CRIM

CRIM — Documentation/Communications

Proceedings of the 22nd IFIP International Conference on Testing Software and Systems: Short Papers

Editors
Alexandre Petrenko
Adenilso Simão
José Carlos Maldonado

October, 2010

ISBN-13: 978-2-89522-136-4

Financial Partner:

Développement économique, Innovation et Exportation Québec

Formal conformance verification

Igor Burdonov[1], Alexander Kosachev[1],

[1] Institute for System Programming of the Russian Academy of Sciences,
A. Solzhenitsyna st. 25, 109004 Moscow, Russia
{igor, kos}@ispras.ru

Abstract. In this paper, we propose a method for verification the so-called ***saco***-relation between an implementation and the specification LTSs. We show that this verification can be finite and complete if the number of states and actions of LTSs is finite.

Keywords: LTS model, conformance relation, formal verification.

1 Introduction

In formal model based conformance verification, one usually assumes that the specification and a system under test (SUT) are represented by the same formal models and there is a binary relation that has to be checked via the verification process. In this paper, we limit ourselves with so-called functional restrictions which describe how a system should interact with its environment; these restrictions are described by the specification. A SUT is conforming if its interaction satisfies the same external restrictions.

In various application domains the functional behavior (or the specification) of a system can be described by the LTS model. In this paper, we assume that we should compare two LTS models, one of which describes the reference behavior while another one describes the behavior of a SUT. We introduce a conformance relation between LTSs and show how this conformance relation can be verified.

2 Conformance relation and how to check it

2.1 Interaction and its safety

The conformance relation is based on the interaction model. We can only observe the behavior that is induced by the environment and can be observed by the environment. Such interaction can be modeled by a testing machine [1-6]. The testing machine is a black box where an implementation (SUT) is embedded. The environment is modeled by an operator who presses buttons allowing an implementation to execute some

actions. Observations can be of two types: an implementation executes an allowed action or an implementation refuses to execute any allowed action.

We underline that not a single action but a set of actions is allowed when pressing a button. For example, when talking about reactive systems the set of actions is partitioned into two sets of inputs and outputs. Each input action is applied to an implementation by pressing a button that corresponds to this input. However, when expecting the reply of an implementation a button is pressed that allows to accept any output action.

Each button has its own set of permissible actions. An action can be observed if the action is allowed by a pressed button and an implementation can execute the action. If an implementation cannot execute any action allowed by a pressed button then a refusal is observed. After observing execution of an action or after observing the refusal all external actions are forbidden before another button is pressed.

Thus, interaction depends on sets of actions associated with buttons as well as on the set of buttons for which a refusal can be observed. For the sake of simplicity, in this paper, we assume that all refusals are observable. Correspondingly, the interaction is completely described by a set L of external actions and a collection R of sets of actions associated with buttons. We also assume that $\cup R = L$ and refer to such an interaction as to R-interaction.

An implementation can often execute not only external actions but also the internal (non-observable) action τ. Such actions are always allowed. As usual, we assume that a finite sequence of internal actions needs finite time to be executed while an infinite sequence of such actions can be executed infinitely long. An infinite sequence of τ-actions is called divergence and is denoted by Δ. Divergence can induce a problem, since when pressing a button an operator does not know how long he has to wait for a response or internal actions will be executed forever. Therefore, an operator is deadlocked as he cannot continue the interaction and cannot stop it.

We also consider a special action that cannot also be controlled by buttons; this actions called destruction is denoted by γ. The action γ models undesirable system behavior including the system destruction that is not allowed through ordinary interaction. For example, in defense devices such an action can correspond to self-destruction. Moreover, the destruction can describe the non-specified behavior in partial specifications. When people do not consider a destruction they motivate this that an implementation should only check if parameters are correct when calling an implementation [7,8]. If parameters of a message are incorrect then the implementation should ignore a message or should mention about an error. Such requirement is understandable when an implementation is for “a common use” and should be foolproof. However, when interacting with internal components or subsystems to which an access is strongly limited such checking seems to be superfluous. If the parameter structure is complex and the correctness conditions are not trivial such checking adds unnecessary complexity when designing a system and the system performance is also adversely affected. There is an alternative to use the strong specification of calling operators [9]. For instance, a call for deallocation memory that was not got earlier via memory request is incorrect as it violates preconditions. Correspondingly we should check not how components response to incorrect requests of other components but whether those requests are correct. In other words, since we would like to assure the implementation correctness (not its

environment) we are not interested in the implementation behavior when a request is incorrect. Specification destruction describes situations when an implementation is allowed to have any behavior including the real destruction.

Destruction semantics assumes that the destruction cannot happen if the environment behavior is correctly implemented. The interaction is safe if the destruction cannot occur and buttons are pressed only there isn't divergence in an implementation.

2.2 LTS and LTS traces

LTS is a tuple $\boldsymbol{S} = LTS(V_S,L,E_S,s_0)$ where V_S is the non-empty set of states with the initial state s_0, L is the non-empty set of external actions, $E_S \subseteq V_S \times (L \cup \{\tau,\gamma\}) \times V_S$ is the set of transitions. As usual, a transition from state s to state s' via action z is denoted $s \text{—} z \rightarrow s'$ while $s \text{—} z \rightarrow$ denotes that there is a transition from state s via action z, $s \text{—} z \rightarrow =_{\mathbf{def}} \exists s'(s \text{—} z \rightarrow s')$. A sequence of sequential transitions is a *path* of LTS.

A step of LTS functioning in a testing machine is an execution of a single transition that is allowed by a current button and is specified at a current state (τ- and γ-transitions are always allowed). If there are several such transitions then only one of them can be executed.

State s is *divergent* ($s\uparrow$) if s is the initial state of an infinite *path* of τ-transitions (in particular, of a τ-cycle); otherwise, state s is *convergent* ($s\downarrow$). State is *stable* if there are no τ- or γ-transitions at this state. Refusal $P \in R$ occurs at a stable state if at this state there are no transitions under actions of the set P.

At each stable state we add a virtual loop $s \text{—} P \rightarrow s$ for each possible refusal P, while at each divergent state Δ-transitions $s \text{—} \Delta \rightarrow$ are added. In the obtained LTS we consider only *paths* which are not prolonged after Δ- and γ-transitions. A *trace* is a sequence of action labels of path transitions without symbol τ. If a path with trace σ has the initial state s and the final state s' then we denote this by $s \Rightarrow \sigma \Rightarrow s'$ while $s \Rightarrow \sigma \Rightarrow$ denotes that there is a trace σ with the initial state s, $s \Rightarrow \sigma \Rightarrow =_{\mathbf{def}} \exists s' \, (s \Rightarrow \sigma \Rightarrow s')$. The set $T(s) = \{\sigma \mid s \Rightarrow \sigma \Rightarrow\}$ is the set of traces at state s.

2.3 Safety hypothesis and safe conformance

We now define the relation "a button is safe after a trace": button P is *safe after trace* σ if there is no divergence after the trace and the destruction cannot occur after an action allowed by the button P. Only safe buttons are pressed through the safe interaction. Formally, the button P is safe:

At state s: P ***safe*** $s =_{\mathbf{def}} s\downarrow \;\&\; \neg s \Rightarrow \langle\gamma\rangle \Rightarrow \;\&\; \forall z \in P \; \neg s \Rightarrow \langle z,\gamma\rangle \Rightarrow$;

For the set S of state: P ***safe*** $S =_{\mathbf{def}} \forall s \in S \; P$ ***safe*** s;

After trace σ with the starting state s: P ***safe*** s ***after*** $\sigma =_{\mathbf{def}} P$ ***safe*** (s ***after*** σ),

where s ***after*** $\sigma =_{\mathbf{def}} \{s` \mid s \Rightarrow \sigma \Rightarrow s`\}$.

The button safety implies the safety of actions and refusals. Refusal P is *safe* if the button P is safe. Action z is *safe* if this action is allowed by some safe button P, i.e., $z \in P$. A trace with the initial state s is safe if 1) $\neg(s \Rightarrow \langle\gamma\rangle \Rightarrow)$, 2) and each trace action

is safe after the corresponding trace prefix. The set of all safe traces at state s is denoted by $\boldsymbol{Safe}(s)$. By default, the initial state of all traces and paths is the initial state of the LTS.

The safety requirement results in the set of safe implementations. This set is determined by the *safety hypothesis*: implementation $\boldsymbol{I}$ is *safe* for the specification $\boldsymbol{S}$ if 1) the implementation does not contain a trace $\langle\gamma\rangle$ if there is no such trace at the initial state of the specification 2) after each common safe trace of an implementation and the specification any button that is safe in the specification is safe in the implementation after this trace:

$\boldsymbol{I}$ ***safe for*** $\boldsymbol{S} =_{\mathbf{def}} (\gamma \notin T(s_0) \Rightarrow \gamma \notin \boldsymbol{T}(i_0))$ & $\forall \sigma \in \boldsymbol{Safe}(s_0) \cap T(i_0)$ $\forall P \in R$

(P ***safe*** s_0 ***after*** $\sigma \Rightarrow P$ ***safe*** i_0 ***after*** σ).

An implementation $\boldsymbol{I}$ is *conforming* to the specification $\boldsymbol{S}$ if $\boldsymbol{I}$ is safe and the following *conformance condition* holds: any observation that is possible in the implementation after pressing a safe button is allowed by the specification.

$\boldsymbol{I}$ ***saco*** $\boldsymbol{S} =_{\mathbf{def}} \boldsymbol{I}$ ***safe for*** $\boldsymbol{S}$ &

$\forall \sigma \in \boldsymbol{Safe}(s_0) \cap T(i_0)$ $\forall P$ ***safe*** s_0 ***after*** σ ($\boldsymbol{obs}(i_0$ ***after*** $\sigma, P) \subseteq \boldsymbol{obs}(s_0$ ***after*** $\sigma, P)$),

where $\boldsymbol{obs}(M,P) =_{\mathbf{def}} \{u \in P \cup \{P\} \mid \exists m \in M$ & $m \Rightarrow \langle u \rangle \Rightarrow\}$ is the set of observations which are possible at states of the set M when pressing the button P.

When the safety hypothesis cannot be checked the hypothesis becomes a precondition of testing; the testing objective then is to check the conformance relation. Both conditions can be checked when formal verification is used.

3 Verification technique

We first establish restrictions for the interaction model, implementation and specification which allow finite conformance verification and which are considered in this paper. We assume that the set L of actions is finite, thus, the number of the sets R is finite, correspondingly, a set corresponded to each button and the set $L \cup R$ of observations also are finite; moreover, the sets of states of an implementation and the specification are finite, i.e., their sets of transitions are finite.

The idea behind our approach is as follows. Consider all the states of an implementation which are reachable by traces that are safe in the specification: $\cup\{i_0$ ***after*** $\sigma \mid \sigma \in \boldsymbol{Safe}(s_0)\}$. For each such state consider a collection $S(i)$ of subsets of states of the specification: $S(i) = \{s_0$ ***after*** $\sigma \mid \sigma \in \boldsymbol{Safe}(s_0)$ & $i \in (i_0$ ***after*** $\sigma)\}$. This collection has subsets of states of the specification after all traces which are safe in the specification, are traces of the implementation and have the final state i. When verifying we will construct such collections $S(i)$ step by step adding to them sets s_0 ***after*** σ. At the beginning we check the condition $\langle\gamma\rangle \notin T(s_0) \Rightarrow \langle\gamma\rangle \notin T(i_0)$ as a part of the safety hypothesis. If this condition holds then at each time instance at each state i another part of the safety hypothesis and of the conformance relation is checked: $\forall P \in R$ $\forall S \in S(i)$ (P ***safe*** $S \Rightarrow P$ ***safe*** i & $\boldsymbol{obs}(\{i\},P) \subseteq \boldsymbol{obs}(S,P)$). If these conditions do not hold then a fault is claimed. If all the collections $S(i)$ are completely constructed then the verification is completed with the verdict "OK".

We first construct intermediate data structures for the specification and implementation. Specification data structures do not depend on an implementation

and are used without modification when verifying any implementation using the same *R*-interaction. Correspondingly, implementation data structures do not depend on the specification and can be used for checking its conformance w.r.t. any specification when using the same *R*-interaction. Correspondingly, we would not take into account such data structures when estimating the verification algorithm complexity.

For the specification, we consider a collection of subsets of final states of safe traces: $\boldsymbol{safeder}(s_0) = \{s_0 \textbf{ after } \sigma | \sigma \in \boldsymbol{Safe}(s_0)\}$. For each subset $S \in \boldsymbol{safeder}(s_0)$, the set $A(S) = (\cup\{P \cup \{P\} | P \textbf{ safe } S\}) \cup \{\tau\}$ contains all safe observations (actions and refusals) and symbol τ while the set $B(S,u) = \cup\{s \textbf{ after } \langle u \rangle | s \in S\}$ contains each state s' such that there is a transition (s,u,s'), $s \in S$ in the specification; we also assign $B(S,\tau) = S$. Apparently, $B(S,u) = \varnothing$ if there are no such transitions.

In fact, when proceeding in the way discussed above we derive the observable form of the specification LTS; states of the observable form are sets of states $S = s_0 \textbf{ after } \sigma$, where $\sigma \in \boldsymbol{Safe}(s_0)$, a transition $S \text{—} u \rightarrow S'$ means that $S' = B(S,u)$ and $B(S,u) \neq \varnothing$.

For an implementation, we consider a set of states which are reachable via safe traces (in the implementation): $\cup\boldsymbol{safeder}(i_0)$. For each state $i \in \cup\boldsymbol{safeder}(i_0)$ we define the following sets:

$C(i) = \langle i \text{—} u \rightarrow i` | u = \tau \vee \exists P \textbf{ safe } i\ u \in P \rangle$ is a set of all safe transitions at state i;

$D(i) = \langle i \text{—} u \rightarrow i` | \forall P \textbf{ safe } i\ u \notin P \rangle$ is a set of transitions at state i which are not safe.

The safe hypothesis for an implementation is equivalent to the condition

$\forall i \text{—} u \rightarrow i` \in D(i)\ \forall S \in S(i)\ u \notin A(S)$

while the conformance relation is equivalent to the condition

$\forall i \text{—} u \rightarrow i` \in C(i)\ \forall S \in S(i)\ u \in A(S) \Rightarrow B(S,u) \neq \varnothing$.

A proposed algorithm is based on deriving sets $S(i)$ step by step while checking safety and conformance at each step. A special intermediate list W of pairs $(S, i \text{—} u \rightarrow i`)$ is constructed where $S \in S(i)$ and $i \text{—} u \rightarrow i` \in C(i) \cup D(i)$. At the beginning it holds that $S(i_0) = \{S_0\}$ where $S_0 = \{s_0 \textbf{ after } \langle\rangle\}$ is a set of states of the specification after the empty trace and W has each pair $(S_0, i_0 \text{—} u \rightarrow i`)$ where $i_0 \text{—} u \rightarrow i` \in C(i_0) \cup D(i_0)$.

An algorithm step is as follows. If the list W is empty then the algorithm stops with the verdict "conforming". Otherwise, the head item $(S, i \text{—} u \rightarrow i`)$ of the list $W$ is selected and is deleted from the list. For the selected item, first, the safety hypothesis is checked. If $i \text{—} u \rightarrow i` \in D(i)$ & $u \in A(S)$ then an implementation is claimed as a faulty implementation and the algorithm stops. If there is no fault then the conformance is checked. If $i \text{—} u \rightarrow i` \in D(i)$ or $i \text{—} u \rightarrow i` \in C(i)$ & $u \notin A(\mathrm{S})$ then the next step of the algorithm is performed. If $i \text{—} u \rightarrow i` \in C(i)$ & $u \in A(\mathrm{S})$ & $B(\mathrm{S},u) = \varnothing$ then an implementation is claimed as a faulty implementation and the algorithm stops. If there is no fault and $B(\mathrm{S},u) \in S(i`)$ then the next step of the algorithm is performed. Otherwise, the set $B(\mathrm{S},u)$ is added to $S(i`)$ while each pair $(B(\mathrm{S},u), i` \text{—} u` \rightarrow i``)$ where $i` \text{—} u` \rightarrow i`` \in C(i`) \cup D(i`)$ is added to the list W, and the next step of the algorithm is performed.

It is easy to show that an implementation is claimed as a conforming implementation if and only if the implementation conforms to the specification.

We also notice that the algorithm is easy for parallel programming, since algorithm steps corresponded to different items of the list W can be performed in parallel; if

there is no fault then the algorithm stops when there are no steps to be performed and the list W is empty.

We now estimate the algorithm complexity in the worst case that is in the case when parallel programming is useless. Since each pair $(S, i \text{—} u \rightarrow i')$ can appear in the list W at most once, the number of algorithm steps is equal to $O(m \cdot 2^n)$ where $m=|E_I|$ is a number of implementation transitions, and $n = |V_S|$ is the number of states of the specification. At each step we can assume that time needed for all actions such as checking whether an item belongs to a given set, extracting items of the set B and adding an item to $S(i')$, is a proper constant, all observations, all implementation states and all subsets of the set of specification states are hashed by integers, A and S are two-dimensional Boolean arrays, C and D are three-dimensional Boolean arrays and B is a two-dimensional array of numbers of subsets of the set of specification states. The exception should be made for adding pairs $(B(S,u), i' \text{—} u \rightarrow i'')$ to the list W where $i' \text{—} u \rightarrow i'' \in C(i') \cup D(i')$. Summarizing the above we obtain $O(m \cdot 2^n)$ pairs. If sets of transitions $C(i')$ and $D(i')$ are additionally represented as lists for every state i', this operator needs $O(m \cdot 2^n)$ time units to be executed. Thus, the resulting estimation equals $O(m \cdot 2^n)$. Coefficient 2^n is the maximal number of states of the observable form $(2^n - 1)$ of the LTS specification. If the observable form has the same number of states as the initial specification that happens when the specification is deterministic, i.e., has no τ transitions and at each state there is at most one transitions for each action, then this coefficient equals n. Intuitively seems that the algorithm complexity on average is much less especially when using parallel programming and we are intended to study this upper bound more rigorously.

References

1. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V.: Formalization of Test Experiments. Programming and Computer Software, 2007, Vol. 33, No. 5, pp.239-260
2. Bourdonov, I.B., Kossatchev, A.S., and Kuliamin, V.V.: Conformance theory for the systems with input refusals and destruction . Moscow, Nauka, 2008 (in Russian) http://panda.ispras.ru/~RedVerst/RedVerst/Publications/TR-03-2006.pdf
3. Bourdonov, I.B.: Conformance theory for functional formal model based testing of software systems . D-r Thesis, Moscow, ISP RAS, 2008 (in Russian) http://panda.ispras.ru/~RedVerst/RedVerst/Publications/TR-01-2007.pdf
4. van Glabbeek, R.J.: The linear time - branching time spectrum. CONCUR'90 (J.C.M. Baeten and J.W. Klop, ed.), LNCS 458, Springer-Verlag, 1990, pp 278–297.
5. van Glabbeek, R.J.: The linear time - branching time spectrum II; the semantics of sequential processes with silent moves. CONCUR'93 (E. Best, ed.), LNCS 715, Springer-Verlag, 1993, pp. 66–81
6. Milner, R.: Modal characterization of observable machine behaviour. CAAP 81 (G. Astesiano and C. Bohm, ed.), LNCS 112, Springer-Verlag, 1981, pp. 25–34
7. Tretmans, J.: Test Generation with Inputs, Outputs and Repetitive Quiescence. Software-Concepts and Tools, Vol. 17, Issue 3, 1996.
8. Heerink, L.: Ins and Outs in Refusal Testing. PhD thesis, University of Twente, Enschede, The Netherlands, 1998.
9. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM, 12(10), October 1969.