Action Refinement in Conformance Testing

Machiel van der Bijl[1⋆], Arend Rensink[1], and Jan Tretmans[2]

[1] Software Engineering, Department of Computer Science, University of Twente,
P.O. Box 217, 7500 AE Enschede, The Netherlands,
{vdbijl, rensink}@cs.utwente.nl
[2] Informatics for Technical Applications,
Nijmegen Institute for Computing and Information Sciences (NIII),
Radboud University, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands,
tretmans@cs.ru.nl

Abstract. In *model based testing* test cases are derived from a model (the specification) of the system we want to test. In general the model 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. The standard approach is to rewrite the model by hand to the required level of detail and regenerate the test cases. This is error-prone and time consuming.
In this paper we present an approach to automatically obtain test cases at the required level of detail by means of action refinement. Action refinement is a way to add information to the abstract model. It relates actions from the abstract model to concrete actions of the system under test. We apply this approach to a simple case of action refinement, so-called atomic linear input-inputs refinement. In order to reason about correctness between an abstract model and a concrete implementation we introduce a new implementation relation. We show that this relation is equivalent with the **uioco** implementation relation on the refined model. Furthermore we show under which conditions the refinement of a complete abstract test suite is again complete.

1 Introduction

A problem in model based testing is that the generated test cases may not have the required level of detail, and hence are not executable against the implementation under test. The test cases are generated from the model (the specification) and in general, the model is more abstract than the implementation. The usual solution is to add the required level of detail to the model by hand. This has some obvious drawbacks; it is time consuming and error-prone.

In this paper we use *action refinement* to automatically obtain test cases at the required level of detail. Action refinement has been studied extensively; see Gorrieri and Rensink for an overview [2]. Action refinement adds extra information to the model by relating an action of the model to more detailed behavior.

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

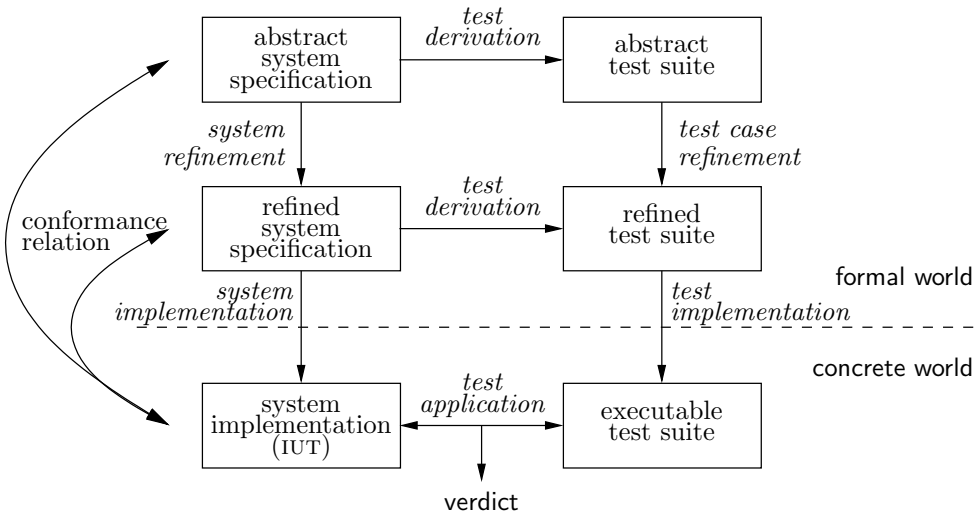

Fig. 1. Action refinement approach

Wherever we read the action in the model we replace it with the more detailed behavior. For example, suppose that the model specifies to input two euros and the implementation also allows the insertion of two one euro pieces. With action refinement we can define that wherever we read two euros we can also read the more detailed behavior one euro followed by one euro. Action refinement in model based testing has not been studied at all. This is surprising, because it is a well known problem in practice and occurs often.

Figure 1 shows our general approach for action refinement in testing. We see six objects in the figure. The objects on the left hand side denote models and the objects on the right hand side denote test suites. **System implementation** is the system that we want to test, also known as IUT (Implementation Under Test); a real system in the physical world. **Abstract system specification** is a (formal) model of the system implementation. It is called *abstract* because it does not have the required level of detail with respect to the system implementation. **Refined system specification** is the refined model of the system implementation with the required level of abstraction with respect to the system implementation. **Abstract test suite** is the test suite that is derived from the abstract system specification. As with the abstract system specification, it is too abstract with respect to the system implementation. **Refined test suite** is a test suite with the required level of abstraction with respect to the system implementation. There are two ways to derive such a test suite. One way is to refine the abstract test suite, another way is to derive test cases from the refined system specification. We do both and proof both approaches to be equivalent under certain restrictions. **Executable test suite** is a test suite in the physical

world that we can execute against the system implementation. This results in a verdict whether or not the implementation is correct with respect to the refined (or abstract) system specification. This notion of correctness is defined in a so-called *implementation relation* between the system specification (abstract or refined) and the system implementation. The conformance relation is depicted on the left side of the Figure.

This paper is a first step in our effort towards action refinement in model based testing and we use a simple, though non-trivial case of action refinement: *atomic linear input-inputs refinement*.

In this paper we show how to refine traces, transition systems and test cases. In order to reason about correctness between an abstract specification and a concrete implementation we introduce the implementation relation $\mathbf{uioco}_r$ and we show that it is equivalent with **uioco** between the refined specification and the same implementation (**uioco** is a further evolution of **ioco**; see [4] and [6]). We show under which conditions the refinement of a complete abstract test suite results in a complete refined test suite.

The main contribution of this paper is that refinement of a complete test suite results in a complete refined test suite (under certain restrictions). Furthermore we argue that our approach for atomic linear input-inputs refinement can be extended to more general types of action refinement. This extension is the next step in our research. One of the surprising (theoretic) consequences of this paper is that specification equivalence is not preserved by action refinement.

We start with summarizing some results and notations that we will use throughout the paper in Section 2. In Section 3 we introduce atomic linear input-inputs refinement. We present trace refinement in Section 4 and the refinement of labeled transition systems in Section 5. In Section 6 we present the implementation relation $\mathbf{uioco}_r$, followed by the refinement of test cases in Section 7. Conclusions can be found in Section 8.

2 Formal preliminaries

This section recalls some aspects of the theory behind **uioco** that are used in this paper; see [6] and [4] 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 partition the label set of an LTS in an input and output set; a deviation from the standard definition of labeled transition systems.

Definition 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*,* $I \cap U = \emptyset$*;* $T \subseteq Q \times (I \cup U \cup \{\tau\}) \times Q$ *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 $q \xrightarrow{\mu} q'$ for $(q, \mu, q') \in T$. We use a question mark before a label to denote that it is input and an exclamation mark to denote that it is 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 stable state from which no output action is possible is called *quiescent*. We use the symbol δ ($\notin \mathbf{L}_\tau$) to represent quiescence: $p \xrightarrow{\delta} p$ stands for the absence of any transition $p \xrightarrow{\mu} p'$ with $\mu \in U_\tau$. For an arbitrary $L \subseteq \mathbf{L}$, we use L_δ as a shorthand for $L \cup \{\delta\}$. We use the label μ, respectively λ to range over $\mathbf{L}_\tau$, respectively $\mathbf{L}_{\tau\delta}$.

An LTS is *strongly responsive* if it always eventually enters a quiescent state; in other words, if it does not have any infinite U_τ-labeled paths. The **ioco** theory is restricted to strongly responsive systems, hence we also use this restriction.

A *trace* is a 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. Concatenation is extended in the standard way to sets of traces and also to $\Sigma \cdot a$ where Σ is a set of traces and a an action. We use the standard notation with single and double arrows for traces: $q \xrightarrow{\lambda_1 \cdots \lambda_n} q$ denotes $q \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_n} q'$, $q \stackrel{\epsilon}{\Longrightarrow} q'$ denotes $q \xrightarrow{\tau \cdots \tau} q'$ and $q \stackrel{\lambda_1 \cdots \lambda_n}{\Longrightarrow} q$ denotes $q \stackrel{\epsilon}{\Longrightarrow} \xrightarrow{\lambda_1} \stackrel{\epsilon}{\Longrightarrow} \cdots \xrightarrow{\lambda_n} \stackrel{\epsilon}{\Longrightarrow} q'$. We will use Σ to denote a set of traces. If $\sigma = \lambda_1 \cdots \lambda_n$ then $\sigma | i = \lambda_i$ where $1 \leq i \leq |\sigma| = n$, and $L(\sigma) = \{\lambda_1, \cdots, \lambda_n\}$. We use the symbol $\sqsubseteq$ to denote trace prefix and the symbol $\downarrow$ to denote prefix closure, as follows: $\sigma_1 \sqsubseteq \sigma \Leftrightarrow \exists \sigma_2 : \sigma_1 \cdot \sigma_2 = \sigma$, $\downarrow \sigma = \{\sigma' \mid \sigma' \sqsubseteq \sigma\}$, $\downarrow \Sigma = \bigcup \{\downarrow \sigma \mid \sigma \in \Sigma\}$

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, $p \stackrel{\sigma}{\Longrightarrow} q_1$ instead of $q_0 \stackrel{\sigma}{\Longrightarrow} q_1$.

Input-output transition systems. We call a labeled transition system that is completely specified for input actions an *input-output transition system* (IOTS). This means that all states can do all input actions from the label set, if necessary by first doing one or more internal actions. The class of input-output transition systems with input actions in I and output actions in U is denoted by $\mathcal{IOTS}(I, U)$ ($\subseteq \mathcal{LTS}(I, U)$).

Definition 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:* $\forall q \in Q, a \in I : q \stackrel{a}{\Longrightarrow}$ *(*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 [4]. 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 [1].

Given a specification s and an (assumed) model of the implementation i, the relation i $\mathbf{ioco}_{\mathcal{F}}$ s expresses that i conforms to s based on a set of traces $\mathcal{F}$. This is formalized as follows: Let $s \in \mathcal{LTS}(I,U)$, $i \in \mathcal{IOTS}(I,U)$, $S \subseteq Q_s$ be a set of states in s, $\sigma \in \mathbf{L}_\delta^*$ and $\mathcal{F} \subseteq L_\delta^*$.

$$s \,\mathbf{after}\, \sigma =_{\text{def}} \{s' \mid s \stackrel{\sigma}{\Longrightarrow} s'\} \tag{1}$$

$$out(s) =_{\text{def}} \{x \in U \mid s \xrightarrow{x}\} \cup \{\delta \mid s \xrightarrow{\delta}\} \tag{2}$$

$$out(S) =_{\text{def}} \bigcup \{out(s) \mid s \in S\} \tag{3}$$

$$Straces(s) =_{\text{def}} \{\sigma \in L_\delta^* \mid s \stackrel{\sigma}{\Longrightarrow}\} \tag{4}$$

$$Utraces(s) =_{\text{def}} \{\sigma \in Straces(s) \mid \forall q, (\sigma_1 \cdot a) \sqsubseteq \sigma : (a \in I \wedge s \stackrel{\sigma_1}{\Longrightarrow} q) \text{ implies } q \stackrel{a}{\Longrightarrow}\} \tag{5}$$

$$i \;\mathbf{ioco}_{\mathcal{F}}\; s =_{\text{def}} \forall \sigma \in \mathcal{F} : out(i \,\mathbf{after}\, \sigma) \subseteq out(s \,\mathbf{after}\, \sigma) \tag{6}$$

For $\mathcal{F} = Straces(s)$ we abbreviate $\mathbf{ioco}_{\mathcal{F}}$ to **ioco** and for $\mathcal{F} = Utraces(s)$ to **uioco**. In other words **ioco** is based on suspension traces (*Straces*: traces in L_δ^*) whereas **uioco** is based on a subset of suspension traces: universal traces. All states that a universal trace leads to can do the same set of input actions. This is a necessary prerequisite to make **uioco** a pre-congruence for parallel composition and hiding.

Test cases. A test case is the specification of a tester in an experiment with the system under test. It is modeled as a special labeled transition system with **pass** and **fail** predicates on states to decide about the success of a test. It is a special LTS because it has the following restrictions:

Definition 3. *A test case* $t = \langle Q, S, R, T, t_0, \mathbf{pass}, \mathbf{fail}\rangle$ *over a set of stimuli* S *and a set of responses* R *is an acyclic labeled transition system such that:*
- t *is deterministic and has finite behavior.*
- $\mathbf{pass} \subseteq Q, \mathbf{fail} \subseteq Q$. **pass** *and* **fail** *states do not have outgoing transitions.*
- *A state in* Q *that is no* **pass** *or* **fail** *state has either* one *outgoing transition with a stimulus label, or has outgoing transitions for* all *labels in* R.

The class of test cases over S and R is denoted as $\mathcal{TEST}(S,R)$. A *test suite* T is a set of test cases: $T \subseteq \mathcal{TEST}(S,R)$. An implementation $i \in \mathcal{IOTS}(I,U)$ **passes** a test case $t \in \mathcal{TEST}(I,U_\delta)$ if there is no suspension trace of i that leads to a **fail** state in t. Note that a stimulus of the test case is an input of the implementation and vice versa for the responses.

Definition 4. *Let* $s \in \mathcal{LTS}(I,U)$ *be a specification and* $T \subseteq \mathcal{TEST}(I,U_\delta)$ *a test suite; then for the implementation relation* **uioco***:*

T is **complete** $=_{\text{def}} \forall i \in \mathcal{IOTS}(I,U) : i \;\mathbf{uioco}\; s \Leftrightarrow i \;\mathbf{passes}\; T$
T is **sound** $=_{\text{def}} \forall i \in \mathcal{IOTS}(I,U) : i \;\mathbf{uioco}\; s \Rightarrow i \;\mathbf{passes}\; T$
T is **exhaustive** $=_{\text{def}} \forall i \in \mathcal{IOTS}(I,U) : i \;\mathbf{uioco}\; s \Leftarrow i \;\mathbf{passes}\; T$

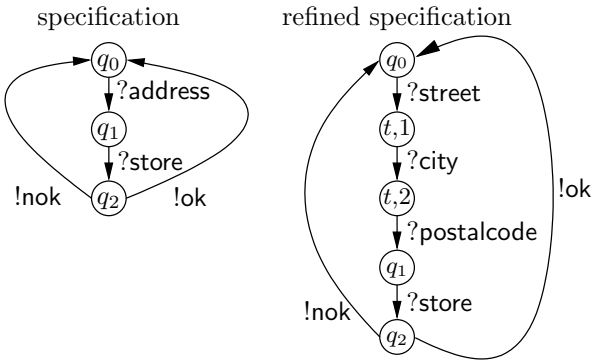

Fig. 2. Abstract and refined specification of data entry system

3 Atomic input-inputs action refinement

As stated in the introduction we treat the problem that test cases derived from a specification may not be executable on the system under test. Example 1 illustrates this problem (we use this as our running example).

Example 1. In Figure 2 we see a specification (left) and a refined specification (right) of a simple data entry application (forget the state labels for now). The specification tells us that we can enter address data, push the store button after which the system either stores the address data (ok) or returns nok. Suppose that our specification is too abstract, because an address is entered in three steps: street, city and postal code, like the refined specification on the right.

The left hand side of Figure 3 shows a test case generated from the abstract specification. On the right we see two test cases with the required level of detail to test the actual system. We can read the abstract test case as follows: enter the address data, press the store button and then observe the response of the IUT. The IUT passes the test if we observe ok or nok, but fails when we observe quiescence. Note that the direction of inputs and outputs in the test case are reversed with respect to the specification.

Of course the data entry example is very simple, because of its educational purposes. This may give the illusion that refinement of transition systems and test cases is straightforward. The more extended technical report of this paper shows that simple refinements may quickly result in a complex system [7].

There are several types of action refinement [5]. In this paper we treat *atomic linear input-inputs refinement*. Atomic means that no actions are allowed to interfere with the refinement; we treat the behavior of the refinement as atomic. Linear means that we allow no branching behavior in the refinement and input-inputs means that we only refine an input action with one or more other input actions. The refinement in Figure 2 is an example of such a refinement. It is

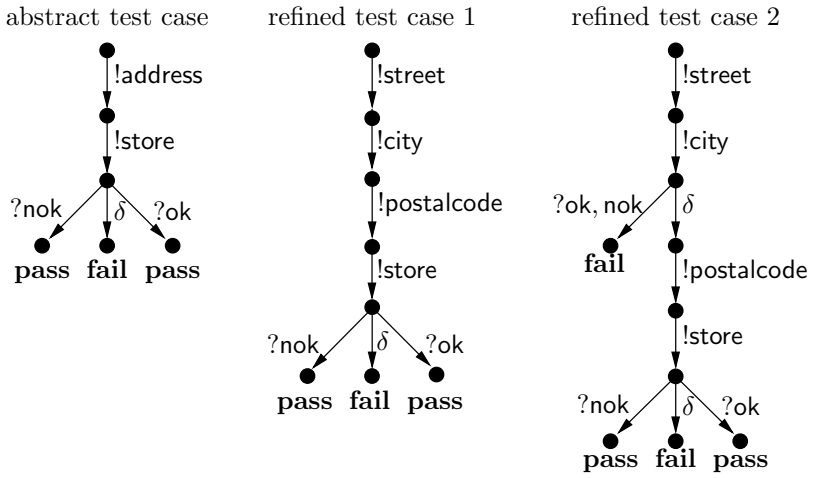

Fig. 3. Abstract and refined test cases for data entry example

our goal to extend this action refinement approach in the future to more general cases of action refinement. We believe that this can be done in a similar way as the atomic linear input-inputs refinement case that we treat in this paper, as we discuss in the concluding section.

In this paper we show what correctness means in terms of a conformance relation between the abstract system specification and the system implementation. Furthermore we show two ways to obtain a refined test suite as shown in Figure 1. One is to refine the abstract system specification and derive a refined test suite and the other is to refine the abstract test suite directly. We show that both approaches are equivalent under some restrictions.

Sometimes we use the terms abstract and concrete as synonyms for unrefined and refined, respectively .

4 Trace refinement

We define refinement as a pair $r = (a_r, \sigma_r)$ with respect to an input label set I and an output label set U. a_r is the *refinement label*, i.e., the abstract label that we want to refine and σ_r is the *refinement trace*, i.e., the trace that we want to replace the refinement label with. There are the following restrictions: $a_r \in I$, $L(\sigma_r) \cap L_\delta = \emptyset$ (the labels in σ_r are fresh) and $\sigma_r \neq \epsilon$.

In cases where there may be confusion about label sets we use the subscript r to tag the label set after refinement, for example: $I_r = (I \backslash \{a_r\}) \cup L(\sigma_r)$.

The goal of trace refinement is to refine a trace from an abstract specification such that it becomes a trace of the refined system. In a refined trace all occurrences of the refinement label have been replaced with its refinement.

Input-inputs refinement allows quiescence within a refinement. To get all possible suspension traces within the refinement trace, we saturate the refinement trace with δ's (this technicality is explained in Example 2).

Definition 5 (δ-saturation). *Let $\sigma = a_1 \cdots a_n$ then $\lceil\sigma\rceil = a_1{\cdot}\delta^*{\cdot}a_2 \cdots \delta^*{\cdot}a_n$*

The 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 $\lceil\sigma_r\rceil$. Formally this is expressed as follows.

Definition 6 (Trace refinement). *Let $\sigma \in L_\delta^*$ then $\sigma[r]$ denotes the refinement of a trace in the following way.*

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

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

An important concept in this paper is the concept of an *r-complete* trace. This is a trace that does not end in the middle of a refinement; or in other words, a trace σ is r-complete when $\sigma \in L_\delta^*[r]$.

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 7 (Trace contraction). *Let $r = (a_r, \sigma_r), \sigma \in \downarrow(L_\delta^*[r])$.*

$$\sigma\langle r\rangle = \begin{cases} 1)\ \epsilon & \text{if } \sigma = \epsilon \\ 2)\ \sigma_1\langle r\rangle{\cdot}a_r & \text{if } \sigma = \sigma_1{\cdot}\sigma_2 \wedge \sigma_2 \in \lceil\sigma_r\rceil \\ 3)\ \sigma_1\langle r\rangle & \text{if } \sigma = \sigma_1{\cdot}\sigma_2 \wedge \sigma_2 \in \downarrow\lceil\sigma_r\rceil \backslash (\lceil\sigma_r\rceil \cup \{\epsilon\}) \\ 4)\ \sigma_1\langle r\rangle{\cdot}\lambda & \text{if } \sigma = \sigma_1{\cdot}\lambda \textit{ and none of the above holds} \end{cases}$$

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

Example 2. Let us illustrate trace refinement and trace contraction with our running example in Figure 2. We refine the action address into street followed by city followed by postalcode: the refinement pair is $r = (\mathsf{address}, \mathsf{street}{\cdot}\mathsf{city}{\cdot}\mathsf{postalcode})$. Suppose we want to refine the trace address·store·ok. This results in the following set of traces of the refined specification.

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

To contract $\mathsf{street}{\cdot}\delta{\cdot}\mathsf{city}{\cdot}\mathsf{postalcode}{\cdot}\mathsf{store}{\cdot}\mathsf{ok}{\cdot}\mathsf{street}{\cdot}\delta$, we obtain the following:

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

5 Atomic refinement of transition systems

In this section we present a way to refine transition systems. The crux of this refinement is that we make a transition system from our refinement trace and insert this into the abstract transition system at the place where there is a transition with the abstract refinement label. A formal definition is given in Definition 8, it is illustrated in Example 3.

Definition 8 (Atomic transition system refinement). *Let $r = (a_r, \sigma_r)$ be the refinement pair and let $p = \langle Q, I, U, T, q_0 \rangle$ be an LTS. We define the refinement of p as $p[r] = \langle Q_r, I_r, U_r, T_r, q_0 \rangle$. For a transition $t = (q, a_r, q')$, we use $(t, 0) = q$ and $(t, n) = q'$ for $n = |\sigma_r|$.*
$Q_r = Q \cup \{(t, i) \mid \exists q, q' \in Q : t = (q, a_r, q') \in T, 1 \leq i < n = |\sigma_r|\}$
$I_r = I \backslash \{a_r\} \cup I(\sigma_r)$
$T' = \{((t, i), \sigma_r|_{i+1}, (t, i+1)) \mid \exists q, q' \in Q : t = (q, a_r, q') \in T, 0 \leq i \leq |\sigma_r| - 1\}$
$T_r = \{(q, a, q') \in T \mid a \neq a_r\} \cup T'$

To prevent confusion between transitions in the abstract and refined transition system we add the subscript 'r' to the transition arrow for refined systems: $q \overset{\sigma}{\Longrightarrow}_r q'$. Likewise we use the subscript for the set of states, transitions, etc., as shown in the definition.

Example 3. We use our running example in Figure 2 to explain Definition 8 (the states are numbered according to this definition). For the abstract transition $t = (q_0, \mathsf{address}, q_1)$ we add the states $(t, 1)$ and $(t, 2)$ to Q_r ($(t, 0)$ and $(t, 3)$ correspond to states q_0 and q_1 respectively). T' consists of the transitions: $((t, 0), \mathsf{street}, (t, 1)), ((t, 1), \mathsf{city}, (t, 2))$ and $((t, 2), \mathsf{postalcode}, (t, 3))$. In T_r we delete the address transition from the set of abstract transitions and we add T'. We add all labels from the refinement trace: $\{\mathsf{street}, \mathsf{city}, \mathsf{postalcode}\}$ to I_r and delete the refinement label "address" (the output label set stays the same).

Lemma 1 states that the prefix closure of the refined *Utraces* of the abstract specification equals the set of *Utraces* of the refined specification. This result holds because we defined trace refinement in such a way that the refinement of a trace results in a trace from the refined system. To include traces that end in the middle of the refinement, we apply the prefix closure.

Lemma 1. $\downarrow(\mathit{Utraces}(s)[r]) = \mathit{Utraces}(s[r])$

Lemma 2 states that for completely refined *Utraces* the set of outputs after such a trace in the refined system equals the set of outputs in the abstract system after the contracted trace. This holds because r-complete traces end in states that come from the abstract system (old states). Because atomic linear input-inputs refinement does not add outputs to the refined system, the output behavior of the old states is not altered by the refinement.

Lemma 2. $\forall \sigma \in \mathit{Utraces}(s)[r] : \mathit{out}(s[r] \textbf{ after } \sigma) = \mathit{out}(s \textbf{ after } \sigma\langle r \rangle)$

For not completely refined *Utraces* (traces in $\downarrow(Utraces(s)[r])\backslash Utraces(s)[r]$) Lemma 3 states that the only output of the refined specification after such a trace is quiescence. This holds because not r-complete utraces end inside the refinement (in new states). Because our refinement does not add outputs, the only allowed output inside the refinement is quiescence.

Lemma 3. $\forall \sigma \in \downarrow(Utraces(s)[r])\backslash Utraces(s)[r] : out(s[r]\,\mathbf{after}\,\sigma) = \{\delta\}$

6 uioco$_r$ for testing refined systems

We introduce the implementation relation **uioco**$_r$ that express correctness of the concrete implementation in terms of the abstract specification and the refinement pair. We show that **uioco**$_r$ is equivalent to **uioco** for refined specifications.

Definition 9 (uioco$_r$). *Let* $s \in \mathcal{LTS}(I_1, U)$, $i \in \mathcal{IOTS}(I_2, U)$, $r = (a_r, \sigma_r)$, $I_2 = I_1\backslash\{a_r\} \cup I(\sigma_r)$.
i **uioco**$_r$ $s =_{\text{def}} \forall \sigma \in \downarrow(Utraces(s)[r]) :$
if $\sigma \in Utraces(s)[r]$ *then* $out(i\,\mathbf{after}\,\sigma) \subseteq out(s\,\mathbf{after}\,\sigma\langle r\rangle)$
else $out(i\,\mathbf{after}\,\sigma) \subseteq \{\delta\}$

For completely refined *Utraces* the allowed output behavior of the implementation is restricted to the output behavior of the abstract specification after the contracted trace (see Lemma 2). For not completely refined *Utraces* the allowed output behavior of the implementation is restricted to quiescence (see Lemma 3). Because of Lemma 1 we know that we have covered all possible traces of the refined specification. Theorem 1 states the equality between **uioco**$_r$ and **uioco**.

Theorem 1. *Let* $s \in \mathcal{LTS}(I_1, U), i \in \mathcal{IOTS}(I_2, U)$, $r = (a_r, \sigma_r)$, *and* $I_2 = I_1\backslash\{a_r\} \cup I(\sigma_r)$

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

Example 4. We look again at the abstract and refined specification in Figure 2, to illustrate Definition 9 and Theorem 1. We use the following two traces: street·city·postalcode·store is a complete refinement of address·store and street·city a not complete refinement. Both traces are in the set of *Utraces* of the refined specification, as stated in Lemma 1. The trace address·store leads to state q_2 in the abstract specification and the trace street·city·postalcode·store leads to state q_2 in the refined specification; the set of outputs is in both states the same, conform to Lemma 2. The not r-complete trace street·city leads to state $(t, 2)$ in the refined specification. This state is quiescent, as stated in Lemma 3. When we put these results together, it illustrates that the **uioco**$_r$ definition for the abstract specification is equal to the **uioco** definition for the refined specification.

7 Test case refinement

In the previous sections we have shown how to obtain a refined test suite by refining the specification; from this refined specification we can generate a complete

test suite. In this section we show how to refine existing abstract test cases, like the test cases shown in Figure 3. Furthermore, we show under what conditions the refinement of a complete abstract test suite results in a complete refined test suite with respect to $\mathbf{uioco}_r$.

To test inside the refinement we need several test cases (we can make several observations). Therefore we generate a set of mini test cases that test the entire behavior of the refined action. We replace transitions with the refinement label in the abstract test case with these mini test cases.

7.1 Generation of mini test cases

We present an algorithm to generate mini test cases that test the entire behavior inside the refinement. The algorithm is closely related to the test generation algorithm of Tretmans [4]. There are some minor differences:

1. The only pass state is at the end of a mini test case. A possible error can be anywhere within the refinement, so it is no use to stop testing before the end of the refinement.
2. There are no observations at the start and the end state of the mini test. Because atomic linear input-inputs refinement does not add or change output actions we use the observations of the abstract system in these states.

Definition 10 (Generation of mini tests). *$MT \subseteq \mathcal{TEST}(I(\sigma_r), U_\delta)$, a set of mini tests, is obtained from σ_r (with respect to an input label set I and and output label set U) in the following way. The stimulus and response step are executed in a non-deterministic manner. Let $n = |\sigma_r|$ and $1 \leq i < n$.*

Stimulus step $t_i := \sigma_r|_i; t_{i+1}$
Response step $t_i := \sigma_r|_i; (\Sigma\{x; \mathbf{fail} \mid x \in U\} \Box \delta; t_{i+1})$
Pass step $t_n := \sigma_r|_n; \mathbf{pass}$

The set of mini test is built with the process algebraic operators action prefix (;) and choice ($\Box$ and Σ) in the same style as Tretman's algorithm. For readers that are unfamiliar with this notation,formally we write this as follows:

Let t_i be test cases for $i = 1, 2$ and $\mu \in L_\delta$
; $(\mu; t_1) \xrightarrow{\mu} t_1$
$\Box$ if $t_1 \xrightarrow{\mu} t_1'$ then $t_1 \Box t_2 \xrightarrow{\mu} t_1'$ and $t_2 \Box t_1 \xrightarrow{\mu} t_1'$
Σ if $t_i \xrightarrow{\mu} t_i'$ for $i \in I$ then $\Sigma\{t_i \mid i \in I\} \xrightarrow{\mu} t_i'$

7.2 Test case refinement

Test case refinement is similar to LTS refinement. The main difference is that test case refinement results in a *set* of refined test cases, where LTS refinement results in one transition system. The definition is explained in Example 5.

Definition 11. *Given a test case $t = \langle Q_t, I_t, U_t, T_t, t_0, \mathbf{pass}_t, \mathbf{fail}_t \rangle$ and a refinement pair (a_r, σ_r) we define test case refinement as follows. Let MT be the set of mini tests generated with the algorithm from Definition 10. Let f be a*

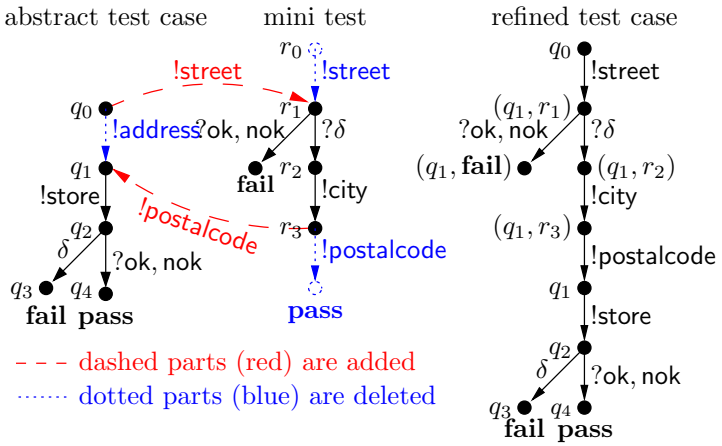

Fig. 4. Example of test case refinement

function from Q_t to MT. For better readability we denote a mini test obtained from f for a state q as $f(q) = \langle Q_q, I_q, U_q, T_q, start_q, \mathbf{pass}_q, \mathbf{fail}_q \rangle$. We assume all states to be unique.
$t[r] = \{t[f] \mid f : Q_t \to MT\}$ where $t[f] = \langle Q_f, I_f, U_t, T_f, t_0, \mathbf{pass}_t, \mathbf{fail}_f \rangle$ is defined as follows.

$$Q_f = Q_t \cup \{(q_2, q) \mid \exists q_1 \in Q_t : (q_1, a_r, q_2) \in T_t \wedge q \in Q_{q_2} \backslash (\mathbf{pass}_{q_2} \cup \{start_{q_2})\})\}$$

$$T_f = \{(q_1, \lambda, (q_2, q)) \mid (q_1, a_r, q_2) \in T_t \wedge (start_{q_2}, \lambda, q) \in T_{q_2}\}$$
$$\cup \{((q_2, q), \lambda, q_2) \mid \exists q_1 \in Q_t : (q_1, a_r, q_2) \in T_t \wedge \exists q_3 \in \mathbf{pass}_{q_2} : (q, \lambda, q_3) \in T_{q_2}\}$$
$$\cup T_t \backslash \{(q_1, a_r, q_2) \in T_t \mid q_1, q_2 \in Q_t\}$$

$$I_f = I_t \backslash \{a_r\} \cup I(\sigma_r)$$

$$\mathbf{pass}_f = \mathbf{pass}_t$$

$$\mathbf{fail}_f = \mathbf{fail}_t \cup \{(q_1, q_2) \in Q_f \mid q_1 \in Q_t \wedge q_2 \in \mathbf{fail}_{q_1}\}$$

We apply a little mathematical trick with our function f. The function maps the states of the abstract test case to the set of mini tests. For every refinement label transition (q_1, a_r, q_2) we get a mini test $f(q_2)$. We replace the refinement label transition with this mini test. $t[f]$ results in one refined test case and when we combine all possible refinements with f we get a set of refined test cases in which a_r transitions are replaced with all possible mini tests. We illustrate test case refinement in the following example.

Example 5. In Figure 4 we show an abstract test case on the left, a mini test in the middle and the resulting refined test case on the right. We use different types of lines: dashed parts are added, dotted parts are deleted and solid parts remain unchanged.

We delete the refinement label transition, $(q_0, \mathsf{address}, q_1)$ from the abstract test case (dotted transition) and all other transitions are added to T_f. All states are copied to Q_f.

From the mini test we delete the start and pass states. All other states are added to Q_f as a pair with q_1. We delete the transitions from the start state and transitions leading to pass states and add all other transitions to T_f.

To finalize the test case refinement we let the first transition in the mini test start in q_0, the start state of the refinement transition: the striped transition labeled with street between q_0 and r_1. In a similar way we redirect the postalcode transition to the pass state to q_1. When we reorganize the dashed parts and the black solid parts we obtain the refined test case on the right.

7.3 Completeness of test case refinement

The test suite derived from the refined specification is complete with respect to **uioco** and $s[r]$ and thus with respect to $\mathbf{uioco}_r$ and s. If we can show that the refinement of a complete test suite results in a complete refined test suite with respect to $\mathbf{uioco}_r$ and s, we know that both test suites are equivalent under completeness.

As usual we divide completeness in *soundness* and *exhaustiveness* [4]. A test case is sound when it does not end in a fail state when executed against a correct implementation. If every incorrect implementation is detected by the test suite, we call a test suite exhaustive.

Test case refinement is defined in such a way that the refinement of a sound test case with respect to **uioco** and s leads to a sound refined test case with respect to $\mathbf{uioco}_r$ and s.

Theorem 2 (Soundness of the refined test suite).
$(t$ is **sound** *w.r.t.* **uioco** *and* $s) \Rightarrow (t[r]$ is **sound** *w.r.t.* $\mathbf{uioco}_r$ *and* $s)$

Intuitively this theorem can be explained as follows. Like with LTS refinement we have the property that completely refined *Utraces* of s end in states of the abstract test case, where the output behavior is completely determined by the abstract system (see Lemma 2). Soundness is guaranteed by the soundness of the abstract test case. Not completely refined *Utraces* test the behavior of the refinement, where the output behavior is limited to quiescence (see Lemma 3). Not completely refined traces lead to states from the mini tests. It can be easily seen that mini tests generated with the algorithm in Definition 10 only lead to fail if the observed output is not quiescent.

It turns out that exhaustiveness of the refined test suite does not necessarily follow from exhaustiveness of the abstract test suite. When the abstract test suite fulfills the following property, exhaustiveness of the refined test suite holds.

Definition 12. *Let* $s \in \mathcal{LTS}(I, U)$ *and* $r = (a_r, \sigma_r)$

$$r\text{-}cov(T, s) =_{\text{def}} \forall(\sigma{\cdot}a_r) \in Utraces(s) : (\exists t \in T : t \xrightarrow{\sigma \cdot a_r})$$

The property states that a test suite T covers a specification s with respect to r if for every utrace of s ending in a_r, there is a test case in T that can perform this trace.

Theorem 3 (Exhaustiveness of the refined test suite). *Let* $s \in \mathcal{LTS}(I, U)$ *and* $r = (a_r, \sigma_r)$ *and let* $r\text{-}cov(T, s)$ *then*
(T is **exhaustive** *w.r.t.* **uioco** *and* s) $\Rightarrow$ ($T[r]$ is **exhaustive** *w.r.t.* $\mathbf{uioco}_r$ *and* s)

For exhaustiveness we follow the same line of thought as in the explanation of soundness. If the implementation is not $\mathbf{uioco}_r$ correct there can be an error in the abstract behavior (from the abstract specification) or in the behavior of the refinement. In case of an error in the abstract behavior, we know that there is a test case that reveals the failure because the abstract test suite is exhaustive.

In case of incorrectness in the refined part of the specification, we run into a problem. It may be that there is an error inside the refinement, but no abstract test case that leads to the refinement. The reason for this is that a complete test suite remains complete when deleting test cases that always lead to pass. The deleted test case may just be the test case that we need to obtain exhaustiveness. We can illustrate this as follows. Suppose that we have a specification that allows all behavior. A test suite with one test case that only consists of a **pass** state is complete. Refinement of this test suite results in the same test suite. Suppose that we have an implementation that can only perform the first refinement action and after that is not quiescent. This implementation is not $\mathbf{uioco}_r$ correct, but the refined test suite does not have a test case to detect this.

For r-cov test suites exhaustiveness holds, because there always is an abstract test case that leads us to the refinement. Within the refinement only quiescence is allowed as output and because the implementation is not $\mathbf{uioco}_r$ correct, we know that it is not quiescent. In the mini test generation algorithm we can easily see that such behavior leads to a fail verdict. We illustrate the soundness and exhaustiveness results with an example.

Example 6. Figure 5 shows an abstract test case (left), a refined test case and two implementations (right) for our data entry system. Both implementations have an error. Implementation 1 is quiescent in state i_3 and implementation 2 allows the output ok in state j_2.

For soundness we want to know if an error detected by a refined test case is indeed an error in the implementation. For implementation 1 we observe quiescence after street, city and postalcode. Our test case leads to fail because it expects ok or nok as observation. Because the fail state is a state from the abstract test case and because we know that the abstract test case is sound, we also know that our refined test case is sound.

For implementation 2, the execution of the refined test case leads to a fail verdict after observing ok after street followed by city. This is a failure within the refinement ($(q_2, \mathbf{fail})$ is a new state). Our observation within the refinement is ok and we know that the only allowed output within a refinement is δ. This means that the **fail** verdict is correct and that the test case is sound.

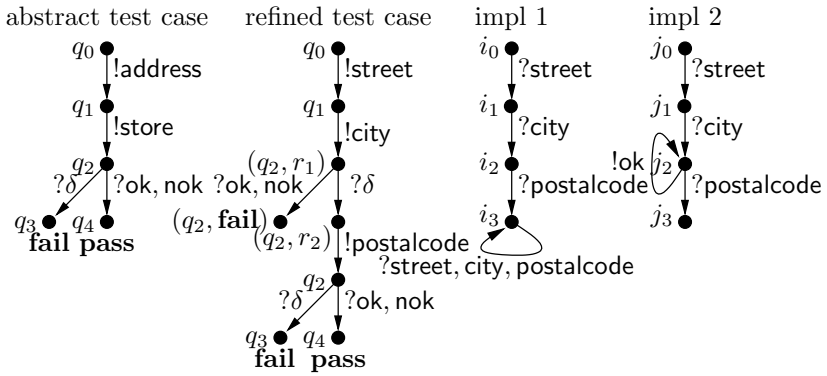

Fig. 5. Figure to illustrate soundness and completeness properties

For exhaustiveness we can follow the same line of thought. Suppose the implementation is not **uioco** correct, like implementations 1 and 2, do we have a test case that detects the error? For implementation 1 this is clear: the error is in the abstract part of the system and because the abstract test suite is complete, there is a test case that tests the specific abstract state of the specification. Because this abstract test case is present, we know that the refined test case will detect the error. For an error inside a refinement, like in implementation 2 we have a problem, because it requires that there is an abstract test case that starts with address. As explained earlier, the existence of such a test case is not guaranteed by completeness.

It may be unclear if the r-cov requirement for exhaustiveness can be met. The test case generation algorithm of Tretmans [4] fulfills this requirement (as it does not optimize test suites by deleting test cases).

Corollary 1. *The refinement of a complete test suite generated with Tretmans algorithm for test case generation, is complete with respect to* $\mathbf{uioco}_r$ *and the abstract specification.*

8 Conclusion

In this paper we have filled in the parts of our action refinement approach in Figure 1 for atomic linear input-inputs refinement. For this special case of action refinement we showed how to refine traces, transition systems and test cases. This enables us to obtain test cases with the required level of detail in an automated way. Furthermore we introduced the implementation relation $\mathbf{uioco}_r$ that relates the abstract specification to the concrete implementation by using the refinement information in the form of the refinement pair. We showed that a complete test

suite can be derived from the refined specification and under which conditions this test suite is equivalent to the refinement of a complete abstract test suite.

Related work In the light of conformance testing, the problem addressed by this paper is well known in practice. However, no research has been carried out in the field of conformance testing nor in the field of action refinement.

In the context of action refinement, the results of Section 7 have an unexpected consequence. The vast majority of research in action refinement has concentrated on the so-called *coarsest congruence* question (given two equivalent specifications, are they still equivalent after refinement?). In this paper we are not primarily interested in equivalences at all: the core issue is the conformance relation, embodied in **uioco**. Still, an obvious derived equivalence is that of *specification strength* — two specifications are equivalent if they are satisfied by the same set of systems. Surprisingly, this equivalence is *not* preserved even under atomic action refinement, as a side-effect of the fact that test case refinement does not always preserve completeness. This is in contrast to previously studied equivalences; see [2].

Future work This paper is only a first step; it treats a non-trivial though rather simple form of atomic action refinement. Future research focuses on arbitrary atomic refinement. This means that no actions are allowed to interfere with 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. Arbitrary atomic refinement is the next research step.

Some research has been done in comparing Finite State Machine (FSM) testing with LTS based testing [3]. With atomic action refinement we can refine the atomic input output pair from an FSM into two sequential actions. This might give an interesting basis for comparison.

References

1. G. Bernot, M. G. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, (November), 1991.
2. R. Gorrieri and A. Rensink. Action refinement. In *Handbook of Process Algebra*, chapter 16, pages 1047–1147. Elsevier, 2001.
3. A. Petrenko, G. v. Bochmann, and R. Dssouli. Conformance relations and test derivation. In *IWPTSVI*, pages 157–178. North-Holland, 1994.
4. J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools*, 17(3):103–120, 1996.
5. M. van der Bijl, A. Rensink, and J. Tretmans. Action refinement roadmap. Technical report, University of Twente, 2004. URL: http://www.cs.utwente.nl/~vdbijl/papers.
6. M. van der Bijl, A. Rensink, and J. Tretmans. Compositional testing with ioco. In *FATES 2003*, volume 2931 of *LNCS*, pages 86–100. Springer, 2004.
7. M. van der Bijl, A. Rensink, and J. Tretmans. Action refinement in conformance testing. Technical report, University of Twente, 2005. URL: http://www.cs.utwente.nl/~vdbijl/papers.