A METHOD FOR VERIFICATION OF TRACE AND TEST EQUIVALENCE

Ivan Christoff

Department of Computer Systems, Uppsala University
Box 520, S-75120, Uppsala, Sweden

ABSTRACT

A method for automatic verification of *trace* and *test equivalence* between *regular CCS* specifications is presented. The method consists of two steps: (i) CCS specifications are transformed to deterministic, and finitely represented attributed trees, which we call *test trees* (TT's), and (ii) TT's are compared for verification of trace or test equivalence between the corresponding CCS specifications.

1 INTRODUCTION

An important problem in verification of concurrent systems is to distinguish specifications based on their observable behaviour. Several relations have been proposed for that purpose. Some of the most well-known are *trace equivalence*, and *observation equivalence* [Mi 80]. In this paper we will present a relation, *test equivalence*, defined by De Nicola and Hennessy [NiHe 84], and give a method for verification of trace and test equivalence between CCS [Mi 80] specifications.

Our approach to verification of trace and test equivalence between CCS specifications, is to transform the specifications to a representation, in which test equivalent specifications have the 'same' representation. We have treated *regular CCS* specifications, which means that we can handle all specifications that can be modelled with nondeterministic finite state machines.

Currently, there is work in progress on automatic verification of relations between finite state systems. For example, a survey of the work done on verification of observation equivalence is given by Bolognesi and Smolka in [BoSm 87]. We have however not found an algorithm for verification of test equivalence in the literature. A 'dual problem' to verification of test equivalence, verification of *failure equivalence* [BHR 84], has been treated by Kanellakis and Smolka [KaSm 83].

The organization of the remaining part of this paper is: section 2 introduces the notation used, and gives the definitions for trace and test equivalence. The verification method is described in section 3. We summarize the results, and present plans for future work, in section 4.

2 NOTATION AND DEFINITIONS

We begin by defining a finite labeled transition system, which is the underlying semantic model for regular CCS specifications. Trace and test equivalence are defined in terms of the "may" and "must" tests of De Nicola and Hennessy [NiHe 84].

Definition 2.1: A *finite labeled transition system* (FLTS) is a quadruple, $\mathcal{S} = (S, E, s_0, T)$, where: S is a finite *set of states* (ranged over by s, s', S, S1, etc.), $E = L \cup \{\tau\}$ is a finite *set of events* (ranged over by e, e', etc., L denotes the *set of observable events*, ranged over by a, a', etc., and τ denotes the *unobservable event*), $s_0 \in S$ is the *initial state*, and $T \subseteq S \times E \times S$ is the *set of labeled transitions* (a labeled transition in T is denoted $s -^{e}\rightarrow s'$, where $s, s' \in S$, and $e \in E$).

Notation: We will use $s -^{e}\rightarrow$ to denote that event e can be performed at state s, and $s \not-^{e}\rightarrow$ to denote its negation. In addition we will use: $s =^{a}\rightarrow$ as shorthand notation for $s -^{\tau}\rightarrow s_1 -^{\tau}\rightarrow \dots s_k -^{a}\rightarrow$ (where k is finite), and $s -\tau^{n}\rightarrow s'$ to denote $s -^{\tau}\rightarrow s_1 -^{\tau}\rightarrow \dots s_{n-1} -^{\tau}\rightarrow s'$. For strings, we will use L^* to denote the *set of strings* over L (ranged over by σ, σ_1, etc.), and ε to denote the *empty string*.

Definition 2.2: For a FLTS $\mathcal{S}$, we define the *τ-closure of a state* as: $\mathcal{T}(s) = \{s' \mid s -\tau^{n}\rightarrow s' \text{ and } n \geq 0\}$, i.e. the set of all states that can be reached from s (including s) by performing τ's. Note that if $s \not-^{\tau}\rightarrow$, $\mathcal{T}(s) = \{s\}$.

Definition 2.3: For a FLTS $\mathcal{S}$, we define the **after** function on states and strings, inductively as:

i) $\{s\}$ **after** $\varepsilon = \mathcal{T}(s)$
ii) $\{s\}$ **after** $a = \text{U}_i \mathcal{T}(s_i)$, where $s_i \in \{s_i \mid s =^{a}\rightarrow s_i\}$
iii) $\{s\}$ **after** $a\sigma = (\{s\}$ **after** $a)$ **after** σ
iv) $\{s\} \cup S_1$ **after** $a = (\{s\}$ **after** $a) \cup (S_1$ **after** $a)$, where $S_1 \neq \emptyset$, and $S_1 \subseteq S$

Intuitively, $\{s_1, \dots, s_n\}$ **after** σ, defines the set of states that $s_1, \dots, s_n$ can transform to after performing the sequence σ.

2.1 Regular CCS specifications

Given a FLTS $\mathcal{S}$, we will interpret S as a set of regular CCS specifications (expressions). We will now define the syntax for regular CCS specifications.

$$\text{S} ::= \textbf{nil} \mid e.\text{S} \mid \text{S} + \text{S} \mid \mu\text{XS},$$

where X is a variable from a set of variables (denoted X), and the following rules apply for the operators:

i) The *nil operator* (**nil**), has no rules.
ii) The *prefix operator* (.), $e.\text{S} -^{e}\rightarrow \text{S}$
iii) The *summation operator* (+), $\text{S1} -^{e}\rightarrow \text{S1'}$ implies $\text{S1} + \text{S2} -^{e}\rightarrow \text{S1'}$
$\text{S2} -^{e}\rightarrow \text{S2'}$ implies $\text{S1} + \text{S2} -^{e}\rightarrow \text{S2'}$
iv) The *recursion operator* (μ), $\text{S}[\mu\text{XS}/\text{X}] -^{e}\rightarrow \text{S'}$ implies $\mu\text{XS} -^{e}\rightarrow \text{S'}$, where S1[S2/X] denotes the expression which results from substituting every free occurrence of X in S1, with S2.

Notation: A recursive equation, e.g. μS(a.S + b.**nil**), will be displayed as: S=a.S + b.**nil**.

2.2 TRACE AND TEST EQUIVALENCE

In this paragraph we will give the definitions for trace and test equivalence from [NiHe 84].

Definition 2.4: For a FLTS $\mathcal{S}$, we define a *test* as, $t = \{a_1, ..., a_n\}$, i.e. a set of observable events, such that $t \subseteq L$.

Definition 2.5: For a FLTS $\mathcal{S}$, we define **may** and **must** predicates on states and tests, inductively as:

i) $\{s\}$ **may** $t \Leftrightarrow \exists s' \in \mathcal{T}(s), \exists a \in t$: $s' =a\rightarrow$
$\{s_1, ..., s_n\}$ **may** $t \Leftrightarrow (\{s_1\}$ **may** $t) \vee ... \vee (\{s_n\}$ **may** $t)$

ii) $\{s\}$ **must** $t \Leftrightarrow \forall s' \in \mathcal{T}(s), \exists a \in t$: $s' =a\rightarrow$
$\{s_1, ..., s_n\}$ **must** $t \Leftrightarrow (\{s_1\}$ **must** $t) \wedge ... \wedge (\{s_n\}$ **must** $t)$

(i) means that there is a state in $\mathcal{T}(s)$, in which at least one of the events in t can be performed. (ii) means that at every state in $\mathcal{T}(s)$, at least one of the events in t can be performed. It should be noted that for any CCS specification S, the set of must tests for S, is a subset of the set of may tests for S.

Notation: We will use $\{s\}$ **may not** t, to denote the negation of $\{s\}$ **may** t, similarly, we will use $\{s\}$ **must not** t, to denote the negation of $\{s\}$ **must** t.

As an example of may and must tests, consider the specification, S = $\tau.a.$**nil** + $b.$**nil**, and the tests $t_1=\{a\}$, and $t_2=\{b\}$. Both t_1 and t_2 are may tests for S, since they satisfy (i). In addition, t_1 is a must test for S, since it satisfies (ii). Note that t_2 is not a must test for S, since: ({S} **after** ε) **must not** $\{b\}$, i.e. t_2 does not satisfy (ii).

Definition 2.6: For a FLTS $\mathcal{S}$, and two arbitrary CCS specifications, S1 and S2, constructed in $\mathcal{S}$, we define *trace* ($=_{tr}$) and *test equivalence* ($=_t$) as:

1) $S1 =_{tr} S2$ iff: $\forall \sigma, \forall t$ (({S1} **after** σ) **may** $t \Leftrightarrow$ ({S2} **after** σ) **may** t)

2) $S1 =_t S2$ iff: $\forall \sigma, \forall t$ (({S1} **after** σ) **must** $t \Leftrightarrow$ ({S2} **after** σ) **must** t)

where $\sigma \in L^*$ and $t \subseteq L$.

(1) means that S1 can perform the same sequences of observable events as S2. (2) means that in addition to (1), after performing the same sequence of observable events S1 and S2 have the 'same deadlock properties'. It should be noted from the above definitions that test equivalence implies trace equivalence.

3 THE VERIFICATION METHOD

In this section we will present the verification method. We will use *synchronization trees* (ST's) [Mi 80], to model regular CCS specifications.

Our approach to verification of test equivalence is to transform CCS specifications to a representation, in which test equivalent specifications have 'identical' representations (which means that the only

possible differences in the representations of two test equivalent specifications are: ordering of events, e.g. *a*.**nil** + *b*.**nil** and *b*.**nil** + *a*.**nil**, and modelling of recursion, e.g. T1=*a*.T1 and T2=*a*.*a*.T2).

The definition of test equivalence (see paragraph 2.2) indicates that the representation we are looking for is a deterministic tree, in which must tests are associated as attributes with each node. With that in mind we defined our representation, *test trees* (TT's).

The method for verification of trace or test equivalence between CCS specifications consists of two steps: (1) CCS specifications are transformed to the corresponding TT's and (2) the TT's are compared for verification of trace or test equivalence between the corresponding specifications. As an example, consider figure 1.

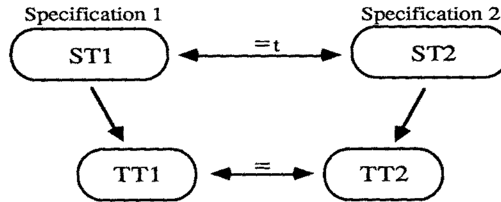

Figure 1: Overview of the verification method.

To establish test equivalence between two specifications represented with ST's, the ST's are transformed to the corresponding TT's (ST1 to TT1, and ST2 to TT2). The problem of verifying test equivalence between the specifications, is consequently transformed to the problem of establishing equality between the corresponding TT's.

3.1 Test trees

We define the previously outlined deterministic attributed tree model:

- A TT is a deterministic ST, in which must tests are associated as attributes with each node. We will refer to a node in a TT as a pair, ⟨T, ***A***⟩, where T is the node name, and $A=\{t_1, ..., t_n\}$ is the set of must tests associated with that node.

The TT node, corresponding to a **nil** node, or a node with a τ...τ.**nil** branch, in a ST, has $A=\{\{\}\}$, while the TT node corresponding to a ST node with a τ-loop, has $A=\{\{\Diamond\}\}$. As an example of a ST (S) and the corresponding TT (T), consider figure 2.

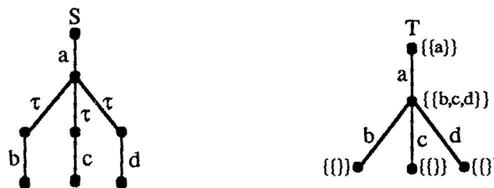

Figure 2: A ST, and the corresponding TT.

The nondeterminism in S is represented in the corresponding TT (T) by the must test associated with the second node, $\{b,c,d\}$, which means that: ({S} **after** *a*) **must** $\{b,c,d\}$.

3.2 THE TRANSFORMATION

The problem of transforming a ST to the corresponding TT is divided in the following three problems: (i) generation of must tests, (ii) elimination of nondeterminism, and (iii) detection of recursion. The solutions for the above three problems are sketched below.

GENERATION OF MUST TESTS

For generation of must tests, we can think of a ST as consisting of two types of nodes: *stable nodes* (which have no τ-labeled branches), and *unstable nodes* (which have τ-labeled branches). As an example, consider the ST S2 shown in figure 3, for which S2 and S3 are unstable nodes, while S4 and S5 are stable nodes.

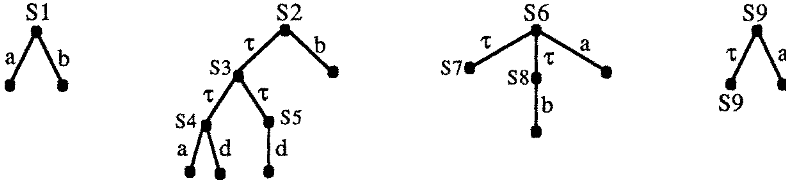

Figure 3: Stable and unstable nodes.

Using the above classification of nodes in a ST, and the definition of a must test (see paragraph 2.2), we observe that *only events at stable nodes need be part of must tests*. For example, for S2 the event *b* need not be part of the must tests, since S2 is an unstable node. Must tests for stable and unstable nodes, are generated in the following manner:

i) **Stable nodes:** Each unique event at a stable node is a must test for that node. As an example consider the ST S1 shown in figure 3. S1 has two must tests: $\{a\}$, and $\{b\}$. Note that a **nil** node is a special case of a stable node. It has the must test $\{\}$.

ii) **Unstable nodes:** For unstable nodes it is necessary to examine if stable nodes are reachable via the τ-branches. There are two cases:
   1) A τ-loop is detected. In that case there are no must tests, since an infinite τ-sequence is possible (e.g. see S9 in figure 3).
   2) Only stable nodes are reached (note that if only one stable node is reached, the generation of must tests is given under (i)). There are two cases:
      - A **nil** node is reached. In that case there are no must tests, e.g. see S6 in figure 3.
      - No **nil** nodes are reached. In that case the events at the stable nodes must be 'combined' to compute the must tests. For example, consider the ST S2 shown in figure 3. S2 has two stable nodes, S4 and S5, with the observable events: *a* or *d*, and *d*. 'Combining' these events results in: $t_1=\{a\}\cup\{d\}=\{a,d\}$ and $t_2=\{d\}\cup\{d\}=\{d\}$, which are the must tests for S2. As we shall see in paragraph 3.3, t_1 is redundant since it contains t_2 as subset.

ELIMINATION OF NONDETERMINISM

We will illustrate how nondeterminism is eliminated in the transformation of a ST to the corresponding TT, by describing the transformation of a node in a ST, to the corresponding node in a TT.

If mapping of recursion is neglected, the transformation of a node in a ST, to the corresponding node in a TT, can be divided in three steps:

i) Eliminate the τ's down to all reachable stable nodes. The unique observable events at each stable node are saved for generation of must tests.

ii) Eliminate nondeterminism between observable events, according to axiom (N1) from [NiHe 84]:

$$\text{(N1)} \qquad e.S1 + e.S2 =_t e.(\tau.S1 + \tau.S2),$$

where e is an observable event, or τ, while S1 and S2 are arbitrary CCS expressions.

iii) Compute the must tests for the node.

As an example of the above outlined transformation, consider the ST S shown in figure 4.

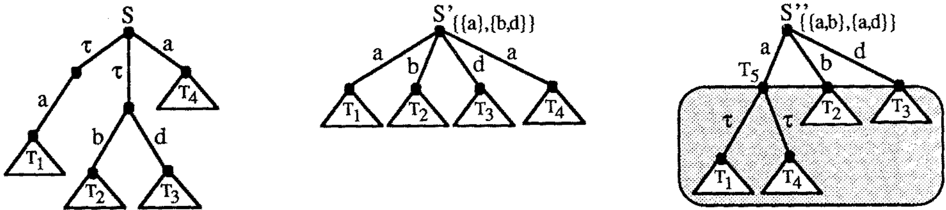

Figure 4: The transformation of one node.

Starting at the root of S, we eliminate the τ's down to all reachable stable nodes, and save the unique events at these nodes, $\{a\}$ and $\{b,d\}$, for computation of must tests. S' shows the effect of step (i) on S. In the next two steps nondeterminism between the a's is eliminated, and the must tests, $\{a,b\}$, and $\{a,d\}$, computed. S'' shows the effect of steps (ii) and (iii) on S'. Note that after the elimination of nondeterminism between the a's, τ's are introduced. The next step in the transformation would be to transform the nodes T_2, T_3, and T_5 (which can be done simultaneously).

DETECTION OF RECURSION

Our representation for a ST (T) is a sum of event prefixed subtrees:

$$T = \tau.T_1 + \ldots + \tau.T_{k-1} + a_k.T_k + \ldots + a_n.T_n$$

where $\{a_k, \ldots, a_n\}$ are observable events, and all subtrees $(T_1, \ldots, T_n)$ are in turn ST's. Recursion w.r.t. T is modelled by for example letting T_k=T. T_k is then called a *recursive node*.

An unique *identification attribute* is assigned to each nonrecursive node in a ST, for detection of recursion during the transformation. We use *Id*(T) to denote the identification attribute for a ST T. For example, for $T_1 = a.T_2 + a.T_3$: $Id(T_1) = \{1\}$, $Id(T_2) = \{2\}$, and $Id(T_3) = \{3\}$.

As indicated by axiom (N1), when nondeterminism between observable events is eliminated 'new' nodes are introduced. For example, if the nondeterminism between the a's at the root of T_1 is eliminated according to (N1), $T_1 = a.T_4$, where $T_4 = (\tau.T_2 + \tau.T_3)$, is a 'new' node, and $Id(T_4) = Id(T_2) \cup Id(T_3) = \{2,3\}$.

To detect recursion during the transformation of a ST to a TT, before a node is transformed the identification attribute for the node is matched against the list of identification attributes for the already transformed nodes.

3.3 COMPARING TWO TEST TREES

If recursion is neglected, the comparison of two TT's, for verification of trace (or test) equivalence between the corresponding specifications, can be divided in two steps:

1) Compare the roots of both TT's w.r.t. trace (or test) equivalence (see criteria (i) and (ii), shown below). If the roots meet the criterion for trace (or test) equivalence, they are considered *bisimilar nodes* (w.r.t. trace or test equivalence).
2) If the roots of the TT's are bisimilar nodes, the comparison is continued for their sons (i.e. starting at the roots of both TT's, for each pair of branches labeled by the same events, the roots of the subtrees are compared).

In order to establish trace (or test) equivalence between two TT's, the TT's are traversed and compared according to (1) and (2).

The criteria for comparing a pair of TT nodes are:

i) **Trace equivalence:** Both nodes must have the same number of branches, labeled by the same events.
ii) **Test equivalence:** In addition to (i), the nodes should have the same must tests.

Note that trace equivalence is checked by both criteria. Since trace equivalence is checked in the criterion for test equivalence, if t_1 and t_2 are must tests for a node, and $t_1 \subset t_2$, then t_2 is redundant and can be eliminated.

3.4 A VERIFICATION EXAMPLE

Consider the specifications represented by the ST's S and S', shown in figure 5.

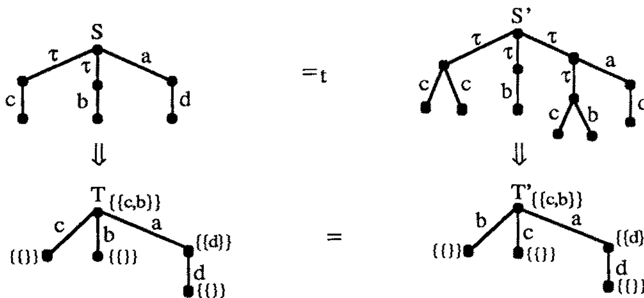

Figure 5: Two ST's, and the corresponding TT's.

If we want to determine whether $S =_t S'$, or $S =_{tr} S'$, we begin by generating the corresponding TT's, T (from S) and T' (from S'). By inspecting the TT's (see figure 5), we can easily determine that they are identical, i.e. S and S' are trace and test equivalent. That conclusion is less obvious if S and S' are inspected.

4 SUMMARY OF RESULTS AND PLANS FOR FUTURE WORK

We have presented a method for automatic verification of trace and test equivalence between regular CCS specifications. Algorithms for: (i) transformation of a ST to the corresponding TT, and (ii) verification of trace and test equivalence between two TT's, are outlined in section 3. The algorithms have been implemented in a tool [AfLi 88].

In [Ch 88a] we have related TT's to the *acceptance trees* (AT's) of Hennessy [He 85]. TT's are isomorphic to AT's, and can be viewed as a representation of AT's, suitable for verification of test equivalence between specifications. We have also found TT's useful for generation of tests [Ch 88b], when trace and test equivalence are used as conformance criteria between an implementation and its specification. Algorithms for test generation from a TT have been implemented [AfLi 88].

Currently, we are working with a probabilistic transition system. We distinguish processes modelled in that system by their interaction with tests [Ch 89]. Rather than stating what tests may or must be accepted by a process, in a probabilistic model it is possible to map the outcome of 'executing' a test on a process, on a number in the interval [0,1]. Based on a notion of tests and test outcomes, we are working on defining equivalences, and the corresponding decision procedures.

ACKNOWLEDGEMENTS

I would like to thank Dr. Bengt Jonsson for reading a draft version of this paper. His suggestions have greatly improved the presentation. This research has been sponsored by the Swedish Board for Technical Development (STU).

REFERENCES

[AfLi 88] H. Afsharazad, L. Lindberg: "Implementing Algorithms and a Graphical Interface for Test Trees", M.Sc. thesis DoCS 88/12, Department of Computer Systems, Uppsala University, 1988, (in Swedish)

[BoSm 87] T. Bolognesi, S.A. Smolka: "Fundamental Results for the Verification of Observational Equivalence: a Survey", *Proc. 7th Intl. Symp. on Protocol Specification, Testing and Verification*, (1987)

[BHR 84] S.D. Brookes, C.A.R. Hoare, A.W. Roscoe: "A Theory of Communicating Sequential Processes", *J. ACM* **31**:3, pp 560-599, (1984)

[Ch 88a] I. Christoff: "Test Trees" (preliminary report), Research report DoCS 88/14, Department of Computer Systems, Uppsala University, 1988

[Ch 88b] I. Christoff: "Testing for Conformance", *Proc. 2nd Intl. Symp. on Interoperable Information Systems*, pp 273-280, (1988)

[Ch 89] I. Christoff: "Distinguishing Probabilistic Processes Through Testing", Research report DoCS 89/16, Department of Computer Systems, Uppsala University, 1989

[He 85] M. Hennessy: "Acceptance Trees", *J. ACM* **32**:4, pp 896-928, (1985)

[KaSm 83] P.C. Kanellakis, S.A. Smolka: "CCS Expressions, Finite State Processes, and Three Problems of Equivalence", *Proc. 2nd ACM Symp. on Principles of Distributed Computing*, pp 228-240, (1983)

[Mi 80] R. Milner: *A Calculus of Communicating Systems*, Springer-Verlag, LNCS **92**, (1980)

[NiHe 84] R. De Nicola, M. Hennessy: "Testing Equivalences for Processes", *Theoretical Computer Science* **34**, pp 83-133, (1984)