicate the correspondence to the safety notion

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

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

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

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

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

The notion of robustness, introduced in [1], can also be expressed as $IMP \sqsubseteq_T SPEC$. This relation has not received much attention in the literature about CSP refinement, though it is a common requirement in practical applications: For example, robustness covers the situation where the specification contains nondeterminism for exception handling. Failures refinement only requires that every guaranteed behaviour of the specification will also be performed by the implementation. Robustness additionally requires that exceptional behaviours of the specification are also covered by the implementation.

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

Lemma 1.

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

□

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

3 Transition Graphs and Test Classes

In this section we describe an implementable encoding of the semantics of *CSP* processes by means of transition graphs. Afterwards we discuss those results of Hennessy's testing theory [5] that are relevant for the development of implementable test drivers.

3.1 Transition Graphs

Automated test generation will be performed by mechanized analysis of the specification, which results in a choice of traces and possible continuations to be exercised as test cases on the target system. *Automated test evaluation* will be performed by observing traces and their continuations in the target system and checking mechanically, if these behaviours are correct with respect to the specification. Obviously, these tasks are fundamentally connected to the problem of mechanized *simulation* of the specification which is in general based on the following theorem [5, p. 94].

Theorem 2 (Normal Form Theorem). *Let P be a CSP process, interpreted in the failures-divergence model.*

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

□

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

With these general ideas in mind, the first problem to solve is how to retrieve the semantic representation – i. e., the failures and divergences – of a specification written in CSP syntax. This has been solved by Formal Systems Ltd and implemented in the FDR system [3], for the subset of CSP specifications satisfying:

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

Under these conditions the CSP specification may be represented as a *labelled transition system* [10] which can be encoded as a *transition graph* with only a finite number of nodes and edges. Basically, the nodes of this directed graph are constructed from Hennessy's *Acceptance Tree* representation [5] by identifying

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

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

3.2 Test Classes

Tests to Characterize Refinement In this section we recall results of Hennessy's testing theory [5] that are relevant for the construction of the test drivers in Section 4.3.

Hennessy introduced processes U, so-called *experimenters*, with $\alpha(SPEC) = \alpha(U) \setminus \{w\}$, where w is a specific event denoting successful execution of the experiment which consists of U running in parallel with the process to be tested[4]. Experimenters coincide with our notion of *test cases*, so we will only use the latter term. An execution of the test case U for the test of some system P is a trace $s \in Traces(P \parallel U)$. The execution is successful if $\langle w \rangle$ **in** s. Depending on U and P, two satisfaction relations may be distinguished with respect to the outcome of test executions:

Definition 3. For a process P and an associated test case U we say

1. $P \underline{may}\ U \equiv_{df} (\exists\, s : Traces(P \parallel U) \bullet \langle w \rangle \textbf{ in } s)$
2. $P \underline{must}\ U \equiv_{df} (\forall\, s : maxTraces(P \parallel U) \bullet \langle w \rangle \textbf{ in } s)$

□

[4] In [5] also another local experimenter event '1' has been introduced which enables the experimenter to control the course of a test execution. However, for the specific Hennessy test classes referenced in this article, this event is not needed.

P *may* U holds if there exists at least one successful execution of $(P \,||\, U)$. Only if every execution of $(P \,||\, U)$ leads to success P *must* U holds.

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

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

Definition 4. Let U be a test case.

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

□

A main result of [5] is the definition of test classes which detect exactly the failures introduced in the previous definition.

Definition 5. For a given specification $SPEC$, let $s \in \alpha(SPEC)^*$, $a \in \alpha(SPEC)$, and $A \subseteq \alpha(SPEC)$. The class of *Hennessy Test Cases* is defined by the following collection of test cases:

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

$$U_S(s,a) =_{df} \textbf{if } s = \langle\,\rangle \textbf{ then } (w \rightarrow SKIP \,\Box\, a \rightarrow SKIP) \textbf{ else } (w \rightarrow SKIP \,\Box\, (head(s) \rightarrow U_S(tail(s), a))$$

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

$$U_C(s,A) =_{df} \textbf{if } s = \langle\,\rangle \textbf{ then } (a : A \rightarrow w \rightarrow SKIP) \textbf{ else } (w \rightarrow SKIP \,\Box\, head(s) \rightarrow U_C(tail(s), A))$$

3. *Divergence Tests* $U_D(s)$*:*

$$U_D(s) =_{df} \textbf{if } s = \langle\,\rangle \textbf{ then } w \rightarrow SKIP \textbf{ else } (w \rightarrow SKIP \,\Box\, head(s) \rightarrow U_D(tail(s)))$$

4. *Robustness Tests* $U_R(s)$:

$$U_R(s) =_{df} \textbf{if } s = \langle\,\rangle \textbf{ then } w \rightarrow SKIP \textbf{ else } head(s) \rightarrow U_R(tail(s))$$

□

Definition 5 is motivated by the following lemma:

Lemma 6.

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 *characterize* the associated failure types: If $s^\frown\langle a\rangle \notin Traces(P)$ then P *must* $U_S(s, a)$ follows. Analogous results hold for $U_C(s, A)$, $U_D(s)$, $U_R(s)$. However, we are less interested in this property, because test cases of practical relevance should be able to detect more than one failure type during test execution.

In our context $s \in Div(P)$ means $P/s = CHAOS$ in the sense of [6], that is, P/s may both *diverge internally (livelock)* and produce and refuse arbitrary *external* events. The tests $U_D(s)$ have been designed by Hennessy to detect internal divergence only. Conversely, the tests $U_S(s, a)$ and $U_C(s, A)$ can detect external chaotic behaviour but cannot distinguish internal divergence from deadlock. However, using the three test classes together enables us to distinguish deadlock, livelock and external chaotic behaviour. Note that P *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 *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.

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

Theorem 7.

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

□

If $SPEC \sqsubseteq_D IMP$ holds, the four implications of the theorem become equivalences. Theorem 7 shows that only *requirements-driven* test design is needed: It is only necessary to execute test cases that will succeed for the specification. Due to possible nondeterminism in $SPEC, IMP$ and U the properties covered by Theorem 7 cannot be verified by means of black-box tests alone, because they require the analysis of *every* possible execution of $SPEC \,||\, U$ and $IMP \,||\, U$. Therefore a *test monitor* collecting information about the executions performed so far is, in general, unavoidable. Note, that this is no disadvantage of the defined classes of tests but inherent in every testing approach that is sensitive to nondeterminism.

4 Minimal Test Classes and Test Drivers

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

4.1 Admissible Tests

First of all we characterize a class of tests that is particularly well-suited for implementation. These tests satisfy the following requirements: 1) If the test execution is successful success will be indicated within a bounded number of events, 2) as test drivers have to know when a test execution has been successfully completed, these tests perform a termination event after signalling success, 3) success is signalled at most once during a test execution, and 4) the tests can be successfully passed (according to the *must* interpretation) by at least one process.

This leads to the following definition:

Definition 8. An *admissible test case* for the test against $SPEC$ is a CSP process U satisfying

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

$$
\begin{aligned}
&S_U(s,R) \equiv \\
&\quad (\exists\, n \in \mathbb{N} \bullet \forall\, s \in \mathit{Traces}(SPEC) \bullet \forall\, R \in \mathit{Ref}(SPEC/s) \bullet \\
&\qquad w \in [U/s]^0 \Rightarrow \\
&\qquad\quad w \notin R \wedge \#s \leq n \wedge \neg\, (\langle w \rangle \ \mathbf{in}\ s) \wedge U/s^\frown\langle w \rangle = \mathit{SKIP})
\end{aligned}
$$

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 1. The test case $U = a \rightarrow SKIP \,\Box\, b \rightarrow (w \rightarrow SKIP \sqcap U)$ would not be admissible in the sense of Definition 8, because it is uncertain whether success will be indicated after event b.
□

Example 2. The test case $U = a \rightarrow w \rightarrow SKIP \sqcap STOP$ would not be admissible in the sense of Definition 8, because no process can satisfy U as a $\underline{must}$-test.
□

Example 3. The test case

$$U = \sqcap_{n:\mathbb{N}} U(n)$$
$$U(n) = (n > 0) \& a \rightarrow U(n-1) \,\Box\, (n = 0) \& w \rightarrow SKIP$$

would be well-defined in the infinite traces model of Roscoe and Barret [17], and $P \underline{must}\ U$ holds for process $P = a \rightarrow P$. 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 8, because no global upper bound exists after that every execution of $(P \,||\, U)$ would show success.
□

Lemma 9. *The Hennessy tests specified in Definition 5 are admissible in the sense of Definition 8.*
□

4.2 Minimal Test Classes

When performing a test suite to investigate the correctness properties of a system, a crucial objective is to perform a *minimal* number of test cases. The following definition specifies minimal sets of Hennessy test, which are still trustworthy in the sense that if the implementation passes these tests then it is a refinement of the specification w.r.t. the currently chosen semantics.

Definition 10. For a given specification $SPEC$, we define the following collections of test cases:

1. $\mathcal{H}_S(SPEC) = \{ U_S(s, a) \mid s \in Traces(SPEC) - Div(SPEC) \wedge a \notin [SPEC/s]^0 \}$
2. $\mathcal{H}_C(SPEC) = \{ U_C(s, A) \mid s \in Traces(SPEC) - Div(SPEC) \wedge$
$A \subseteq [SPEC/s]^0 \wedge$
$(\forall R : Ref(SPEC/s) \bullet A \not\subseteq R) \wedge$
$(\forall X : \mathbb{P}\, A - \{A\} \bullet (\exists R : Ref(SPEC/s) \bullet X \subseteq R)) \}$

3. $\mathcal{H}_D(SPEC) = \{U_D(s) \mid s \in Traces(SPEC) - Div(SPEC) \wedge$
$(\forall u : Traces(SPEC) - Div(SPEC) \bullet$
$s \leq u \wedge [SPEC/u]^0 = \varnothing \Rightarrow s = u)\}$
4. $\mathcal{H}_R(SPEC) = \{U_R(s) \mid s \in Traces(SPEC) \wedge$
$(\forall u : Traces(SPEC) \bullet$
$s \leq u \wedge [SPEC/u]^0 = \varnothing \Rightarrow s = u)\}$

□

The following theorems state that in order to characterize the refinement notions addressed by Theorem 7, it suffices already to exercise the tests specified in Definition 10 on the implementation. Compared to the full set of Hennessy tests, defined for *all* sequences $s \in \alpha(P)^*$ of events and sets $A \subseteq \alpha(P)$, this represents a considerable reduction of the test cases to be considered.

Theorem 11. *If*

$$\mathcal{H}(SPEC) =_{df} \mathcal{H}_S(SPEC) \cup \mathcal{H}_C(SPEC) \cup \mathcal{H}_D(SPEC) \cup \mathcal{H}_R(SPEC)$$

for a given specification SPEC, then SPEC must U holds for all $U \in \mathcal{H}(SPEC)$.
□

Theorem 12. *Given SPEC and the corresponding test classes* $\mathcal{H}_x(SPEC), x \in \{S, C, D, R\}$, *the following properties hold:*

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

□

This theorem shows that for terminating systems, refinement properties can be verified by performing a finite number of tests. (Note, that all processes have only *finite* internal nondeterminism.)

The definitions of $\mathcal{H}_S, \mathcal{H}_C, \mathcal{H}_D$ indicate further that it is not necessary to perform any tests for traces s after which *SPEC* diverges[5], since in such a case $SPEC/s$ will allow chaotic behaviour which does not restrict the admissible behaviours 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 not necessary to exercise any tests $U_C(s, A)$, if $SPEC/s$ may refuse the full alphabet.

The definitions of $\mathcal{H}_D$ and $\mathcal{H}_R$ are motivated by the fact that for the test of divergence and robustness properties we only have to analyze *maximal* traces:

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

If $SPEC$ terminates or blocks after a trace u, the tests corresponding to proper prefixes of u are covered by $U_D(u)$ and $U_R(u)$, so only the latter are contained in $\mathcal{H}_D$ and $\mathcal{H}_R$ respectively.

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

Theorem 13. *Given SPEC and the corresponding test classes $\mathcal{H}_S, \mathcal{H}_C$, the following properties hold:*

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

□

Theorem 13 shows that $\mathcal{H}_S$ and $\mathcal{H}_C$ are indeed minimal: If one test $U(s, a)$ is removed from $\mathcal{H}_S$, a process with safety failure $s^\frown\langle a\rangle$ could be constructed, for which all the remaining tests would succeed. Removing a test $U_C(s, A)$ from $\mathcal{H}_C$ would admit processes P satisfying the remaining tests without refining $SPEC$ in the failures model. Moreover, the set A cannot be reduced in $U_C(s, A)$ in $\mathcal{H}_C$, since otherwise $SPEC$ would no longer pass this test.

The test collections $\mathcal{H}_D$ and $\mathcal{H}_R$, however, cannot be defined as minimal sets, as soon as $SPEC$ describes a non-terminating system: If $s \in maxTraces(SPEC)$ is an infinite computation of $SPEC$, $\mathcal{H}_D$ and $\mathcal{H}_R$ must contain infinitely many tests associated with prefixes $s_1 < s_2 < s_3 < \ldots$ of s, and each infinite subset of these tests would suffice to verify correct behaviour along s. At least we can state that any $\mathcal{H}_D^0 \subseteq \mathcal{H}_D$ 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.

4.3 Test Drivers

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

Definition 14. 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.
□

We will focus on test drivers of the form

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

with admissible test cases U_i. A test driver of this type will execute the test cases in a certain order $U_1, U_2, \ldots$; one test case at a time and with only one copy of the target system $X = IMP$ running. As soon as a test case signals success w, the execution will be interrupted. An input $monitor?next$ will be required from a process monitoring the test coverage achieved so far with the actual test U_i[6]. If the monitor signals $next = true$, the next test case U_{i+1} will be performed, otherwise U_i will be repeated. If U_i is a *may*-test, *next* is always set to *true*.

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

Definition 15. Let $\mathcal{D}(X)$ be a test driver for the test against *SPEC*, performing test cases of a collection $\mathcal{U}$ in the order $U_1, U_2, U_3, \ldots$. Let $\sqsubseteq \in \{\sqsubseteq_T, \sqsubseteq_F, \sqsubseteq_{FD}, \sqsubseteq_R\}$. Then $\mathcal{D}(X)$ is called *trustworthy for* $\sqsubseteq$*-test against SPEC*, iff the following conditions hold:

1. $\mathcal{U}$ contains a subset $\mathcal{U}_{\dot{\sqsubseteq}}$ which characterizes $\sqsubseteq$-refinement against *SPEC*.
2. For every safety-, requirements coverage-, divergence- or robustness-failure violating $\sqsubseteq$, there exists an $n \in \mathbb{N}$ such that $U_n \in \mathcal{U}_{\dot{\sqsubseteq}}$ can detect this failure in the sense of Definition 4.

□

Definition 15 covers the intuitive understanding of trustworthiness in a formal way: whenever a fault may occur for *IMP*, 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.

Theorem 16.

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

applying the tests $U \in \mathcal{H}$ *according to Definition 10, ordered by the length of the defining traces, is trustworthy for* $\sqsubseteq_{FDR}$*-refinement.*
□

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

[6] The implementation of test monitors is not addressed in this paper.

Test Drivers for Reactive Systems The testing methodology presented so far will now be specialized on the development of test drivers for the automated test of *reactive systems.*

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 paradigm 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 have to *test* the target system behaviour while simultaneously *simulating* the operational environment.

To formalize the notion of an operational environment we 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 interface events. Therefore the specification consists of $\mathcal{E}(ASYS)$ with all events apart from I concealed. The implementation can be described by the term

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

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

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

Definition 17. A *standard configuration* $(E, ASYS, SYS, I)$ *(for reactive systems)* consists of CSP processes $E, ASYS, SYS$ and a set I of events such that $I = \alpha(E) \cap \alpha(ASYS) = \alpha(E) \cap \alpha(SYS)$. Context $\mathcal{E}_0(X) = (E \,||\, X)$ is called the *environment.* $SPEC = \mathcal{E}_0(ASYS) \setminus (\alpha(\mathcal{E}_0(ASYS)) - I)$ is called the *specification,* and $IMP = \mathcal{E}_0(SYS) \setminus (\alpha(\mathcal{E}_0(SYS)) - I)$ the *implementation.* For $\sqsubseteq \in \{ \sqsubseteq_T, \sqsubseteq_F, \sqsubseteq_{FD}, \sqsubseteq_{FDR}, \sqsubseteq_R, \sqsubseteq_C, \sqsubseteq_D \}$, a standard configuration is called $\sqsubseteq$*-correct,* if $SPEC \sqsubseteq IMP$ holds.
□

In the following we use the abbreviation $P_I = P \setminus (\alpha(P) \setminus I)$. Note that in a standard configuration $(E \,||\, ASYS)_I = (E_I \,||\, ASYS_I)$ and $(E \,||\, SYS)_I = (E_I \,||\, SYS_I)$ holds, because the hiding operator distributes through $||$, if none of the interface events shared between the parallel components are concealed [6, p. 112].

4.4 A Trustworthy $\sqsubseteq_{FD}$-Test Driver for Reactive Systems

Now we are prepared to state the main result of this article, an implementable test driver that is trustworthy for $\sqsubseteq_{FD}$-refinement. The test driver uses test cases derived from the Hennessy Test Cases introduced in Definition 10 and simultaneously simulates the operational environment of the process to be tested. The properties of these test cases are formally expressed by Theorem 18. Their main advantage when compared to the Hennessy Test Cases is that they allow to investigate safety, requirements coverage and non-divergence at the same time, while the Hennessy Cases require to perform different test suites for each correctness feature. Therefore our test cases are more efficient in practical applications.

Theorem 18. *Let $(E, ASYS, SYS, I)$ be a standard configuration of a reactive system. Define a collection $\mathcal{U} = \{U(n) \mid 0 \leq n\}$ of test cases by*

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

$$\begin{aligned} U(n,s) = (e : ([E_I/s]^0 \setminus [ASYS_I/s]^0) &\rightarrow \dagger \rightarrow SKIP) \\ \Box\ & \\ (\textbf{if } \#s < n\ & \\ \textbf{then } \sqcap_{R:Ref(E_I/s)} & U(n,s,[E_I/s]^0 \setminus R) \\ \textbf{else } (\textbf{if } A(s) = \varnothing\ & \\ \textbf{then } & (w \rightarrow SKIP) \\ \textbf{else } \sqcap_{R:Ref(E_I/s),A:A(s)} & U(n,s,A \setminus R))) \end{aligned}$$

$$\begin{aligned} U(n,s,M) = (M = \varnothing)\&(w \rightarrow SKIP)\ & \\ \Box\ & \\ (e : M \rightarrow (\textbf{if } e \in [ASYS_I/s]^0\ & \\ \textbf{then } (\textbf{if } \#s = n\ & \\ \textbf{then } & (w \rightarrow SKIP) \textbf{ else } U(n, s^\frown\langle e\rangle)) \\ \textbf{else } & (\dagger \rightarrow SKIP))) \end{aligned}$$

where

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

Then

1. *If SYS_I must $U(n)$ for all test cases in $\mathcal{U}$, then $(E \,||\, ASYS)_I \sqsubseteq_{FD} (E \,||\, SYS)_I$ follows.*
2. *If $(E \,||\, ASYS)_I \sqsubseteq_{FD} (E \,||\, SYS)_I$ and $Div(SYS) = \varnothing$ then SYS_I must $U(n)$ for all test cases in $\mathcal{U}$.*
3. *If $Div(ASYS) = \varnothing$ then $ASYS_I$ must $U(n)$ for all test cases in $\mathcal{U}$.*
4. *For all $n \in \mathbb{N}$, test $U(n)$ is admissible.*

□

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, in parallel with a combination of test cases $U_S(s,a)$ and $U_C(s,A)$. $U(n,s)$ 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 failure of the target system SYS. Such a safety failure will be indicated by a special event $\dagger$, if the target system does not diverge before indication becomes possible. Note that the first alternative $(e : ([E_I/s]^0 \setminus [ASYS_I/s]^0) \rightarrow \dagger \rightarrow SKIP)$ in the definition of $U(n,s)$ is redundant, since a safety failure $s^\frown\langle a\rangle$ would also be detected by the tests $U(n)$, $n \geq \#s$ in the last branch of a process $U(n,s,M)$ satisfying $a \in M$. However, for practical reasons it is desirable to detect safety violations as soon as possible, therefore $U(n)$ never refuse a safety failure which might be accepted by the environment E in the actual state of the test execution. As long as $\#s < n$, $U(n,s)$ will behave as E_I/s with respect to the refusal of events. For $\#s = n$, $U(n,s)$ will only admit events contained in a minimal acceptance set $A \in A(s)$, so that $U(n)$ can detect requirement coverage failures of SYS occurring after traces of length n, when running in environment E. The nondeterministic $\sqcap$-operator used in the definition of $U(n,s)$ shows where internal decisions with respect to the control of the test execution 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.

Using the results of Theorem 16 and Theorem 18, now we can state that test drivers using the test cases $U(n)$ have the desired correctness properties:

Theorem 19. *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*

$$\mathcal{D}(X) = (n := 0);\ *\,(U(n) \,||\, X^\frown\langle w \rightarrow monitor?next \rightarrow (\textbf{if } next \textbf{ then } i := i+1;\ SKIP \textbf{ else } SKIP)\rangle);$$

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

□

5 Conclusion

This article focused on the development of test drivers performing automized generation, execution and evaluation of tests for reactive systems against CSP specifications. Given a correctness relation between specifications and implementations, a test driver should be capable of

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

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

The work presented in this article reflects a "building block" of a joint enterprise of ELPRO LET GmbH, JP Software-Consulting, Bremen University and Kiel University in the field of test automation for reactive real-time systems. An overview of these activities is given in [16].

References

1. E. Brinksma: A theory for the derivation 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.
2. ELPRO LET GmbH: *Programmablaufplan - Bahnübergang.* ELPRO LET GmbH (1994).
3. Formal Systems Ltd.: *Failures Divergence Refinement.* User Manual and Tutorial Version 1.4. Formal Systems (Europe) Ltd (1994).
4. 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 (1995).
5. M. C. Hennessy: *Algebraic Theory of Processes.* MIT Press (1988).
6. C.A.R. Hoare. *Communicating sequential processes.* Prentice-Hall International, Englewood Cliffs NJ (1985).
7. H. M. Hörcher and J. Peleska: The Role of Formal Specifications in Software Test. Tutorial, held at the FME '94.
8. H. M. Hörcher: Improving Software Tests using Z Specifications. To appear in J. P. Bowen and M. G. Hinchey (Eds.): *ZUM '95: 9th International Conference of Z Users,* LNCS, Springer (1995).
9. E. Mikk: Compilation of Z Specifications into C for Automatic Test Result Evaluation. To appear in J. P. Bowen and M. G. Hinchey (Eds.): *ZUM '95: 9th International Conference of Z Users,* LNCS, Springer (1995).
10. R. Milner: *Communication and Concurrency.* Prentice-Hall International, Englewood Cliffs NJ (1989).

11. M. Müllerburg: Systematic Testing: a Means for Validating Reactive Systems. In *EuroSTAR'94: Proceedings of the 2nd European Intern. Conf. on Software Testing, Analysis&Review.* British Computer Society, (1994).
12. J. Peleska: *Bahnübergangssteuerung Straßenbahn — ELPRO LET GmbH: Prüfspezifikation für formale Verifikation und automatisierte Testdurchführung.* JP Software-Consulting (1994).
13. J. Peleska: *Bahnübergangssteuerung Straßenbahn — ELPRO LET GmbH: Sicherheitsspezifikation und BUE-Spezifikation.* JP Software-Consulting (1994).
14. J. Peleska: *Trustworthy Tests for Reactive Systems — Automation of Real-Time Testing.* In preparation, JP Software-Consulting (1995).
15. J. Peleska and M. Siegel: From Testing Theory to Test Driver Implementation. Technical Report, JP Software-Consulting (1995).
16. J. Peleska: Test Automation for Safety-Critical Systems: Industrial Application and Future Developments. To appear in *Proceedings of the Formal Methods Europe Conference, FME '96.*, LNCS, Springer (1996).
17. A. W. Roscoe and G. Barret: Unbounded Nondeterminism in CSP. In *MFPS '89*, volume LNCS 298, Springer-Verlag, (1989).