Conformance Testing With State Mapping

Vadim Mutilin
Institute for System Programming of Russian Academy of Sciences
25, B. Kommunisticheskaya, Moscow, Russia
mutilin@ispras.ru

***Abstract*— This work considers conformance testing (functional testing). The main distinction from the other works is the availability of mapping from implementation states to specification ones. This information allows us to simplify test development and to reduce test execution time. We introduce a novel implementation relation called conff and composite test suites. The conff relation minimizes the size of generated test suite. Composite test suite is a compact representation of traditional test suite. Furthermore, it allows to reduce test execution time.**

I. Introduction

Testing is an essential part of modern software engineering. With the development of software engineering methods, the complexity of created systems increases. As result, testing becomes more complex. By estimations of experts, testing takes more than a half of all the time of creation of modern software systems, that enables to speak about testing as about one of the key activities in modern software engineering.

Testing is an execution of implementation with the subsequent checking of results on conformity to the requirements. Depending on the requirements, several kinds of testing are distinguished: conformance testing (functional testing), performance testing, robustness testing, stress testing, reliability testing and availability testing.

There are black box testing and a white box testing. In a black box testing only the information on interaction of system with an environment is accessible. Whereas in a white box testing the information on structure of system is accessible. There are a plenty of gray box testing methods in which the part of the information on system is accessible.

In this work we concentrate on gray box, conformance testing. That distinguishes the work from traditional methods of conformance testing, in which testing is performed by black box methods.

In conformance testing, functional specification and implementation exhibiting some behaviour are given. The purpose of testing is to check the conformance of the given implementation to the specification. Let $SPECS$ be a set of specifications, $IMPS$ be set of implementations, then conformance could be represented as a relation **conforms-to** $\subseteq IMPS \times SPECS$. Expression IUT **conforms-to** M means that IUT is a correct implementation of specification M.

In this work we consider specifications given in the formal notations. However, unlike specifications, implementations are real physical objects, such as pieces of hardware or software. They are considered as objects interacting with the environment, but not amenable to formal reasoning. This makes it difficult to give a formal definition of **conforms-to**. Therefore in order to reason formally about implementations, we assume that any implementation is modelled by formal object $Sys_{IUT} \in MODS$, where $MODS$ is a set of models. This assumption is called the *test hypothesis* [2], [19]. Note that the test hypothesis assumes only that the model exists, but not that it is known a priory.

Thus, the test hypothesis allows to define conformance as a relation between formal models of implementation and specification. This relation is called *implementation relation* **imp** $\subseteq MODS \times SPECS$. Implementation $IUT \in IMPS$ is said to be correct with respect to $M \in SPECS$, IUT **conforms-to** M, if and only if the model of implementation $Sys_{IUT} \in MODS$ is **imp**-related to M: Sys_{IUT} **imp** M.

The behaviour of implementation is investigated by performing experiments with it and observing reactions that the implementation produces to these experiments. A specification of such experiments is called a *test case*, and the process of applying a test to an implementation – *test execution*. Formally we define test execution on the model of implementation, however, according to the test hypothesis, the same behaviour will be observed at test execution on the real implementation.

During test execution, a number of observations are made, e.g. occurring events, stimuli, reactions are recorded. At the end of the test execution we like to interpret the observations in terms of being right or wrong, so we return verdict. If the test confirms the correctness of implementation, then verdict is **pass**, otherwise – **fail**. We say that the test *passes*, if it ends with a verdict **pass**, and otherwise we say that it *fails* .

A set of tests is called a *test suite*. Implementation *passes* a test suite, if it passes all the tests in it. Ideally, we would like to have such test suite that implementation passes it if and only if it conforms to the specification. A test suite with this property is called *complete*. It can distinguish exactly between conforming and non-conforming implementations. Unfortunately, in practice this requirement is very strong: complete tests are usually infinite and consequently not executable. Hence, a weaker requirement is usually posed: the test suite should be *sound*, which means that any correct implementation passes this suite. Probably, that some incorrect implementations also pass the suite, but if implementation does not pass, it indeed does not conform to the specification.

In the conformance testing process there are two main phases: *test generation/writing* and *test execution*. The impor-

tant characteristics of these phases are size of input data for obtaining a test suite and time of test execution.

In automatic test generation, the only input is specification from which tests are generated automatically. As it was noted, in practice we need additional data for test selection to get a test suite practically suitable for execution. For some reasons tests can be created manually. Imperfection of test generation and selection methods concern to such reasons, for example, if selection methods do not allow to get a good-enough test suite, executable in practice. Besides, representation of specification can not allow to perform automatic generation, for example, if the specification is given in the form of preconditions and postconditions representing predicates on states, input and output data.

On test execution phase it can be used both fully prepared test suite and partially prepared test suite from which tests are constructed during execution. Process, in which tests are completely generated at execution, is called on-the-fly generation. Time of execution is interpreted as quantity of the executed interactions with implementation. The more interactions the longer is test execution. Thus, we can recognize two important characteristics of testing process:

1) Size of input data;
2) Time of test execution.

In this work we consider implementations, which provide information about their state. This information can be obtained in different ways: by reading of accessible fields, global variables; by function calls, with the established correctness; by reading data from a database. Besides, it is required to provide the mapping from this information to specification states. This mapping is developed by test developers. In practice it is required to write a function, which first reads the information of an implementation state, and secondly, establishes connection between the information and specification state.

Formally, we consider, that there is a set of mappings $MAPPINGS : SPECS \times MODS \rightarrow FUNC$, where for each specification M and implementation Sys the function $f = MAPPINGS(M, Sys)$ is given, which maps implementation states to specification ones. Note that f can be partially defined, i.e. the corresponding specification state can be defined not for all implementation states.

The purpose of the work is to use the given state mapping from implementation states to specification ones for reduction of size of input data and time of test execution.

This work is organized as follows. Section II introduces labelled transition systems used as formalism for specifications and implementations. Section III gives basic testing concepts for labelled transition systems, such as a test case, a test suite, a test run, passing a test case, passing a test suite and the notions of completeness, soundness and exhaustive of a test suite. Also the implementation relation **conf** is described in this section. The main results of this work are presented in section IV. A novel implementation relation **conff** which uses state mappings given in $MAPPINGS$ is introduced in it. The algorithm for complete test suite generation that is more effective than algorithm in [18] for **conf** is developed. Then, composite test suites are introduced, which are the compact representations of regular test suites. Test cases in these suites are intended for execution in different specification states, not only in the initial state as it is in traditional test suites. Besides reduction of input data size composite test suites allow to reduce test execution time as compared with test suites derived from them. In section V some concluding remarks are given, among which a relation to practice and a future work.

II. Labelled Transition Systems

We use labelled transition systems (LTS) [18] for formalization of the behaviour of systems.

Definition 1: Labelled transition system is a 4-tuple $\langle S, L, T, s_0\rangle$, where S is a set (possibly infinite) of states, L is a set of labels, T is the transition relation $T \subseteq S \times (L \cup \{\tau\}) \times S$, s_0 is the initial state.
The labels in L represent the observable interactions of a system. The special label $\tau \notin L$ represents unobservable, internal action.

We denote the class of all labelled transition systems over L as $LTS(L)$. We restrict $LTS(L)$ to labelled transition systems that do not have cycles composed of internal actions.

The set of all finite sequences in alphabet L is denoted by L^*, with ϵ denoting the empty sequence. If $\sigma_1, \sigma_2 \in L^*$, then $\sigma_1 \cdot \sigma_2$ is the concatenation of σ_1 and σ_2. With $|\sigma|$ we denote the length of the sequence. Some additional notations are introduced in definitions 2 and 3.

Definition 2: Let $p = \langle S, L, T, s_0\rangle$ be a labelled transition system, $s, s' \in S$, $\mu_{(i)} \in L \cup \{\tau\}$, $\nu_{(i)} \in L$ and $\sigma \in L^*$.

$$
\begin{array}{lll}
s \xrightarrow{\mu} s' & =_{def} & (s, \mu, s') \in T \\
s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} s' & =_{def} & \exists s_0, \ldots, s_n : s = s_0 \xrightarrow{\mu_1} \ldots \xrightarrow{\mu_n} s_n = s' \\
s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} & =_{def} & \exists s' : s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} s' \\
s \overset{\mu_1 \cdot \ldots \cdot \mu_n}{\not\longrightarrow} & =_{def} & \text{not } \exists s' : s \xrightarrow{\mu_1 \cdot \ldots \cdot \mu_n} s' \\
s \overset{\epsilon}{\Longrightarrow} s' & =_{def} & s = s' \text{ or } s \xrightarrow{\tau \cdot \ldots \cdot \tau} s' \\
s \overset{\nu}{\Longrightarrow} s' & =_{def} & \exists s_1, s_2 : s \overset{\epsilon}{\Longrightarrow} s_1 \xrightarrow{\nu} s_2 \overset{\epsilon}{\Longrightarrow} s' \\
s \overset{\nu_1 \cdot \ldots \cdot \nu_n}{\Longrightarrow} s' & =_{def} & \exists s_0, \ldots, s_n : s = s_0 \overset{\nu_1}{\Longrightarrow} \ldots \overset{\nu_n}{\Longrightarrow} s_n = s' \\
s \overset{\sigma}{\Longrightarrow} & =_{def} & \exists s' : s \overset{\sigma}{\Longrightarrow} s' \\
s \overset{\sigma}{\not\Longrightarrow} & =_{def} & \text{not } \exists s' : s \overset{\sigma}{\Longrightarrow} s'
\end{array}
$$

Definition 3: Let $p = \langle S, L, T, s_0\rangle$ be a labelled transition system and $s \in S$, $\sigma \in L^*$.

1) $traces(s) =_{def} \{\sigma \in L^* \mid s \overset{\sigma}{\Longrightarrow}\}$
2) $init(s) =_{def} \{\nu \in L \mid s \overset{\nu}{\Longrightarrow}\}$
3) s **after** $\sigma =_{def} \{s' \in S \mid s \overset{\sigma}{\Longrightarrow} s'\}$
4) $der(p) =_{def} \{s \mid \exists \sigma \in L^* : s_0 \overset{\sigma}{\Longrightarrow} s\}$
5) p has finite behaviour if $\exists n \in Nat$, such that $\forall \sigma \in traces(s_0) : |\sigma| < n$
6) p is a finite state if $der(p)$ is finite.
7) s is deterministic if for all sequences $\sigma \in L^*$, s **after** σ has at most one element. If $\sigma \in traces(p)$, then we overload s **after** σ to denote this element.

Definition 4: The state is finite if $\forall \mu \in L \cup \{\tau\}$ holds $s \overset{\mu}{\not\rightarrow}$.

Definition 5: State is stable if $s \overset{\tau}{\not\rightarrow}$.
The set of all stable states is denoted by S_δ. Later on, we consider that $s_0 \in S_\delta$.

III. Conformance Testing

Starting point for conformance testing is a specification given in some formal notation and implementation, that is an object interacting with it's environment. We make an assumption that each implementation has a formal model. This assumption is referred to as the test hypothesis.

In this paper labelled transition systems, or any language with underlying semantics of labelled transition systems, are considered as the formal notation for specifications. A model of implementation also considered as labelled transition system.

A. Implementation Relation

Conformance is defined by means of an implementation relation between the models of implementation and the specifications, in our case it is relation **imp** $\subseteq LTS(L) \times LTS(L)$: implementation $Sys \in LTS(L)$ conforms to specification $M \in LTS(L)$ if and only if Sys **imp** M.

Many possibilities for implementation relations have been studied, for example, observation equivalence [12], strong and weak bisimulation equivalence [13], [16], failure equivalence and preorder [7], testing equivalence and preorder [15], failure trace equivalence and preorder [1], generalized failure equivalence and preorder [10], and many others [20], [6].

In this paper we chose the implementation relation **conf** [5]. This relation was extensively studied in the literature, see for example [18].

Definition 6: A deadlock of p in a state s is a trace $\sigma \in L^*$, after which no more observable actions are possible:

$$s \textbf{ after } \sigma \textbf{ deadlocks} =_{def} \exists s' : s \stackrel{\sigma}{\Rightarrow} \text{ and } \quad init(s') = \emptyset$$

Definition 7: The sets of observations, obs and obs' respectively, that an observer $u \in LTS(L)$ of system $p \in LTS(L)$ can make, are given by deadlocks, respectively the traces of the synchronized parallel composition of u and p:

$$obs(u,p) \quad =_{def} \quad \{\sigma \in L^* \mid (u \| p) \textbf{ after } \sigma \textbf{ deadlocks}\}$$
$$obs'(u,p) \quad =_{def} \quad \{\sigma \in L^* \mid (u \| p) \stackrel{\sigma}{\Longrightarrow}\}$$

Definition 8: Let $p \in LTS(L)$, $s \in S$, $\sigma \in L^*$ and $A \subseteq L$, then

$$s \textbf{ after } \sigma \textbf{ refuses } A =_{def} \exists s' : s \stackrel{\sigma}{\Rightarrow} s' \text{ and } \forall \nu \in A : s' \stackrel{\nu}{\not\Rightarrow}$$

Definition 9: Let $M = \langle S^M, L, T^M, s_0^M \rangle$ be a specification and $Sys = \langle S, L, T, s_0 \rangle$ be an implementation.
Sys **conf** $M =_{def} \forall u \in LTS(L) : (obs(u, Sys) \cap traces(s_0^M)) \subseteq obs(u, M)$ and $(obs'(u, Sys) \cap traces(s_0^M)) \subseteq obs'(u, M)$

Statement 1: Sys **conf** M if and only if ($\forall \sigma \in traces(M)$, $\forall A \subseteq L$: Sys **after** σ **refuses** A implies M **after** σ **refuses** A).
The proofs are given in [14], [17].

Figure 1 gives examples of labelled transition systems. p_2 **conf** p_1 does not hold, since p_2 **after** a **refuses** $\{c\}$ not implies p_1 **after** a **refuses** $\{c\}$. It is easy to check that p_3 **conf** p_1, p_5 **conf** p_4, but p_4 **conf** p_5 does not hold.

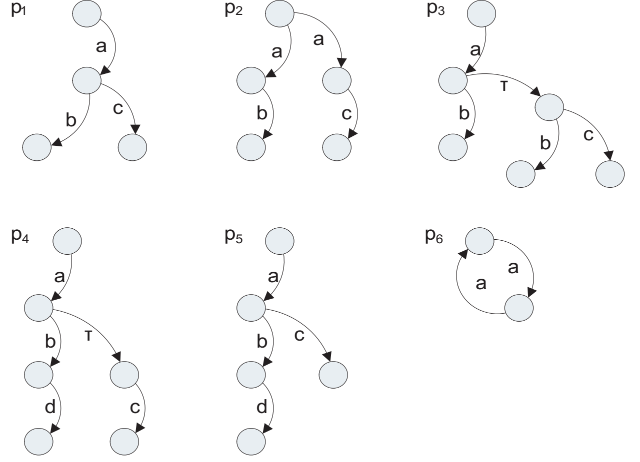

Fig. 1. Examples of labelled transition systems

B. Test Suite

Test cases are derived from the specification, and applied to the implementation. Test cases are constructed in such a way that from the results of applying them it can be concluded whether the implementation conforms to specification.

A test case is a specification of the behaviour of a tester in an experiment to be carried out with an implementation under test. Such behaviour, like other behaviours, can be specified by a labelled transition system. An experiment should last for finite time, so the test case should have finite behaviour. Moreover, a tester executing a test case would like to have as much control as possible, so nondeterminism in a test case is undesirable. To be able to decide about the success of a test a verdict is attached to each state of the test case.

Definition 10: A test case t is a 5-tuple $\langle S^t, L, T^t, \nu, s_0^t \rangle$, such that $\langle S^t, L, L_\tau^t, T^t, s_0^t \rangle$ is a deterministic labelled transition system with finite behaviour, and $\nu : S^t \rightarrow \{pass, fail\}$ is a verdict function.
The class of test cases over action in L is denoted by $LTS_t(L)$. Definitions for $LTS(L)$ is extended to $LTS_t(L)$ by applying them to underlying labelled transition system.

Definition 11: A test suite T is a set of test cases, i.e. a subset of $LTS_t(L)$.

A test case run is modelled by synchronous parallel composition of the test case with an implementation, which continues until the deadlock is reached. This deadlock may occur when test case reaches a final state, or when the composition reaches a state where the actions proposed by the test case cannot be accepted be the implementation. An implementation passes a test run if and only if the verdict of the test case in a deadlock state is pass. Since an implementation can behave nondeterministically different test runs of the same test case can lead to different deadlock states, and hence to different verdicts. An implementation passes a test case if and only if all possible test runs lead to verdict pass.

Definition 12:

1) A *test run* of a test case $t \in LTS_t(L)$ with an implementation $Sys \in LTS(L)$ is a trace of the synchronous

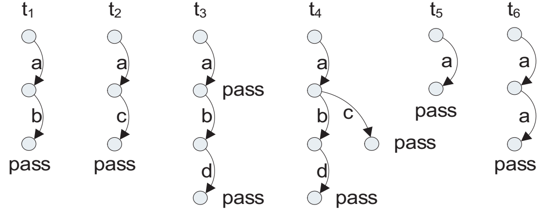

Fig. 2. Examples of test cases

parallel composition of t and Sys leading to a deadlock:

$$\sigma \text{ is a test run of } t \text{ and } Sys =_{def}$$

$$(t \| Sys) \textbf{ after } \sigma \textbf{ deadlocks}$$

2) An implementation Sys *passes* test case t, if all test runs lead to a deadlock states, where verdict is pass:

$$Sys \textbf{ passes } t =_{def}$$

$$(t \| Sys) \textbf{ after } \sigma \textbf{ deadlocks} \text{ implies } \nu(t \textbf{ after } \sigma) = pass$$

3) An implementation Sys *passes* a test suite T, if it passes all test cases $t \in T$:

$$Sys \textbf{ passes } T =_{def} \forall t \in T : Sys \textbf{ passes } t$$

Figure 2 shows examples of test cases. p_4 **not passes** t_1, since $t_1 \| p_4$ **after** a **deadlocks** and $\nu(t_1$ **after** $a) = fail$. p_1 **passes** $\{t_1, t_2\}$, p_2 **not passes** t_1, p_3 **passes** $\{t_1, t_2\}$, p_4 **passes** $\{t_3, t_4\}$, p_5 **passes** $\{t_3, t_4\}$, p_6 **passes** $\{t_5, t_6\}$.

Definition 13: Let M be a specification, **imp** be an implementation relation and T be a test suite, then

$$\begin{aligned} T \text{ is complete} \quad &\text{if} \quad \forall Sys : Sys \textbf{ imp } M \Leftrightarrow Sys \textbf{ passes } T \\ T \text{ is sound} \quad &\text{if} \quad \forall Sys : Sys \textbf{ imp } M \Rightarrow Sys \textbf{ passes } T \\ T \text{ is exhaustive} \quad &\text{if} \quad \forall Sys : Sys \textbf{ imp } M \Leftarrow Sys \textbf{ passes } T \end{aligned}$$

IV. Using State Mapping

In this section we consider specifications and implementations for which the information about correlation between their states is given. We assume that for each specification and implementation there is a mapping from implementation state to specification state. In practice this mapping is accomplished by the procedure which reads state data from the implementation and then constructs the corresponding specification state.

By the semantics, τ action in LTS can be executed at arbitrary time. Hence, reading of state data in the states with this action (unstable states) is complicated. In practice we cannot guarantee that the data will be read in a certain unstable state. In this connection we assume that the mapping exists only for stable implementation states.

Formally, we assume that for each pair (specification, implementation) there is a function $f : S_\delta \to S^M_\delta$, where S_δ is a set of stable implementation states and S^M_δ is a set of stable specification states. By $MAPPINGS$ we refer to a mapping, which for each pair (specification, implementation) contains a corresponding mapping f. Implementation relation depending on $MAPPINGS$ is referred to as **imp**(f). We write Sys **imp**(f) M, if Sys **imp** M with state mapping $f = MAPPINGS(M, Sys)$.

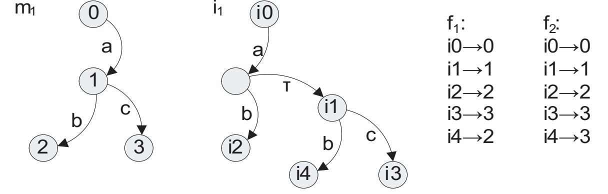

Fig. 3. Specification and implementation with state mappings

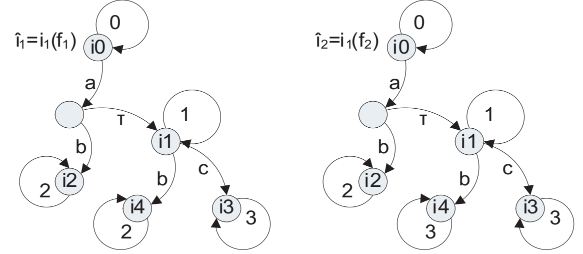

Fig. 4. Implementation supplemented with state mappings

Existence of f allows to supplement system Sys with transitions representing the current specification state. Hence, we define

$$\hat{Sys} =_{def} \langle S, L \cup S^M_\delta, T \cup \hat{T}, s_0 \rangle,$$

$$\hat{M} =_{def} \langle S^M, L \cup S^M_\delta, T^M \cup \hat{T}^M, s^M_0 \rangle,$$

where

$$\hat{T} = \{(s, f(s), s) \mid s \in S_\delta\},$$

$$\hat{T}^M = \{(s^M, s^M, s^M) \mid s^M \in S^M_\delta\}.$$

Let's consider an example of specification and implementation with the given state mapping. Figure 3 shows specification m_1, implementation i_1, and two possible state mappings f_1 and f_2. Figure 4 shows implementations i_2 and i_3, which are the result of supplementation of implementation i_1 with state mappings f_1, f_2 respectively.

A. Implementation Relation conff

Consider implementation relation **conff**, which depends on state mappings given in $MAPPINGS$. We assume that for each specification $M = \langle S^M, L, T^M, s^M_0 \rangle$ and implementation $Sys = \langle S, L, T, s_0 \rangle$ mapping $f : S_\delta \to S^M_\delta$ ($f = MAPPINGS(M, Sys)$) is given.

Definition 14: Let $M = \langle S^M, L, T^M, s^M_0 \rangle$ be a specification, $Sys = \langle S, L, T, s_0 \rangle$ be an implementation, and f be a state mapping.

Sys **conff** $M =_{def}$

1) Sys **conf** M, and
2) $\forall \sigma \in traces(M)$: $s_0 \overset{\sigma}{\Longrightarrow} s$ in Sys, where $s \in S_\delta$ implies $f(s_0) \overset{\sigma}{\Longrightarrow} f(s)$ in M.

Statement 2: Sys **conff** M implies Sys **conf** M.

In example shown in figure 3 i_1 **conff** m_1 holds, if mapping is f_1, and does not hold, if mapping is f_2.

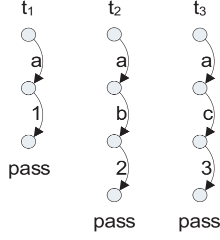

Fig. 5. Test suite example for **conff**

Consider another way of **conff** definition. For implementations supplemented with state mappings the following statement holds.

Statement 3: Sys **conff** M if and only if $\hat{Sys}$ **conf** $\hat{M}$.

From this follows that test suite generation for implementation relation **conff** can be done by test generation algorithm for **conf** described in [18] by applying it to $\hat{M}$.

B. Test Generation for conff

The notions of test case, test suite, test run and passing a test case are the same, with the only difference that as implementation we consider system $\hat{Sys}$, supplemented with state mapping. Furthermore, test cases together with labels from L include labels corresponding to specification states S_δ^M.

In contrast to **conf**, test generation for **conff** becomes simpler because it is stronger: besides inclusion of observable traces and deadlocks it requires coincidence of the corresponding specification states.

Test suite generation algorithm consists in applying construction 1 to all specification traces $traces(M)$.

Construction 1: Let $\sigma = \mu_1 \cdot \ldots \cdot \mu_n \in traces(M)$ is given. Construct a labelled transition system $x = \langle S^x, L, T^x, s_0^x \rangle$ that consists of the only trace $traces(x) = \{\sigma\}$, deterministic for all sequences, i.e. x consists of transitions $s_0^x \xrightarrow{\mu_1} \ldots \xrightarrow{\mu_n} s_n^x$. From x construct test case $t = \langle S^x \cup S_\eta, L \cup S_\delta^M, T^x \cup T_\eta, \nu, s_0^x \rangle$, where function ν on states $s^x \in S^x$, S_η and T_η are as follows.

Let $\tilde{\sigma}$ be a trace leading to s^x, i.e. $s^x = s_0^x$ **after** $\tilde{\sigma}$. Consider two cases.

1) If $t \| M$ **after** $\tilde{\sigma}$ **deadlocks**, then
   a) Assign $\nu(s^x) = fail$
   b) Add to T_η the following set of transitions $\{(s^x, s', new) \mid s' \in S' \cap S_\delta^M,\ new$ – new state$\}$, where S' – reachable specification state, in which deadlock is possible, i.e. $S' = \{s^M \mid (s_0^x, s_0^M) \stackrel{\tilde{\sigma}}{\Longrightarrow} (s^x, s^M)$ and $(s^x, s^M) \not\Longrightarrow\}$.
   c) Enlarge S_η with new states added to T_η, for which we assign $\nu(new) = pass$.
2) Otherwise, assign $\nu(s^x) = fail$.

The example of the test suite generated from all specification traces of m_1 (fig. 3) with applying construction 1, is shown in figure 5.

Theorem 1: Let T be a test suite obtained by applying construction 1 to all specification traces, then T is complete for implementation relation **conff**.

Proof: We devide proof into two parts. To show that T is complete it is sufficient to prove that

1) $\forall \tilde{T} \subseteq T$, $\tilde{T}$ is sound;
2) T is exhaustive.

Prove 1. Suppose that one of the test cases $t \in T$ does not pass on $\hat{Sys}$. Then test case deadlocks in a state s_t: $\nu(s_t) = fail$, σ is it's test run. By construction 1, s_t is not finite state. Hence, there exists $\mu \in L \cup S_\delta^M$: $\hat{Sys}$ **after** σ **refuses** $\{\mu\}$. But by construction 1, $\sigma \cdot \mu \in traces(\hat{M})$. Hence, $\hat{M}$ **after** σ **refuses** $\{\mu\}$ does not hold. From statement 1 follows that $\hat{Sys}$ **conf** $\hat{M}$ does not hold.

Prove 2. It is required to show that if $\forall t \in T$ $\hat{Sys}$ passes t, then Sys **conff** M, i.e. by definition 14 and statement 1 it is sufficient to prove that

1) $\forall \sigma \in traces(M)$, $\forall A \subseteq L$: Sys **after** σ **refuses** A implies M **after** σ **refuses** A
2) $\forall \sigma \in traces(M)$: $s_0 \stackrel{\sigma}{\Longrightarrow} s$ in Sys, where $s \in S_\delta$ implies $f(s_0) \stackrel{\sigma}{\Longrightarrow} f(s)$ in M.

At first we show 2. Let $\sigma \in traces(M)$ and $s_0 \stackrel{\sigma}{\Longrightarrow} s$ in Sys. Let t be a test constructed on the base of σ and $\hat{Sys}$ passes t. By construction 1, $f(s) \in s_0^M$ **after** σ; otherwise, test case finishes with the fail verdict. From this it follows that $f(s_0) \stackrel{\sigma}{\Longrightarrow} f(s)$ in M.

Now we show 1. Let $\sigma \in traces(M)$ and Sys **after** σ **refuses** A, i.e.

$$\exists s' \in Sys \textbf{ after } \sigma : \forall \mu \in A : s' \stackrel{\mu}{\not\Longrightarrow} . \qquad (1)$$

We prove by contradiction. Let M **after** σ **refuses** A does not hold, i.e. $\forall s^M : s_0^M \stackrel{\sigma}{\Longrightarrow} s^M\ \exists \mu \in A : s^M \stackrel{\mu}{\Longrightarrow}$. By proved condition 2, we have $f(s_0) \stackrel{\sigma}{\Longrightarrow} f(s')$. Hence, $\exists \mu \in A : f(s') \stackrel{\mu}{\Longrightarrow}$.

Let t be a test case constructed on the base of $\sigma \cdot \mu$ and $s^t = t$ **after** σ. By condition, $\hat{Sys}$ passes t. By construction 1, the test case state s^t does not have transitions with label $f(s')$, since there is no transition by μ in specification M ($f(s') \stackrel{\mu}{\Longrightarrow}$), and by assumption (1), implementation does not have transition by μ in state s' ($s' \stackrel{\mu}{\not\Longrightarrow}$). Hence $t \| \hat{Sys}$ **after** σ **deadlocks**. But by construction 1, $\nu(s^t) = fail$. This contradicts the fact that implementation passes the test case. ■

Note 1: Test generation algorithm described in [18], generates a test suite such that total number of traces of test cases is no less than the number of specification traces. As distinct from it, algorithm proposed here is restricted to all specification traces.

In example shown in figure 6, proposed test generation algorithm for specification m_2 generates a complete test suite consisting of three test cases t_1, t_2 and t_3. Whereas algorithm in [18] generates a complete test suite consisting of infinite number of test cases. The cause of infinite number is self-loops representing transitions by specification states. However, even

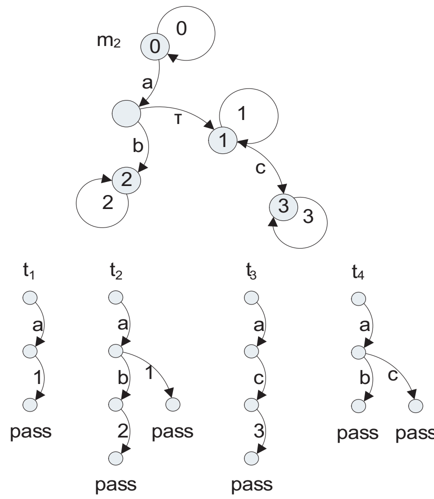

Fig. 6. Comparison of generated test suites

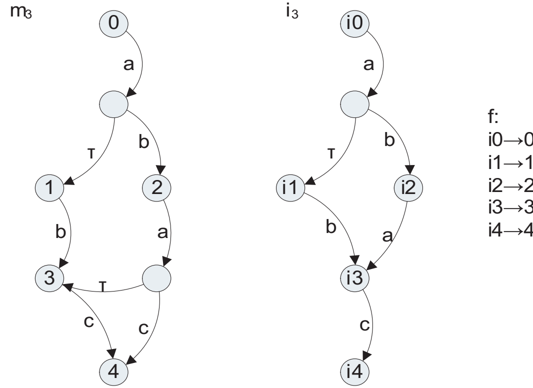

Fig. 7. Specification and implementation with state mapping

if we apply the algorithm to the original system without self-loops, generated test suite will additionally contain test case t_4.

C. Composite Test Suites

We consider test suite T containing test cases that can be executed not only in the initial state, but in arbitrary state as well. For each state s^M with the help of the function $enabled$ we define the possibility of execution of a test case $t \in T$ in this state.

Definition 15: Composite test suite T_e for specification M is a pair $\langle T, enabled\rangle$, where T is a test suite and $enabled : S_\delta^M \rightarrow 2^T$ is a function defining a set of test cases which can be executed for each state $s^M \in S_\delta^M$.

Figure 8 shows composite test suite for specification m_3 in figure 7.

The notions of test run, passing a test case and a test suite with an implementation are the same with the only difference that definitions are made not only for initial state, but for arbitrary state as well.

Definition 16:

1) A *test run* of a test case $t \in LTS_t(L)$ with an

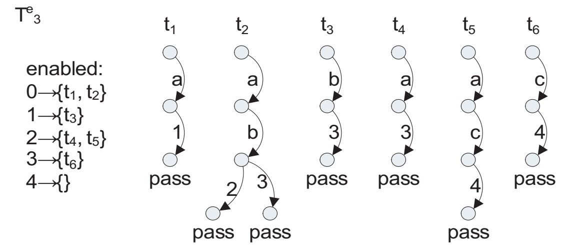

Fig. 8. Example of composite test suite

implementation $Sys \in LTS(L)$ in a state s is a trace of the synchronous parallel composition of t and Sys leading to a deadlock:

$$\sigma \text{ is a test run of } t \text{ and } Sys \text{ in } s =_{def}$$

$$(t \| Sys') \textbf{ after } \sigma \textbf{ deadlocks},$$

where Sys' – system Sys with the initial state s.

2) An implementation Sys *passes test run* σ of a test case t in a state s, if the test run leads to a deadlock, where verdict is pass:

$$Sys \textbf{ passes test run } \sigma \text{ of } t \text{ in } s =_{def}$$

$$\nu(t \textbf{ after } \sigma) = pass$$

3) An implementation Sys *passes* test case t in s, if it passes all test runs in this state:

$$Sys \textbf{ passes } t \text{ in } s =_{def}$$

$$\forall \sigma : Sys \textbf{ passes test run } \sigma \text{ of } t \text{ in } s$$

To check the possibility of execution of test case t in implementation state s we need information about corresponding specification state. We assume that for each specification and implementation we have function f mapping stable implementation states into stable specification states.

Definition 17 introduces additional notations for test case execution in stable states.

Definition 17: Given an implementation $Sys = \langle S, L, T, s_0\rangle$, a mapping f, a test suite $T^e = \langle T, enabled\rangle$, $s_1, s_2 \in S_\delta$, $t \in X$ and $\sigma \in L^*$.
$s_1 \xrightarrow{t,\,\sigma} s_2 =_{def}$ σ is test run of t in s_1, $t \in enabled(f(s_1))$, and $s_1 \xRightarrow{\sigma} s_2$
$s_1 \xrightarrow{t} s_2 =_{def} \exists \sigma : s_1 \xrightarrow{t,\,\sigma} s_2$

Definition 18:

1) A *run* of composite test suite T^e with implementation Sys is a set Ω, containing such chains as:

$$s_0 \xrightarrow{t_1,\,\sigma_1} s_1 \xrightarrow{t_2,\,\sigma_2} \ldots \xrightarrow{t_n,\,\sigma_n} s_n,$$

where $\sigma_i \in L^*$, $t_i \in T$, $s_i \in S$ and s_0 is the initial state of Sys. Elements $s \xrightarrow{t,\,\sigma} s'$ of chains are called run items.

2) An implementation Sys *passes run* Ω of composite test suite T^e, if for each run item $s_1 \xrightarrow{t,\,\sigma} s_2$ in Ω Sys passes test run σ of t in s_1.

3) An implementation Sys *passes* test suite T^e, if it passes all test runs Ω:

$$Sys \textbf{ passes } T^e =_{def} \forall \Omega : Sys \textbf{ passes run } \Omega$$

Note, that for any composite test suite there exists a run including any other runs. Obviously, that a run constructed as a union of chains in all possible runs has such property.

Consider examples of runs Ω of composite test suite in figure 8 with implementation i_3 (fig. 7).

$$\Omega_1 = \left\{ \begin{array}{l} i0 \xrightarrow{t_1,\, a\cdot 1} i1 \xrightarrow{t_3,\, b\cdot 3} i3 \xrightarrow{t_6,\, c\cdot 4} i4 \\ i0 \xrightarrow{t_2,\, a\cdot b\cdot 2} i2 \end{array} \right\}$$

$$\Omega_2 = \left\{ \begin{array}{l} i0 \xrightarrow{t_2,\, a\cdot b\cdot 2} i2 \xrightarrow{t_4,\, a\cdot 3} i3 \xrightarrow{t_6,\, c\cdot 4} i4 \\ i0 \xrightarrow{t_2,\, a\cdot b\cdot 2} i2 \xrightarrow{t_5,\, a\cdot c\cdot 4} i4 \\ i0 \xrightarrow{t_2,\, a\cdot b\cdot 3} i3 \end{array} \right\}$$

Definition 19: Let M be a specification, **imp** be an implementation relation and T^e be a composite test suite, then

$$\begin{array}{rll} T^e \text{ is complete} & \text{if} & \forall Sys : Sys \textbf{ imp } M \Leftrightarrow Sys \textbf{ passes } T^e \\ T^e \text{ is sound} & \text{if} & \forall Sys : Sys \textbf{ imp } M \Rightarrow Sys \textbf{ passes } T^e \\ T^e \text{ is exhaustive} & \text{if} & \forall Sys : Sys \textbf{ imp } M \Leftarrow Sys \textbf{ passes } T^e \end{array}$$

1) Derived test suites: Let T^e be a composite test suite. Consider the notion of concatenation t of two test cases $t_1, t_2 \in T^e$.

Definition 20: Let $t_1 = \langle S^1, L \cup S^M_\delta, T^1, \nu^1, s^1_0 \rangle$ and $t_2 = \langle S^2, L \cup S^M_\delta, T^2, \nu^2, s^2_0 \rangle$. Test case t is the result of concatenation of two test cases t_1, t_2, if $t = \langle S^1 \cup S^2, L \cup S^M_\delta, T^1 \cup T^2 \cup T^t, \nu^t, s^1_0 \rangle$, where $T^t = \{(s_1, s^M, s^2_0)\}$, at that $\nu^1(s_1) = pass$, $t_2 \in enabled(s^M)$, $\nu^t(s) = \nu^1(s)$ for $s \in S^1$ and $\nu^t(s) = \nu^2(s)$ for $s \in S^2$.

Now we construct test suite $T = derive(T^e)$ from T^e. Each test case $t \in T$ in it is successive concatenation of test cases $t_0, \ldots, t_n$, $t_i \in T^e$, where $t_0 \in enabled(s^M_0)$. Test suite T is called *derived* from composite test suite T^e.

Figure 9 shows test suite T_3 derived from composite test suite T^e_3 in figure 8.

Derived test suites have very important property: they pass with the same set of implementations as composite ones from which they are derived.

This enables us to talk about composite test suites as about compact representations of regular test suites. Indeed, derived test suites are always larger by both a number of test cases and a total number of transitions of all test cases in the test suite.

Statement 4: For any composite test suite T^e, a test suite $T = derive(T^e)$ derived from it and an implementation Sys, the following property holds

$$Sys \textbf{ passes } T \Leftrightarrow Sys \textbf{ passes } T^e$$

Proof: For proving this fact it is sufficient to note that for each failed run Ω of T^e with failed chain $s_0 \xrightarrow{t_1,\sigma_1} s_1 \xrightarrow{t_2,\sigma_2} \ldots \xrightarrow{t_n,\sigma_n} s_n \in \Omega$, where k is the number of the first failed test case, there exists a test case t in suite T constructed by successive concatenation of $t_1 \ldots t_k$, where transitions added during concatenation of test cases t_i, t_{i+1} contain labels $f(s_i)$ respectively, and the beginning of transition is a state where test case t_i finishes, i.e. t_i **after** σ_i. And vice versa, for each failed test run $\sigma_1 \cdot s^M_1 \cdot \sigma_2 \cdot \ldots \cdot s^M_{n-1} \cdot \sigma_n$ of a test case $t \in T$ constructed by successive concatenation of $t_1 \ldots t_n$, there exists run Ω where the test case t_n fails in chain $s_0 \xrightarrow{t_1,\sigma_1} s_1 \xrightarrow{t_2,\sigma_2} \ldots \xrightarrow{t_n,\sigma_n} s_n$, and $s^M_i = f(s_i)$. ■

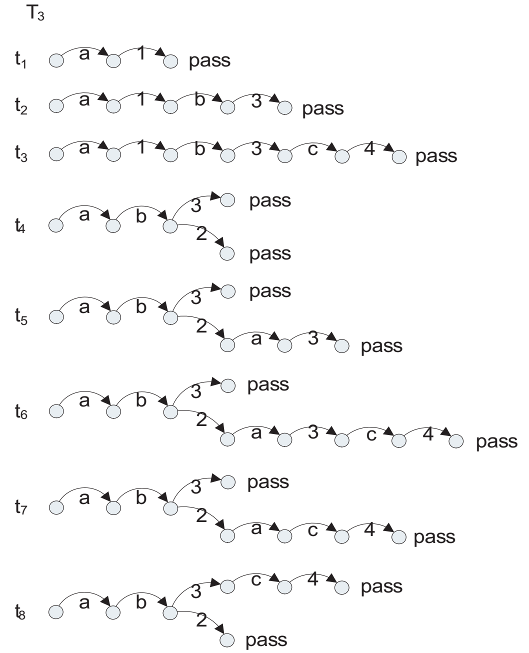

Fig. 9. Example of derived test suite

2) Time of Test Execution: For estimation of test execution time we count the number of interactions between a test case and an implementation. For the sake of simplicity we do not consider the possible difference in execution time between different interactions. Moreover we do not take into account execution times of internal actions. At first, this metrics becomes independent from labels L. At second, it does not depend on the number of internal actions which can vary for the same test run.

Definition 21: An execution time of test run σ of a test case t with implementation Sys is $\rho(\sigma) =_{def} |\sigma|$.

A total execution time of a test suite T can be calculated by the following formula

$$\rho(T) = \sum_{t \in T} \sum_{\sigma \text{ is run of } t} \rho(\sigma)$$

This formula is applicable only for finite test suite with finite number of test runs for each test case. Obviously, that for other test suites execution time is infinite.

Let's compute execution time for derived test suite T_3 on i_3. For that consider all possible test runs of $t_1, \ldots, t_8 \in T_3$.

$$\begin{array}{ll} t_1: & a\cdot 1 \\ t_2: & a\cdot 1\cdot b\cdot 3 \\ t_3: & a\cdot 1\cdot b\cdot 3\cdot c\cdot 4 \\ t_4: & a\cdot b\cdot 2,\ a\cdot b\cdot 3 \\ t_5: & a\cdot b\cdot 2\cdot a\cdot 3,\ a\cdot b\cdot 3 \\ t_6: & a\cdot b\cdot 2\cdot a\cdot 3\cdot c\cdot 4,\ a\cdot b\cdot 3 \\ t_7: & a\cdot b\cdot 2\cdot a\cdot c\cdot 4,\ a\cdot b\cdot 3 \\ t_8: & a\cdot b\cdot 2,\ a\cdot b\cdot 3\cdot c\cdot 4 \end{array}$$

Total sum of all test runs is $\rho(T_3) = 53$.

An execution time of run of a composite test suite is a total sum of all run items.

Definition 22: An execution time of run Ω is

$$\rho(\Omega) = \sum_{s_0 \xrightarrow{t_1,\sigma_1} s_1 \xrightarrow{t_2,\sigma_2} \ldots \xrightarrow{t_n,\sigma_n} s_n \in \Omega} \rho(\sigma_1) + \rho(\sigma_2) + \ldots + \rho(\sigma_n)$$

An execution time of runs Ω_1, Ω_2 on i_3: $\rho(\Omega_1) = 9$, $\rho(\Omega_2) = 16$.

Let's estimate execution time of composite test suites. As it was noticed earlier for composite test suites there exists a run containing run items of any other run. Thus, for time estimation purpose we compare execution time of runs Ω with all test runs of derived test suite T.

The following statement holds.

Statement 5: Let T^e be a composite test suite, $T = derive(T^e)$. For any system Sys and any run of composite test suite Ω such that Sys **passes** Ω, if total execution times of T and Ω are finite, then the following condition holds:

$$\rho(\Omega) \leq \rho(T)$$

Proof: For proving this fact it is sufficient to show that for any chain $s_0 \xrightarrow{t_1,\sigma_1} s_1 \xrightarrow{t_2,\sigma_2} \ldots \xrightarrow{t_n,\sigma_n} s_n \in \Omega$, there exists a test case t in suite T constructed by successive concatenation of test cases $t_1 \ldots t_n$, transitions added during concatenation of test cases t_i, t_{i+1} contain labels $f(s_i)$, and the beginning of transition is a state where test case t_i finishes, i.e. t_i **after** σ_i. An execution time of this test case $\rho(\sigma_1 \cdot f(s_1) \cdot \sigma_2 \cdot \ldots \cdot f(s_{n-1}) \cdot \sigma_n)$ is not less than execution time of the selected chain of Ω. ■

Note, that passing a run Ω by implementation depends on passing test case in run items $s \xrightarrow{t,\sigma} s'$. From that follows, that runs containing the same sets of run items pass or fail simultaneously. We distinguish maximal runs Ω_{max} containing all possible run items for given implementation.

Definition 23: Maximal run Ω_{max} is a run, such that for any Ω all run items $\forall s \xrightarrow{t,\sigma} s'$ are contained in chains of Ω_{max}.

For test suite T_3^e in figure 8 and implementation i_3 one of the possible Ω_{max} is Ω_3

$$\Omega_3 = \left\{ \begin{array}{l} i0 \xrightarrow{t_1,\, a\cdot 1} i1 \xrightarrow{t_3,\, b\cdot 3} i3 \xrightarrow{t_6,\, c\cdot 4} i4 \\ i0 \xrightarrow{t_2,\, a\cdot b\cdot 2} i2 \xrightarrow{t_4,\, a\cdot 3} i3 \\ i0 \xrightarrow{t_2,\, a\cdot b\cdot 2} i2 \xrightarrow{t_5,\, a\cdot c\cdot 4} i4 \\ i0 \xrightarrow{t_2,\, a\cdot b\cdot 3} i3 \end{array} \right\}$$

Note, that $\rho(\Omega_3) = 20$, whereas $\rho(T_3) = 53$.

Statement 6: For all implementations Sys

$$Sys \textbf{ passes } \Omega_{max} \Leftrightarrow \forall\Omega : Sys \textbf{ passes } \Omega$$

As well as for any run, statement 5 holds for Ω_{max}. Hence, execution time of Ω_{max} is less than execution time of derived test suite.

V. CONCLUSION

The work concerns questions of gray box conformance testing. Besides the possibility of interaction with implementation, the test system has a possibility of reading the states of implementation and mapping these states to specification ones.

In section IV, it is shown, how this possibility, given in the form of mappings $MAPPINGS$, can be used to improve the methods of conformance testing.

A novel implementation relation **conff** was defined and a test suite generation algorithm is simplified in comparison with other known generation algorithms for **conf**. Note, that there are other implementation relations and the possibility of simplifying generation for them needs further investigation. Among such relations it is deserved to mention widely spread relation **ioco** and it's improvements.

Composite tests, presented in this work, allow to essentially simplify the description of test suite. Besides, the way of test definition allows to reduce the execution time in comparison with test suites derived from them with the same capability to recognize incorrect implementations.

The idea of composite tests is very close to test sequence generation methods in testing using automata or finite state machine (FSM) models [11]. These methods are improved for the more general case – labelled transition systems. State mapping can be considered as a more general case of status messages, because the mapping does not require implementation states to be equal to specification ones.

In practice composite test suites are successfully used in UniTESK technology [8], [9]. However, existing formalization of UniTESK approach is based on automata models. Conditions on specifications, implementations and tests at which automata equivalency is guaranteed [3], [4] are determined for these models. In UniTESK composite test runs are limited by the only chain, which corresponds to traversal of automata model of specification. Obviously, it is possible only with constraints on a specification and tests, whereas herein the more general case is considered: runs can contain any number of chains.

Let's note that in this work the question of effective complete composite test suite generation is not addressed. Obviously, that complete test suite in a initial state can serve as such suite. However in this case the advantages of composite tests are neglected.

REFERENCES

[1] J. C. M. Baeten and W. P. Weijland. *Process algebra.* Cambridge University Press, New York, NY, USA, 1990.

[2] Gilles Bernot. Testing against formal specifications: A theoretical view. In *TAPSOFT '91: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Volume 2: Advances in Distributed Computing (ADC) and Colloquium on Combining Paradigms for Software Developmemnt (CCPSD)*, pages 99–119, London, UK, 1991. Springer-Verlag.

[3] Igor B. Bourdonov, Alexander Kossatchev, and Victor V. Kuliamin. Irredundant traversal algorithms of directed graphs. deterministic case. *Programming*, 5:59–69, 2003.

[4] Igor B. Bourdonov, Alexander Kossatchev, and Victor V. Kuliamin. Irredundant traversal algorithms of directed graphs. nondeterministic case. *Programming*, 1:2–17, 2004.

[5] Ed Brinksma. A theory for the derivation of tests. pages 63–74, 1988.

[6] Ed Brinksma and Jan Tretmans. Testing transition systems: an annotated bibliography. pages 187–195, 2001.

[7] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[8] A. Khoroshilov. *Specification and testing components with asynchronous interfaces. PhD thesis*. ISP RAS, Moscow, 2006.

[9] V. Kuliamin, A. Petrenko, A. Kossatchev, and I. Bourdonov. Unitesk approach to test development. *Programming*, 29:25–43, 2003.

[10] Rom Langerak. A testing theory for lotos using deadlock detection. In *Proceedings of the IFIP WG6.1 Ninth International Symposium on Protocol Specification, Testing and Verification IX*, pages 87–98, Amsterdam, The Netherlands, The Netherlands, 1990. North-Holland Publishing Co.

[11] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1126, 1996.

[12] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[13] Robin Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.

[14] R De Nicola. Extensional equivalence for transition systems. *Acta Inf.*, 24(2):211–237, 1987.

[15] R. De Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1–2):83–133, November 1984.

[16] David Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pages 167–183, London, UK, 1981. Springer-Verlag.

[17] Jan Tretmans. *A Formal Approach to Conformance Testing. PhD thesis.* University of Twente, 1992.

[18] Jan Tretmans. Conformance testing with labelled transition systems: implementation relations and test generation. *Comput. Netw. ISDN Syst.*, 29(1):49–79, 1996.

[19] Jan Tretmans. Testing concurrent systems: A formal approach. In *CONCUR '99: Proceedings of the 10th International Conference on Concurrency Theory*, pages 46–65, London, UK, 1999. Springer-Verlag.

[20] Rob J. van Glabbeek. The linear time - branching time spectrum ii. In *CONCUR '93: Proceedings of the 4th International Conference on Concurrency Theory*, pages 66–81, London, UK, 1993. Springer-Verlag.