

Action Refinement in Testing with *uioco*

Machiel van der Bijl and Arend Rensink
Software Engineering, Department of Computer Science,
University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands
email: {vdbijl, rensink}@cs.utwente.nl

Jan Tretmans
Software Technology Research Group, Radboud University
P.O. Box 9010, 6500 GL Nijmegen, The Netherlands
email: tretmans@cs.kun.nl

Abstract

In *model based testing* test cases are derived from a specification of the implementation that we want to test. In general the specification is more abstract than the implementation. This may result in test cases that are not executable, because their actions are too abstract; the implementation does not understand them. One approach is to rewrite the specification to the required level of detail and regenerate the test cases. Rewriting a specification by hand is an error-prone and time consuming exercise that is not always favorable. Very often there is a good reason for the level of abstraction in a specification, for example to illustrate the structure of the system or to separate concerns.

In this paper we present an approach for action refinement. We will apply this approach to a simple case of action refinement, so called atomic linear input-inputs refinement. For this type of action refinement our approach enables us to automatically refine traces, transition systems and test cases. Furthermore, we present an implementation relation that relates an abstract specification with its concrete implementation and show that it is equivalent with the *uioco* implementation relation on the refined specification.

I. INTRODUCTION

Executability of test cases is an important issue in model based testing. Because test cases are derived from the specification, the test cases have the same level of abstraction as the specification. In general, the specification is more abstract than the implementation. As a result the derived test cases may not be executable. There are several ways to deal with this situation. The most straightforward way is to rewrite the specification to a level of detail that ensures executability of the test cases and next to regenerate the test cases. In practice, this has some serious drawbacks. Rewriting a specification can be time consuming and error-prone. Furthermore, very often there is a good reason for the level of detail in the specification. For example to show the structure (or overview) of the system and to separate concerns. As a tester we are not so much interested in rewriting specifications, but in executable test cases. Therefore we are in favor of an automated way to close the abstraction gap between an abstract specification and a concrete implementation, in order to obtain test cases that are concrete enough to execute. The general idea is that we define the missing information between specification and implementation in a refinement relation. This refinement relation tells us that wherever we read a certain abstract action in the specification this corresponds to a set of actions in the implementation. For example if the specification tells us to input two euros and the implementation also allows the insertion of two one euro pieces, the refinement relation tells that wherever we read two euros we can also read one euro followed by one euro.

Action refinement has been studied extensively; see Gorrieri and Rensink for an overview [2]. Action refinement in model based testing has not been studied at all. This is surprising, because everybody that uses model based testing runs into this problem. In this paper we present a first step in our effort towards action refinement in model based testing. It is a simple, though non-trivial case of action refinement: *atomic linear input-inputs refinement*. It is atomic because no abstract actions are allowed to occur in between the

This research was supported by the dutch research program PROGRESS under project: TES5417: Atomyste – ATOM splitting in eMbedded sYStems TESting.

concrete actions, linear because we only consider a sequence of actions (so no branching behavior) and input-inputs because we refine a single input action into a sequence of input actions. With this type of action refinement we fill in the parts of our action refinement approach. We present this approach and show how we fill in the parts. The main parts of the approach are: refinement of traces, transition systems and test cases. Furthermore we introduce a new implementation relation \mathbf{uioco}_τ between an abstract specification and a concrete implementation and we show that it is equivalent with \mathbf{uioco} between the refined specification and the same implementation.

The main contribution of this paper is the approach that we use for atomic linear input-inputs refinement. We will argue that this approach can be extended to more general types of action refinements. This extension is the next step in our research.

We start with recapturing some results and notations that we will use throughout the paper in section II. We present trace refinement in section IV and the refinement of labeled transition systems in section V. In section VI we present the implementation relation \mathbf{uioco}_τ . We conclude with future work and conclusions in section VII.

II. FORMAL PRELIMINARIES

This section recalls some aspects of the theory behind \mathbf{uioco} (a further evolution of \mathbf{ioco}) that are used in this paper; see [4] and [3] for a more detailed exposition.

Labeled Transition Systems. A labeled transition system (LTS) description is defined in terms of states and labeled transitions between states, where the labels indicate what happens during the transition. Labels are taken from a global set \mathbf{L} . We use a special label $\tau \notin \mathbf{L}$ to denote an internal action. For arbitrary $L \subseteq \mathbf{L}$, we use L_τ as a shorthand for $L \cup \{\tau\}$. We use the label a to range over L . We deviate from the standard definition of labeled transition systems in that we assume the label set of an LTS to be partitioned in an input and an output set.

Definition II.1: A labeled transition system is a 5-tuple $\langle Q, I, U, T, q_0 \rangle$ where Q is a non-empty countable set of states; $I \subseteq \mathbf{L}$ is the countable set of input labels; $U \subseteq \mathbf{L}$ is the countable set of output labels, which is disjoint from I ; $T \subseteq Q \times (I \cup U \cup \{\tau\}) \times Q$ is a set of triples, the transition relation; $q_0 \in Q$ is the initial state.

We use L as shorthand for the entire label set ($L = I \cup U$); furthermore, we use Q_p, I_p etc. to denote the components of an LTS p . We commonly write $q \xrightarrow{\lambda} q'$ for $(q, \lambda, q') \in T$. Since the distinction between inputs and outputs is important, we sometimes use a question mark before a label to denote input and an exclamation mark to denote output. We denote the class of all labeled transition systems over I and U by $\mathcal{LTS}(I, U)$. We represent a labeled transition system in the standard way, by a directed, edge-labeled graph where nodes represent states and edges represent transitions.

A state that cannot do an internal action is called *stable*. A state that cannot do an output or internal action is called *quiescent*. We use the symbol δ ($\notin \mathbf{L}_\tau$) to represent quiescence: that is, $p \xrightarrow{\delta} p$ stands for the absence of any transition $p \xrightarrow{a} p'$ with $a \in U_\tau$. For an arbitrary $L \subseteq \mathbf{L}$, we use L_δ as a shorthand for $L \cup \{\delta\}$. We use the label λ to range over L_δ .

An LTS is called *strongly responsive* if it always eventually enters a quiescent state; in other words, if it does not have any infinite U_τ -labeled paths. For technical reasons we restrict $\mathcal{LTS}(I, U)$ to strongly responsive transition systems. From a practical perspective it would be nice if the constraint can be weakened. This is probably possible, but needs further research.

A *trace* is a finite sequence of observable actions. The set of all traces over L ($\subseteq \mathbf{L}$) is denoted by L^* , ranged over by σ , 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 . We use the standard notation with single and double arrows for traces: $q \xrightarrow{a_1 \dots a_n} q'$ denotes $q \xrightarrow{a_1} \dots \xrightarrow{a_n} q'$, $q \xrightarrow{\epsilon} q'$ denotes $q \xrightarrow{\tau \dots \tau} q'$ and $q \xrightarrow{a_1 \dots a_n} q'$ denotes $q \xrightarrow{\epsilon} \xrightarrow{a_1} \xrightarrow{\epsilon} \dots \xrightarrow{a_n} \xrightarrow{\epsilon} q'$ (where $a_i \in \mathbf{L}_\delta$). We will use Σ to denote a set of traces. If $\sigma = \lambda_1 \dots \lambda_n$ then $\sigma|i = \lambda_i$ where $1 \leq i \leq |\sigma| = n$, and $L(\sigma) = \{\lambda_1 \dots \lambda_n\}$. We

use the symbol \sqsubseteq to denote trace prefix and the symbol \downarrow to denote prefix closure.

$$\sigma_1 \sqsubseteq \sigma \Leftrightarrow \exists \sigma_2 : \sigma_1 \cdot \sigma_2 = \sigma \quad (1)$$

$$\downarrow \sigma = \{\sigma' \mid \sigma' \sqsubseteq \sigma\} \quad (2)$$

$$\downarrow \Sigma = \bigcup \{\downarrow \sigma \mid \sigma \in \Sigma\} \quad (3)$$

We will not always distinguish between a labeled transition system and its initial state. We will identify the process $p = \langle Q, I, U, T, q_0 \rangle$ with its initial state q_0 , and we write, for example, $p \xrightarrow{\sigma} q_1$ instead of $q_0 \xrightarrow{\sigma} q_1$.

Input-output transition systems. An *input-output transition system* (IOTS) is a labeled transition system that is completely specified for input actions. The class of input-output transition systems with input actions in I and output actions in U is denoted by $\mathcal{IOTS}(I, U)$ ($\subseteq \mathcal{LTS}(I, U)$). Notice that we do not require IOTS's to be strongly responsive.

Definition II.2: An *input-output transition system* $p = \langle Q, I, U, T, q_0 \rangle$ is a labeled transition system for which all inputs are enabled in all states: $\forall q \in Q, a \in I : q \xrightarrow{a}$ (*weak input enabledness*).

Conformance. The testing scenario on which **uioco** is based wants to establish a notion of conformance between a specification and an implementation [3]. The specification is an LTS, specifying the required behavior. Since the testing approach is black box testing, we do not know anything about the implementation, however we assume that it is possible to model it as an IOTS. This assumption is referred to as the test hypothesis. We want to stress that we do not need to *have* this model when testing the implementation. We only *assume* that the implementation *behaves* as an IOTS [1].

Given a specification s and an (assumed) model of the implementation i , the relation $i \mathbf{uioco} s$ expresses that i conforms to s . Whether this holds is decided on the basis of the *utrac*es of s : it must be the case that, after any such trace σ , every output action (including quiescence) that i is capable of should be allowed according to s . This is formalized by defining p **after** σ (the set of states that can be reached in p after the trace σ), $out(p)$ (the set of output and δ -actions of p), $Straces(p)$ (the suspension traces of p) and $Utraces(p)$ (the universal straces of p). We use the convention $\mathcal{F} \subseteq \mathcal{L}_\delta^*$.

Let $p \in \mathcal{LTS}(I, U)$, let $P \subseteq Q_p$ be a set of states in p and let $\sigma \in \mathcal{L}_\delta^*$.

$$p \text{ after } \sigma =_{\text{def}} \{p' \mid p \xrightarrow{\sigma} p'\} \quad (4)$$

$$out(p) =_{\text{def}} \{x \in U \mid p \xrightarrow{x}\} \cup \{\delta \mid p \xrightarrow{\delta}\} \quad (5)$$

$$out(P) =_{\text{def}} \bigcup \{out(p) \mid p \in P\} \quad (6)$$

$$Straces(p) =_{\text{def}} \{\sigma \in \mathcal{L}_\delta^* \mid p \xrightarrow{\sigma}\} \quad (7)$$

$$Utraces(p) =_{\text{def}} \{\sigma \in Straces(p) \mid \forall q, (\sigma_1 \cdot a) \sqsubseteq \sigma : \\ (a \in I \wedge p \xrightarrow{\sigma_1} q) \implies q \xrightarrow{a}\} \quad (8)$$

Definition II.3: Let $i \in \mathcal{IOTS}(I, U)$, $s \in \mathcal{LTS}(I, U)$.

$i \mathbf{ioco}_{\mathcal{F}} s =_{\text{def}} \forall \sigma \in \mathcal{F} : out(\text{after } \sigma) \subseteq out(s \text{ after } \sigma)$

For $\mathcal{F} = Utraces(s)$ we abbreviate $\mathbf{ioco}_{\mathcal{F}}$ to **uioco**.

III. ATOMIC INPUT-INPUTS ACTION REFINEMENT

In general, specifications are more abstract than implementations. This creates an interesting dilemma in model based testing, because we check for conformance between a specification and an implementation. To explain action refinement in testing we start with an example of this problem (we will use this as our running example).

Example III.1: Figure 1 shows a specification and a test case of a very simple data entry application. On the left hand side we see the abstract specification and on the left hand side we see the test case that is derived from the specification. The abstract specification tells us that we can enter address data, push the store button and then the system either stores the address data or gives an error. We can read the test case as follows: we want to enter the address data, press the store button and then observe the response of the IUT. The IUT

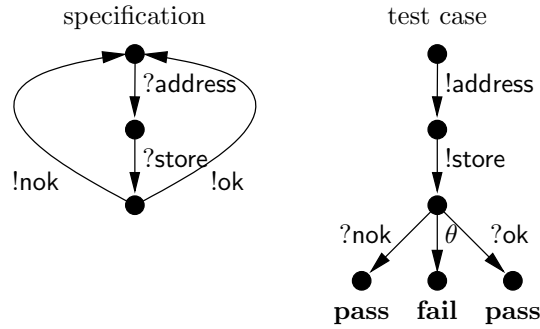


Fig. 1

SPECIFICATION AND TEST CASE OF DATA ENTRY SUB SYSTEM

passes if we observe *ok* or *nok*, but fails when we observe quiescence (θ denotes the observation of quiescence). Note that the direction of inputs and outputs are reversed in the test case. At a certain moment we find out that our implementation differs from our specification in that the *address* data is entered in three steps: street data followed by city data followed by postal code data. This creates the problem that our test case can not be used directly on the implementation, because our implementation does not understand the *abstract* action *address*.

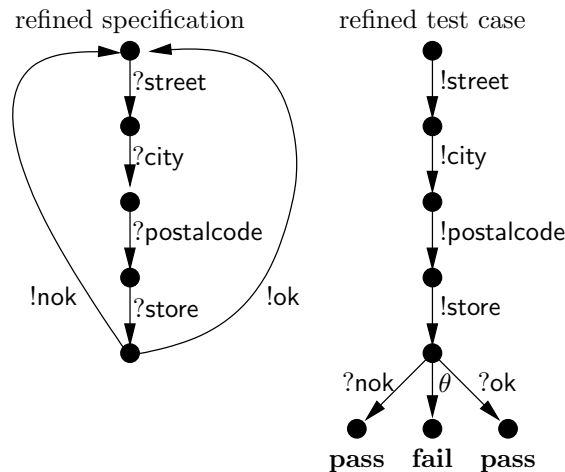


Fig. 2

REFINED SPECIFICATION AND TEST CASE OF DATA ENTRY EXAMPLE

There are basically two solutions to solve this problem. We either refine the *address* action in our specification and regenerate our test cases, or we refine the *address* action directly in our test case. Figure 2 shows a refined specification (left) and a refined test case (right). We see that the specification is refined in a straight forward way by replacing the *address* transition with a sequence of three transitions: *street*, *city*, *postalcode*. If we assume that this specification has the right level of detail, then we can use the test case on the right directly on our implementation. □

Of course the data entry example is very simple, because it is chosen for educational purposes. This may give the illusion that refinement of transition systems and test cases is straightforward. Our next example illustrates that very simple refinements may quickly result in a complex system.

Example III.2: In Figure 3, we see the abstract specification (left hand side) and the refined specification (right hand side) of a game machine. The abstract specification tells us to insert €3 and either press the “game” button to play a video game or press the “reject” button to get the money back. The refined specification (right hand side) is obtained after two refinements. One refinement is that the €3 input action is refined to €2 followed by €1 or vice versa. In between the coins we can press the refund button to get the money back. Likewise the €3 output after pushing the “reject” button is in terms of €1 and €2 coins. To keep the figure readable we left out the refinements for other coins. □

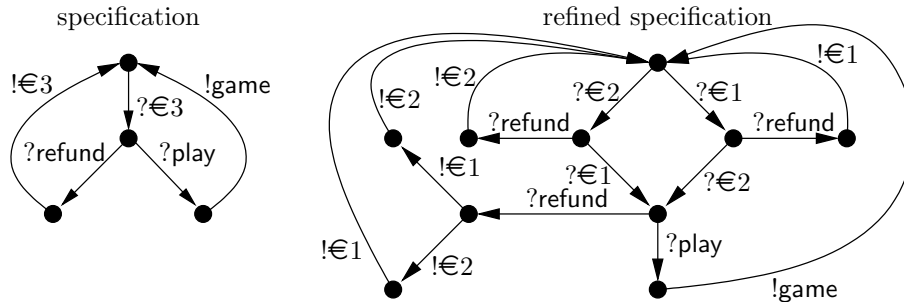


Fig. 3

MORE COMPLICATED ACTION REFINEMENT EXAMPLE

So the core of our problem is that we want to obtain concrete test cases from an abstract specification. There are several solutions to this problem. Figure 4 shows our general approach for action refinement. We see six boxes in the figure. The boxes on the left hand side denote specifications and implementations and the boxes on the right hand side denote test suites. When we start top left we see an abstract system specification. This is an abstraction of the box below: refined system specification. In other words we obtain the refined system specification by refining the abstract system specification. This refined system specification has the needed level of detail and can be implemented to a system implementation that runs in the physical world (as opposed to the mathematical world where our specifications are part of). We can relate the system implementation and the abstract and refined system specification via a conformance relation. This relation expresses if the system implementation conforms to the specification; if the system implementation is a correct implementation according to the specification. We find out whether or not this conformance relation holds by applying tests to the IUT. In model based testing these tests are derived from the specification. Right top in the figure we see that we obtain an abstract test suite from the abstract specification. In order to get test cases with the right level of detail we refine them to a refined test suite. These test cases we can implement and execute to give a verdict whether or not our conformance relation holds. Or in other words, whether or not we have a correct implementation.

There are several types of action refinement. This can be illustrated with the example in Figure 3 (this list is not complete).

- Input-inputs refinement. The input action €3 is refined into the input actions €2 followed by €1.
- Output-outputs refinement. The output action €3 is refined into the output actions €2 followed by €1.
- Branching action refinement. The input action €3 is refined to a choice between €2 followed by €1 or €1 followed by €2.
- Atomic action refinement. No actions are allowed within the refined €3 output actions. In other words the refinement of the output action €3 is treated as if it occurs atomically.
- Non-atomic action refinement. The input action “refund” is allowed to occur between the refined input actions, whereas in the specification the action occurs after the €3 input action.
- “Are you sure” refinement. This is the case when we are first asked if we really want to do a certain action, for example uninstall an application. In terms of our video game, we can imagine that after pressing the game button the system asks if we are sure that we want to play the game and offers a possibility to abort or

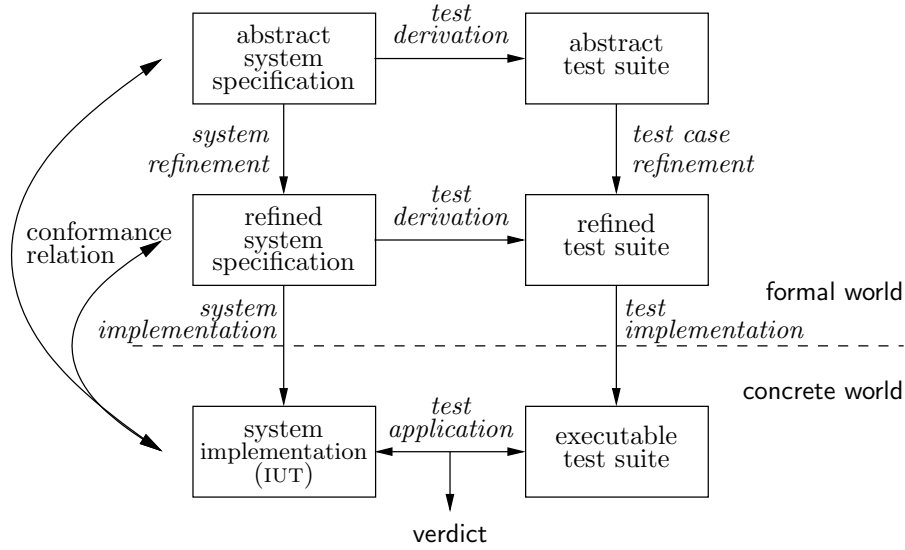


Fig. 4

ACTION REFINEMENT APPROACH

continue.

In this paper we will fill in the parts of the approach in Figure 4 for a special case of action refinement called *atomic linear input-inputs refinement* as a first exercise. It is our goal to extend this approach to more general cases of action refinement. We believe that this can be done in a way very similar to the atomic linear input-inputs refinement case that we treat in this paper, as we will discuss in the concluding section.

The rest of this paper deals with our specific type of atomic linear input-inputs refinement. Sometimes we use the terms abstract, respectively concrete as synonyms for unrefined respectively refined.

IV. TRACE REFINEMENT

The goal of trace refinement is to refine a trace from an abstract specification in such a way that it becomes a trace of the refined system. A refined trace is a trace in which all occurrences of an abstract label have been replaced with its refinement. Formally we denote the relation between the abstract refinement label and the concrete refinement trace as follows.

Definition IV.1: Let L denote the label set from the abstract system. $a_r \in L$ denotes the abstract label that we want to refine. We use $\sigma_r \notin L_\delta^*$ to denote the concrete refinement trace. We define refinement as a pair: $r = (a_r, \sigma_r)$

Because we treat the special case of input-inputs refinement we allow quiescence within a refinement. In order to get all the possible traces within the refinement trace, we saturate the refinement trace with δ 's, as is defined as follows.

Definition IV.2: [δ -saturation] Let $\sigma = a_1 \cdots a_n$ then $[\sigma] = a_1 \cdot \delta^* \cdot a_2 \cdots \delta^* \cdot a_n$

A refinement of a trace results in a set of traces. All labels except the refinement label a_r are unchanged. The refinement label is substituted with every trace in the delta saturated set of σ_r : $[\sigma_r]$, therefore we obtain a set of traces. Formally this is expressed as follows.

Definition IV.3: [Trace refinement] Let $\sigma \in L_\delta^*$ then σr denotes the refinement of a trace in the following way.

$$(\sigma)[r] = \begin{cases} 1) \{\epsilon\} & \text{if } \sigma = \epsilon \\ 2) \{\sigma_2 \cdot \lambda \mid \sigma_2 \in \sigma_1[r]\} & \text{if } \sigma = \sigma_1 \cdot \lambda \wedge \lambda \in L_\delta \setminus \{a_r\} \\ 3) \{\sigma_2 \cdot \sigma' \mid \sigma_2 \in \sigma_1[r] \wedge \sigma' \in [\sigma_r]\} & \text{if } \sigma = \sigma_1 \cdot a_r \end{cases}$$

Likewise we define refinement on sets of traces by refining all the traces in the set.

An important concept in this paper is the concept of an *r-complete* trace. With this we mean a trace that does not end in the middle of a refinement. The set of r-complete traces is denoted by \mathbf{R}_r .

Definition IV.4: [r-complete] $\mathbf{rc}_r(\sigma) =_{\text{def}} \exists \sigma' \in \mathbf{L}_\delta^* : \sigma \in \sigma' r$

Example IV.5: Let us illustrate trace refinement with our running example in Figure 1 and 2. Our refinement pair is $r = (\text{address}, \text{street} \cdot \text{city} \cdot \text{postalcode})$. Suppose we want to refine the trace $\text{address} \cdot \text{store} \cdot \text{ok}$. This results in the following:

$$\begin{aligned} (\text{address} \cdot \text{store} \cdot \text{ok})r &= (\text{address} \cdot \text{store})r \cdot \text{ok} && \text{(step 2)} \\ &= \text{address}r \cdot \text{store} \cdot \text{ok} && \text{(step 2)} \\ &= \text{street} \cdot \delta^* \cdot \text{city} \cdot \delta^* \cdot \text{postalcode} \cdot \text{store} \cdot \text{ok} && \text{(step 3)} \end{aligned}$$

Note that all these refined traces are r-complete (none ends with a concrete label, or a prefix of the δ -saturation of the concrete refinement trace). \square

Trace *contraction* is the opposite of trace refinement. The goal of trace contraction is to transform a concrete trace to a trace of the abstract system.

Definition IV.6: [Trace contraction] Let $r = (a_r, \sigma_r), a_r \notin L_r$.

$$\sigma \langle r \rangle = \begin{cases} 1) \epsilon & \text{if } \sigma = \epsilon \\ 2) \sigma_1 \langle r \rangle \cdot a_r & \text{if } \sigma = \sigma_1 \cdot \sigma_2 : \sigma_2 \in [\sigma_r] \\ 3) \sigma_1 \langle r \rangle & \text{if } \sigma = \sigma_1 \cdot \sigma_2 \wedge \exists \sigma_3 \in [\sigma_r] : \epsilon \sqsubseteq \sigma_2 \sqsubseteq \sigma_3 \\ 4) \sigma_1 \langle r \rangle \cdot \lambda & \text{if } \sigma = \sigma_1 \cdot \lambda \text{ and none of the above holds} \end{cases}$$

Likewise we define contraction on sets of traces by contracting the traces in the set.

Example IV.7: To illustrate trace contraction with our running example, suppose that we want to contract the trace $\text{street} \cdot \delta \cdot \text{city} \cdot \text{postalcode} \cdot \text{store} \cdot \text{ok} \cdot \text{street} \cdot \delta$. Note that this trace is not r-complete, but it is a prefix of an r-complete trace. We obtain the following:

$$\begin{aligned} (\text{street} \cdot \delta \cdot \text{city} \cdot \text{postalcode} \cdot \text{store} \cdot \text{ok} \cdot \text{street} \cdot \delta) \langle r \rangle & \\ &= (\text{street} \cdot \delta \cdot \text{city} \cdot \text{postalcode} \cdot \text{store} \cdot \text{ok}) \langle r \rangle && \text{(rule 3)} \\ &= (\text{street} \cdot \delta \cdot \text{city} \cdot \text{postalcode} \cdot \text{store}) \langle r \rangle \cdot \text{ok} && \text{(rule 4)} \\ &= (\text{street} \cdot \delta \cdot \text{city} \cdot \text{postalcode}) \langle r \rangle \cdot \text{store} \cdot \text{ok} && \text{(rule 4)} \\ &= \text{address} \cdot \text{store} \cdot \text{ok} && \text{(rule 2)} \end{aligned}$$

\square

In order for contraction to be the dual of refinement, r-complete traces are important. The following proposition states that the application of contraction after refinement on a trace σ results in $\{\sigma\}$. This corresponds with our intuition that contraction is the opposite of refinement. Note that this lemma does not hold for non r-complete traces.

Proposition IV.8: Let $\sigma \in L_\delta^*$ and $r = (a_r, \sigma_r)$: $\sigma r \langle r \rangle = \{\sigma\}$

Likewise, the application of contraction followed by refinement on an r-complete trace σ , results in a set of which σ is a member.

Lemma IV.9: $\mathbf{rc}_r(\sigma) \Leftrightarrow \sigma \in \sigma \langle r \rangle r$

Example IV.10: We illustrate Proposition IV.8 and Lemma IV.9 by using the results of Example IV.5 and Example IV.7:

$$\begin{aligned} (\text{address} \cdot \text{store} \cdot \text{ok})r &= \text{street} \cdot \delta^* \cdot \text{city} \cdot \delta^* \cdot \text{postalcode} \cdot \text{store} \cdot \text{ok} \\ (\text{address} \cdot \text{store} \cdot \text{ok})r \langle r \rangle &= \{\text{address} \cdot \text{store} \cdot \text{ok}\} \\ \\ (\text{street} \cdot \delta \cdot \text{city} \cdot \text{postalcode} \cdot \text{store} \cdot \text{ok}) \langle r \rangle &= \text{address} \cdot \text{store} \cdot \text{ok} \\ (\text{street} \cdot \delta \cdot \text{city} \cdot \text{postalcode} \cdot \text{store} \cdot \text{ok}) \langle r \rangle r &= \text{street} \cdot \delta^* \cdot \text{city} \cdot \delta^* \cdot \text{postalcode} \cdot \text{store} \cdot \text{ok} \end{aligned}$$

Note that in the last two lines $\text{street} \cdot \delta \cdot \text{city} \cdot \text{postalcode} \cdot \text{store} \cdot \text{ok}$ is an element of $(\text{address} \cdot \text{store} \cdot \text{ok})r$ and therefore r-complete. \square

A property that is important for *Utraces* is that trace preorder is maintained over trace contraction. We come back to this property in Section VI.

Lemma IV.11: Let $r = (a_r, r), \sigma \in \downarrow \mathbf{R}_r$: $\sigma' \sqsubseteq \sigma \implies \sigma' \langle r \rangle \sqsubseteq \sigma \langle r \rangle$

V. ATOMIC REFINEMENT OF TRANSITION SYSTEMS

In this section we present a way to refine transition systems. Basically what we do is that we make a transition system from our refinement trace and insert this into the abstract transition system at exactly the place where there are transitions with the abstract refinement label. Formally we define its refinement as follows. Note that we use a trick in the labeling of states: for a a_r -transition $t = (q, a_r, q')$, we use $(t, 0) = q$ and $(t, n) = q'$ for $n = |\sigma_r|$. This makes it possible to have concrete refinement traces with one label.

Definition V.1: [Atomic transition system refinement] Let $p = \langle Q, I, U, T, q_0 \rangle$ be an LTS and let $r = (a_r, \sigma_r)$ be the refinement function, with $I(\sigma_r)$ the input labels in the refinement trace. All the labels in σ_r should be “fresh”: $L \cap L(\sigma_r) = \emptyset$. We now define a function r that transforms p into an LTS reflecting the given action refinement function. $pr = \langle Q_r, I_r, U_r, T_r, q_0 \rangle$, where

$$Q_r = Q \cup \{(t, i) \mid \exists q, q' \in Q : t = (q, a_r, q') \in T, 1 \leq i < n = |\sigma_r|\}$$

$$I_r = I \setminus \{a_r\} \cup I(\sigma_r)$$

$$T' = \{((t, i), \sigma_r|_{i+1}, (t, i+1)) \mid \exists q, q' \in Q : t = (q, a_r, q') \in T, 0 \leq i \leq |\sigma_r| - 1\}$$

$$T_r = \{(q, a, q') \in T \mid a \neq a_r\} \cup T'$$

For transitions that take place in the refined LTS we use a subscript “ r ” with the transition arrow, like in $q \xrightarrow{\sigma_r} q'$.

Example V.2: We use our running example to explain Definition V.1. Figure 5 shows again the abstract and refined specification for our data entry sub system. This time we numbered the states in order to refer to our lts-refinement definition. Let us start with the states. For the abstract transition $t = (q_0, \text{address}, q_1)$ we add the states $(t, 1)$ and $(t, 2)$. Note that $(t, 0)$ and $(t, 3)$ correspond to states q_0 and q_1 respectively. Next we add the following transitions: $(q_0, \text{street}, (t, 1))$, $((t, 1), \text{city}, (t, 2))$ and $((t, 2), \text{postalcode}, (t, 3))$. After we delete the original **address** transition and we obtain T_r . Last but not least we update the input label set, by adding all the labels from the concrete refinement trace: **{street, city, postalcode}** and deleting the abstract label “**address**”. Because we treat input-inputs refinement, the output label set stays the same. \square

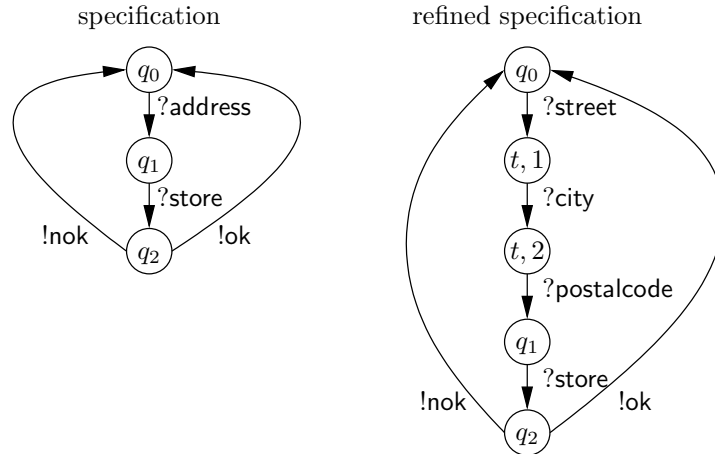


Fig. 5

EXAMPLE OF LTS REFINEMENT

In order to relate refined traces to refined lts’s we have the following two lemmas. Lemma V.3 expresses that r-complete traces always end in an old state, i.e., a state from the abstract system. Vice versa, a trace that ends in an old state is r-complete.

Lemma V.4 expresses that if we can do a trace σ between two states in the abstract system, then in the refined system we can do all refinements of σ between these states. Vice versa, if in the refined system we can do a trace σ' between two old states then we can do the contracted trace in the abstract system between these two states.

Lemma V.3: $\forall q, q' \in Q_r : q \xrightarrow{\sigma} q' \implies (q' \in Q \cap Q_r \Leftrightarrow \mathbf{rc}_r(\sigma))$

Lemma V.4: $\forall q, q' \in Q_r, \sigma \in L_\delta^*, \sigma' \in \sigma r : q \xrightarrow{\sigma} q' \Leftrightarrow q \xrightarrow{\sigma'}_r q'$

VI. \mathbf{uioco}_r FOR TESTING REFINED SYSTEMS

In order to express correctness of the concrete implementation in terms of the abstract specification and the refinement pair, we need a new implementation relation. In this section we define such a relation: \mathbf{uioco}_r and we show that it is equivalent to the \mathbf{uioco} relation over refined specifications.

Definition VI.1: [\mathbf{uioco}_r for atomic input-inputs refinement] Let $s \in \mathcal{LTS}(I_1, U)$, $i \in \mathcal{IOTS}(I_2, U)$, $r = (a_r, \sigma_r)$, $I_2 = I_1 \setminus \{a_r\} \cup I(\sigma_r)$.

$$i \mathbf{uioco}_r s \stackrel{\text{def}}{=} \forall \sigma \in \downarrow(\text{Utraces}(s)r) : \\ \text{if } \sigma \in \text{Utraces}(s)r \text{ then } \text{out}(i \mathbf{after} \sigma) \subseteq \text{out}(s \mathbf{after} \sigma(r)) \\ \text{else } \text{out}(i \mathbf{after} \sigma) \subseteq \{\delta\}$$

To explain this definition, we look at the observable behavior of the refined specification. There are four places in the refined specification that matter, of which three happen to be equivalent because of our special type of action refinement:

- We are completely outside the refinement (also not on the border). In this case the observable behavior of the refined system is completely the same as the abstract system.
- We are adjacent to the refinement. This means that in the abstract specification the refinement action is enabled. It is a special situation, because both observable behavior from the abstract specification, as observable behavior from the refinement are involved. The behavior of the refinement is an input action and this is the same as our refinement label (because of our special atomic linear input-inputs refinement). As a result the observable behavior of refined specification is the same as the behavior of the abstract system.
- We are at the end of the refinement. This is a similar case as the case above and the behavior of the refined specification is the same as the behavior of the abstract system (again because of our special refinement type).
- We are inside the refinement. In this case the observable behavior of the refined system is completely determined by the behavior of the refinement. Since we treat the special case of atomic linear input-inputs refinement, we know that the behavior within the refinement can be at most quiescent. Furthermore we know that Utraces that are not r -complete end within the refinement. This explains the **else** clause in the definition.

As we can see, the first three cases are covered if a trace σ in $\downarrow(\text{Utraces}(s)r)$ happens to be an r -complete trace: $\sigma \in \text{Utraces}(s)r$. Therefore we require the observable behavior of the implementation to be included in the behavior of the abstract specification after contraction of the trace: $\text{out}(i \mathbf{after} \sigma) \subseteq \text{out}(s \mathbf{after} \sigma(r))$. As said, the last case is covered in the **else** part of the definition of \mathbf{uioco}_r : $\text{out}(i \mathbf{after} \sigma) \subseteq \{\delta\}$.

Theorem VI.2: Let $s \in \mathcal{LTS}(I_1, U)$, $i \in \mathcal{IOTS}(I_2, U)$, $r = (a_r, \sigma_r)$

$$i \mathbf{uioco}_r s \Leftrightarrow i \mathbf{uioco} s[r]$$

The difficult part in understanding this equality is that the prefix closure of the refined utrares of the abstract specification is equal to the set of utrares from the refined specification. This may look straightforward, but it is not trivial. The main reason this equality holds is that atomic linear input-inputs refinement does not introduce underspecification of input actions. With this we mean that there is no state that is underspecified for a certain input action, if there is another state (reachable by the same trace) that can do that specific input action.

Example VI.3: Let us look again at Figure 5. On the left hand side we see the abstract specification and on the right hand side the concrete specification. Definition VI.1 identifies two possibilities: either the trace is in $\text{Utraces}(s)r$ or not. If this is not the case then the output of the implementation should be a subset of $\{\delta\}$. With the help of lemma V.3 we see why. Namely, a trace that is not in $\text{Utraces}(s)r$ is not r -complete and a non r -complete trace ends in a new state in the concrete specification. This means that it ends in the middle of the refinement; these are the states $(t, 1)$ and $(t, 2)$ in the concrete specification in Figure 5. In these states, only δ is possible.

Traces in $\text{Utraces}(s)r$ are r -complete. This means that they end in an abstract state. As explained above, we identify three types of abstract: the beginning state of an a_r transition, the ending state of such a transition and a state that is not part of such a transition. In the latter case the transitions that are enabled in an

abstract state in the abstract system are exactly the same as the transitions enabled in the concrete system, like is the case in state q_2 . The same holds for ending states of an a_r transition, like state q_1 . Beginning states of an a_r transition, like state q_0 form a special case. Because a_r is an input action, δ is an output action in the abstract specification if no output actions or internal actions are enabled. In the concrete specification, the a_r transition has been changed to a transition with the first concrete refinement trace action. Because this is also an input action, the δ action is again enabled. \square

VII. CONCLUSION

In this paper we proposed an approach for atomic linear input-inputs refinement, see Figure 4. We applied this approach to atomic linear input-inputs refinement. For this special case of action refinement we showed how to refine traces and transition systems. Furthermore we introduced the implementation relation \mathbf{uioco}_r for concrete implementations that relates the abstract specification to the concrete implementation by using the refinement pair. We showed that \mathbf{uioco}_r is equivalent to the \mathbf{uioco} implementation relation over the refined specification.

Current research Our current research focuses on refinement of test cases. This fills in the next boxes of Figure 4 (the middle two). Our goal is that if we start with a complete abstract test suite that the test suite remains complete after refinement. For the original \mathbf{ioco} theory there is a completeness proof for the test generation algorithm. There are several possibilities to repeat such a proof for test case refinement. We could for example come up with a test derivation algorithm for \mathbf{uioco}_r and proof is equivalent with the test derivation for refined specifications. Another approach is to proof refined test cases equivalent to test cases that are derived from a refined specification. Because test cases are special LTS's test case refinement in this form would be similar, but slightly different to LTS refinement. We have therefore chosen to express test cases as sets of traces (trees). The result is that test case refinement reduces to trace refinement. In return we have to express soundness and completeness as properties of these test cases (as sets of traces). This is not trivial, but would facilitate our current and future work.

Future work We found out that it is surprisingly difficult to do use our approach for such a simple form of action refinement as atomic linear input-inputs refinement is. One of the main reasons for this is that we use *Utraces* in stead of the regular *Straces*. Refinement of *Utraces* from the abstract system should result in *Utraces* from the refined system. This is not trivial to show. With the knowledge gained from this exercise we see however that the approach is rather complete for other forms of action refinement, like arbitrary atomic action refinement. This is still atomic, so no actions are allowed to interfere the refinement, but we drop the linearity and input-inputs constraints. As a result we allow branching (including looping) behavior with a mix of input and output actions.

An important concept that will be reused is “r-completeness”. The refined transition system will show exactly the same behavior after a completely refined trace as the abstract system after the contracted trace. The behavior of the refined transition system after an r-incomplete trace can be defined in terms of the refinement lts. Just like we did in our case: the allowed output after an r-incomplete trace is quiescence. We will briefly cover the changes that this approach will have.

- Trace refinement and trace contraction will be defined in terms of suspension traces of the abstract system and the refinement LTS (or subsets hereof, like *Utraces*). This means that the refinement pair is between an action and an LTS, instead of an action and a trace.
- For LTS refinement transitions with the refinement label (q, a_r, q') will be tied to the refinement LTS. The beginning state q of the transition will coincide with the beginning state of the refinement LTS and the ending state of the transition q' will coincide with the ending state of the refinement LTS.
- The definition of an implementation relation will be split up in behavior for traces that are r-complete and traces that are not. For r-complete traces we use the behavior of the abstract system and for r-incomplete traces we take the behavior of the refinement LTS. Some small changes need to be made. For example, refinement of output actions has an effect of the observable behavior of the refined system in a more complex way than for the quiescence in input-inputs refinement.

As a result we will be able to refine the video game example from our introduction. In Figure 6 we show the

refinements lts's for the refinement of the input action $?€3$ (left hand side) and the output action $!€3$ (right hand side). The states that are labeled with 'B' and 'E' are the beginning respectively the ending states of the refinement lts. When we substitute these two lts's for their respective refinement labels in the abstract specification in Figure 3 (left hand side) we obtain the refined transition system on the right hand side.

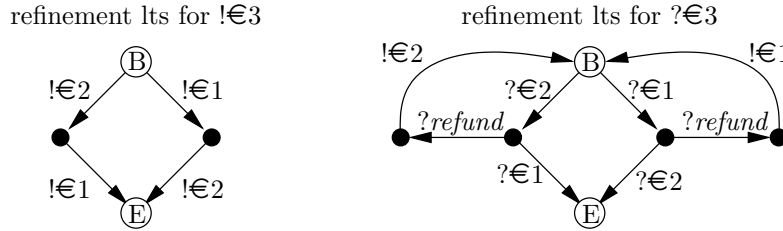


Fig. 6

EXAMPLE OF ATOMIC LTS REFINEMENT

Our future research will first focus on atomic action refinement, but the interesting work is of course *non-atomic action refinement*. At the moment we are collecting case studies to understand this type of action refinement in greater detail.

REFERENCES

- [1] G. Bernot, M. G. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 1991(November):387–405, 1991. Also: Rapport de Recherche 581, L.R.I., Université de Paris-Sud.
- [2] R. Gorrieri and A. Rensink. Action refinement. In J. A. Bergstra, A. Ponse, and S. A. Smolka, editors, *Handbook of Process Algebra*, chapter 16, pages 1047–1147. Elsevier, 2001.
- [3] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.
- [4] M. van der Bijl, A. Rensink, and J. Tretmans. Compositional testing with ioco. In A. Petrenko and A. Ulrich, editors, *FATES 2003*, volume 2931 of *Lecture Notes in Computer Science*, pages 86–100. Springer, 2004.