

Comparing Refinements for Failure and Bisimulation Semantics

Rik Eshuis and Maarten M. Fokkinga

University of Twente, Dept INF
P.O. Box 217, NL 7500 AE Enschede, the Netherlands
{eshuis,fokkinga}@cs.utwente.nl

Abstract. Refinement in bisimulation semantics is defined differently from refinement in failure semantics: in bisimulation semantics refinement is based on simulations between labelled transition systems, whereas in failure semantics refinement is based on inclusions between decorated traces systems. There exist however pairs of refinements, for bisimulation and failure semantics respectively, that have almost the same properties. Furthermore, each refinement in bisimulation semantics implies its counterpart in failure semantics, and conversely each refinement in failure semantics implies its counterpart in bisimulation semantics defined on the canonical form of the compared processes.

Keywords: refinement, decorated traces, labelled transition systems, bisimulation semantics, failure semantics.

1 Introduction

Motivation. Refinement is a transformation of a design from a high level, abstract form to a lower level, more concrete form. The high level design is called the specification, the low level design the implementation. The main usefulness of refinement is that a complex task of implementing a system that satisfies a given specification is made easier by gradually refining the abstract specification until finally a (concrete) implementation is obtained.

Two kinds of transformation are particularly useful. One of them is the reduction of nondeterminism, so that the implementation deadlocks less often. The other kind of transformation is the extension of the functionality of the design, by adding new actions to the system without increasing nondeterminism. Such an extension is necessary during the implementation phase, when (by intention) the high level abstract design does not completely fix the ultimately required functionality. (In this case we call the abstract design a *partial* specification.)

Both kinds of transformation lead to a single notion of substitutability. A specification is refined by an implementation if usage of the system according to the specification is still valid – and has the same observable results – if actually the system is built according to the implementation.

Formalisation. In the sequel, we no longer use the word ‘refinement’ to denote the action of refining, as above, but instead use it to denote a mathematical relation between the (abstract and concrete) designs.

In order to formally express refinement, we first need to formally express a design. We take a design to be a *process*. There exist, however, different semantics for processes, the most common of which are those of CCS [Mil89] and of CSP [Hoa85]. In CCS, bisimulation semantics is used: a process is a *labelled transition system (LTS)*, and processes are identified if they are bisimilar to each other. Consequently, refinement between processes is defined as a kind of simulation. In CSP, failure semantics is used: a process is a *decorated traces system (DTS)*, consisting of traces “decorated” with refusals, and processes are identified if their traces and refusals are the same. Consequently, refinement between processes is defined as inclusions between their traces and refusals. In our comparison of refinements in the two kinds of semantics, we restrict ourselves to *finitely branching, concrete, sequential* processes.

Results. Although refinements on LTSs and DTSs are defined quite differently, they are strongly related. In particular, there exist pairs of LTS and DTS refinements that have (almost) the same properties. Every LTS refinement implies its DTS counterpart. Conversely, and that is what we consider the main result of this paper: every DTS refinement implies —and is equivalent to— its LTS counterpart on the so-called *canonical form* of the processes. In order to substantiate this claim, we had to invent a new LTS refinement and a new DTS refinement, which, after all, turned out to be compositions of well-known refinements.

To discuss the result in more detail, and to see what is new, consider the classification of refinements in figure 1:

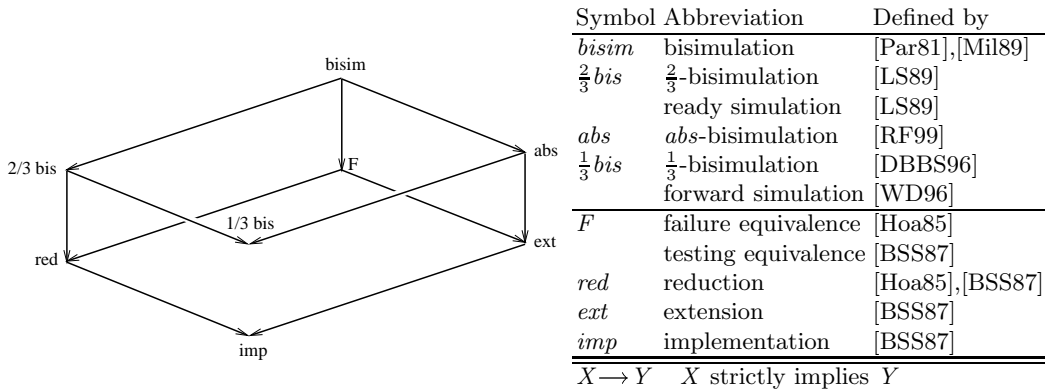


Fig. 1. Some refinements

Refinement *abs* has not yet been studied in the literature. The well-known items in the figure have already been related to each other by the ‘strictly implies’ relation \rightarrow by other people. Van Glabbeek [Gla90] classified the left back face:

$bisim, \frac{2}{3}bis, F, red$. Brinksma *et al.* [BSS87] classified the bottom face, consisting of the refinements for failure semantics: F, red, ext, imp . Finally, Bowman *et al.* [DBBS96] classified the top face excluding abs , consisting of the refinements for bisimulation semantics: $bisim, \frac{2}{3}bis, \frac{1}{3}bis$. To begin with, we add the relationships for abs .

Note that there is a missing edge between $\frac{1}{3}bis$ and imp : neither of them implies the other one. One result of our investigation is the construction of two refinements, being the counterparts of $\frac{1}{3}bis$ and imp respectively. These turn out to be compositions of other refinements in the cube.

The second result is the theorem that each refinement for failure semantics (in the bottom face) also implies, and is equivalent to, its counterpart for bisimulation semantics (the one just above it in the top face) when *applied to the canonical form of the processes*. The construction of a canonical process is due to He Jifeng [He89]. The significance of this result is that every refinement for failure semantics can be proven by proving its counterpart for bisimulation semantics. This means, no traces and refusals have to be computed, only the canonical LTSs of the compared DTSs, which is hopefully a less costly operation. Furthermore, this result makes tools for labelled transition systems applicable for decorated traces systems.

As a minor result of our investigation we would like to mention the classification of what properties hold for what refinement. The properties that we have considered are crucial for practical application of refinement: being a pre-order or even partial order, being compositional (that is, suitable for being applied to *parts* of a process expression), and being safe (w.r.t. traces and refusal). All our results are summarised in Figure 2.

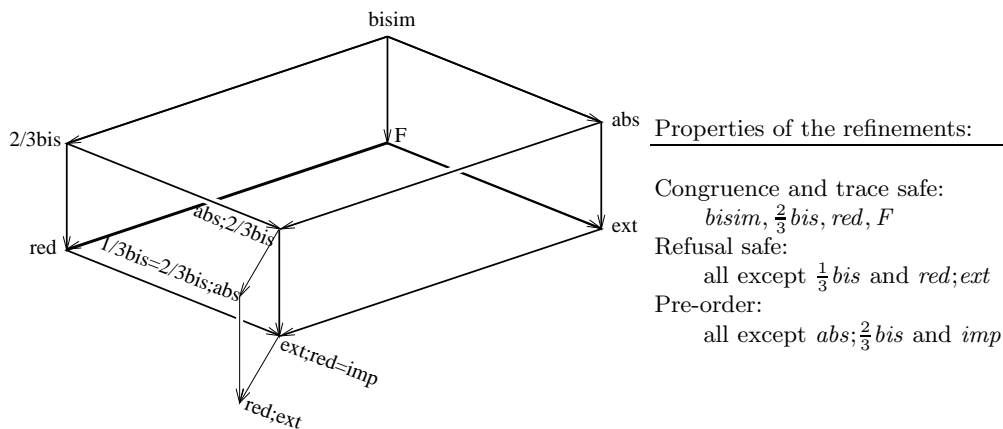


Fig. 2. Classification of refinements

Structure of the paper. The remainder of this paper is organised as follows. In section 2 we present the LTS and DTS notion of process and a transformation

from one to the other. Section 3 firstly introduces informally the notion of refinement, and presents several properties of interest, and then continues to formally define the refinements for DTSs and LTSs and to present their properties. In section 4 we classify the refinements and discuss how LTS refinements relate to their DTS counterpart. The paper ends with a short section on related work.

The proofs are obtainable by ftp [EF99].

2 Processes

In this section, we give two definitions of the notion of process: the LTS variant (a ‘labelled transition system’), as used in CCS, and the DTS variant (also known as ‘decorated traces system’), as used in CSP.

Labelled transition systems. In CCS, a process is a labelled transition system. A *labelled transition system* (LTS) is a quadruple $(Act, State, \rightarrow, start)$ where:

- Act is a set of so-called observable, external actions,
- $State$ is a set of so-called states,
- $(\rightarrow) \subseteq State \times Act \times State$ is a so-called transition relation,
- $start \subseteq State$ is a nonempty set of so-called start states.

One of the start states is chosen as initial state. The definition of an LTS is a generalisation of the original one [Mil89], in which there is only one start state. Below, we will see that, nondeterministically, each start state will play the role of the initial state.

For a process P we denote the components of P by $State_P$, Act_P , \rightarrow_P , and $start_P$ respectively. We drop the subscripts if no confusion can arise. We let s range over $State$ and a over Act .

For an LTS P there are several derived notions, which we now define. (Each of them depends on P , but we omit the subscript here.) First, some notational conventions:

$$\begin{aligned} s \xrightarrow{a} s' &\Leftrightarrow (s, a, s') \in (\rightarrow) \\ s \xrightarrow{a} &\Leftrightarrow \exists s' \bullet s \xrightarrow{a} s' \\ s \not\xrightarrow{a} &\Leftrightarrow \nexists s' \bullet s \xrightarrow{a} s' \end{aligned}$$

In the latter case, we say P *refuses* action a in state s . The *initials* of P at state s is the set of actions that can be done in state s , and the process P *deadlocks* in state s if it does not accept any action in that state:

$$\begin{aligned} initials(s) &= \{a \in Act \mid s \xrightarrow{a}\} \\ deadlock(s) &\Leftrightarrow initials(s) = \emptyset \end{aligned}$$

So far, we have only been concerned with describing the internal structure of processes by giving states and transitions between these states. Every process

however also has some external structure, in which the states of the process cannot be observed, only its actions. Observations of actions gives rise to traces. A *trace* is a sequence $t \in Act^*$; we denote the empty trace by ϵ and trace concatenation by juxtaposition. Relation \rightarrow is extended to traces; for $t = a_1 a_2 \dots a_n$:

$$s \xrightarrow{t} s' \Leftrightarrow s \xrightarrow{a_1} \xrightarrow{a_2} \dots \xrightarrow{a_n} s'$$

The set of *traces* of process P is defined as follows:

$$Tr_P = \{t \in Act^* \mid \exists s \in State \bullet start \ni \xrightarrow{t} s\}$$

where $\ni \xrightarrow{t}$ is the relational composition of \ni and \xrightarrow{t} , that is, $start \ni \xrightarrow{t} s$ is shorthand for $\exists s_0 \bullet s_0 \in start \wedge s_0 \xrightarrow{t} s$.

Furthermore, observations not only comprise actions a system can do, but also actions a system cannot do. This latter aspect is captured by the notion of a refusal set. The *refusal* of a trace $t \in Tr$ is a set of actions such that all actions in the set can be refused after the process has executed t . The *refusal set* $Ref(t)$ belonging to a trace $t \in Tr$ consists of all refusals of t :

$$R \in Ref(t) \Leftrightarrow \exists s \bullet start \ni \xrightarrow{t} s \wedge \forall a \in R \bullet s \not\xrightarrow{a}$$

Note that Ref is closed under inclusion: $X \subseteq R \in Ref(t)$ implies $X \in Ref(t)$.

Another important definition is that of determinism. Process P is *deterministic* if for all traces t the same actions can be done in all possible states that P may end up in after having executed t :

$$\begin{aligned} det_P \Leftrightarrow & \forall s, s' \in State; t \in Tr \bullet start \ni \xrightarrow{t} s \wedge \\ & start \ni \xrightarrow{t} s' \Rightarrow initials(s) = initials(s') \end{aligned}$$

Process P is *nondeterministic* if it is not deterministic. In particular, if s_0, s_1 are in $start$, and $initials(s_0) \neq initials(s_1)$, then the process is nondeterministic.

Decorated traces systems. A decorated traces system (DTS) may be viewed as an abstraction of an LTS, in which only the external properties are kept; it is the triple: (Act, Tr, Ref) . Function Ref “decorates” each trace t with its refusal set $Ref(t)$; hence the name ‘decorated traces’. Any such triple (Act, Tr, Ref) is a DTS, provided that Tr is nonempty and prefix-closed, Ref is nonempty and closed under inclusion, and actions leading outside Tr , must be refused:

$$\begin{array}{ll} \epsilon \in Tr & \text{[non-emptiness } Tr\text{]} \\ tt' \in Tr \Rightarrow t \in Tr & \text{[prefix closedness } Tr\text{]} \\ t \in Tr \Rightarrow \emptyset \in Ref(t) & \text{[non-emptiness } Ref\text{]} \\ t \in Tr \wedge R \in Ref(t) \wedge R' \subseteq R \Rightarrow R' \in Ref(t) & \text{[subset closedness } Ref\text{]} \\ t \in Tr \wedge ta \notin Tr \wedge R \in Ref(t) \Rightarrow R \cup \{a\} \in Ref(t) & \text{[consistency } Ref\text{]} \end{array}$$

For DTSs some notions can be derived that are complementary to the ones for LTSs. The *initials* of P at trace t is the set of actions that can be done after t ,

and the process P necessarily deadlocks after trace t if it cannot do any action after that trace:

$$\begin{aligned} \text{initials}(t) &= \{a \mid ta \in Tr\} \\ \text{deadlock}(t) &\Leftrightarrow \text{initials}(t) = \emptyset \end{aligned}$$

A process P is *deterministic* if after every trace t , every action that is possible (according to Tr) is never refused:

$$\text{det}_P \Leftrightarrow \forall t \in Tr; R \in \text{Ref}(t) \bullet R \cap \text{initials}(t) = \emptyset$$

A process P is *nondeterministic* if it is not deterministic.

From LTS to DTS. For an arbitrary LTS $P = (Act, State, \rightarrow, start)$, we have defined its traces Tr_P and refusals Ref_P . Both Tr_P and Ref_P satisfy the constraints for DTSs, mentioned above. Hence, P determines a DTS, denoted $dts(P) = (Act, Tr_P, Ref_P)$. Now the question arises whether the notions of initials, deadlock and determinism for $dts(P)$ are consistent with those for P . The following theorem asserts that this is the case.

Proposition 1. Let $P = (Act, State, \rightarrow, start)$ be an LTS, and $Q = dts(P)$. Then:

1. $(\cup_{s \in State \mid start \xrightarrow{t} s} \text{initials}_P(s)) = \text{initials}_Q(t)$
2. $(\forall_{s \in State \mid start \xrightarrow{t} s} \text{deadlock}_P(s)) \Leftrightarrow \text{deadlock}_Q(t)$
3. $\text{det}_P \Leftrightarrow \text{det}_Q$

From DTS to LTS. In the previous paragraph, we saw that every LTS P can be transformed into a DTS $dts(P)$. Conversely, a DTS $P = (Act, Tr, Ref)$ is interpreted as an LTS $lts(P)$, called the *canonical form* of P , as follows. The states of $lts(P)$ are pairs (t, R) , where t is a trace of P and R is in the refusal set $Ref(t)$. From such a state (t, R) , an a -transition is possible to (ta, R') if and only if a is not refused, that is $a \notin R$, and (ta, R') is again a state of $lts(P)$. So formally, $lts(P) = (Act, State, \rightarrow, start)$ where:

$$\begin{aligned} State &= \{(t, R) \mid t \in Tr \wedge R \in Ref(t)\} \\ (t, R) \xrightarrow{a} (t', R') &\Leftrightarrow (t, R) \in State \wedge a \notin R \wedge ta = t' \wedge (t', R') \in State \\ start &= \{(\epsilon, R) \mid R \in Ref(\epsilon)\} \end{aligned}$$

We now prove that $lts(P)$ has the same traces and refusals as P .

Proposition 2. Let $P = (Act, Tr, Ref)$ be a DTS, and $Q = dts(lts(P))$. Then Q is the same as P : both have the same set of observable actions, traces and refusals:

$$\begin{aligned} Act &= Act_Q \\ t \in Tr &\Leftrightarrow t \in Tr_Q \\ R \in Ref(t) &\Leftrightarrow R \in Ref_Q(t) \end{aligned}$$

		$P \xrightarrow{a} P'$	$Q \xrightarrow{a} Q'$
$a \xrightarrow{a} \mathbf{stop}$	$a.P \xrightarrow{a} P$	$P + Q \xrightarrow{a} P'$	$P + Q \xrightarrow{a} Q'$
$P \xrightarrow{a} P' \quad a \notin A$	$Q \xrightarrow{a} Q' \quad a \notin A$	$P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q' \quad a \in A$	
$P \parallel_A Q \xrightarrow{a} P' \parallel_A Q$	$P \parallel_A Q \xrightarrow{a} P \parallel_A Q'$	$P \parallel_A Q \xrightarrow{a} P' \parallel_A Q'$	

Table 1. Structural operational semantics for the process notation

Again, now the question arises if the definitions of (non)determinism are consistent with each other, that is, if a deterministic DTS P is transformed into an LTS $lts(P)$, is it still deterministic?

Proposition 3. Let $P = (Act, Tr, Ref)$ and $Q = lts(P)$. Then $det_P \Leftrightarrow det_Q$.

Process notation. In order to denote (many, but certainly not all) finite LTSs and DTSs, we introduce a fairly standard syntax. The notation is not a full-fledged process language; it is only meant to be used in (counter)examples and in congruence claims. Let Act be the set of external actions. Single action a is an elementary process term. We consider only sequencing (\cdot), choice ($+$) and interleaving with communication (\parallel_A) as operators to compose processes. The following grammar describes these process terms:

$$P ::= a \mid a.P \mid P + P \mid P \parallel_A P$$

where $a \in Act$ and $A \subseteq Act$.

Each process term P gives rise to a labelled transition system. The states are the sub-terms of P plus an additional term **stop**, with term P itself as only start state. The transition relation is the least relation satisfying the implications, written as inference rules, in table 1. Since each LTS determines a DTS, a term also denotes a DTS. In the sequel, we shall blur the distinction between processes and process terms: we shall use a process term P for the LTS/DTS process denoted by P .

3 Refinement

As explained in the introduction, one process S (specification) is refined by another process I (implementation) if the environment of S does not note any difference if the machine described by S is replaced by the machine described by I . We then write $S \sqsubseteq I$ (conform the Z notation [Spi92]). Various alternative refinements will be distinguished by subscripts: \sqsubseteq_{ext} , \sqsubseteq_{abs} , etc.

Before we define various refinements \sqsubseteq , let us first mention the properties we are interested in. These are: partial orders and congruence, safety and liveness.

Partial order and congruence. In order to *gradually* refine a specification into a final implementation, reflexivity and transitivity are important, collectively known as pre-order. If in addition anti-symmetry or symmetry holds, the relation is a partial order or equivalence, respectively. When a refinement \sqsubseteq_x is an equivalence, we write $=_x$ instead.

If a refinement is a pre-order that can be used in context, we speak of *compositionality*; one also says that the refinement is a *congruence* for the operations of the language. Formally, for arbitrary process term P , and arbitrary operation \oplus of the language:

$$S \sqsubseteq I \Rightarrow P \oplus S \sqsubseteq P \oplus I \quad [\text{congruence/compositionality}]$$

Strictly speaking, with this property \sqsubseteq is only a pre-congruence; it is a congruence if, in addition, \sqsubseteq is an equivalence.

Safety and liveness. *Safety* is informally defined as “something bad will not happen”. The safety property for refinement states that the implementation should preserve the safety of the specification. There are two different interpretations of “bad”. The first one is to interpret “bad” as something unspecified. Hence, in terms of traces, the safety property states that some “bad” trace that is not accepted by the specification, is not accepted by the implementation:

$$Tr_I \subseteq Tr_S \quad [\text{trace safety}]$$

Another interpretation of “bad” is deadlock. If the implementation deadlocks after trace t by refusing a certain action, whereas the specification does not, the implementation is not safe. We formalise this interpretation as follows:

$$\forall t \in Tr_S \cap Tr_I \bullet Ref_I(t) \subseteq Ref_S(t) \quad [\text{refusal safety}]$$

We do not make a choice between these two views; instead we focus on both of them.

Liveness is informally defined as “something good happens eventually”. The implementation should not execute an infinite number of (internal) actions without making progress. In other words, the implementation should not diverge, if the specification does not. Since we do not consider internal actions, divergence will not happen. Therefore we do not consider this property in the sequel.

Now we come to the formal definitions. First we study refinements for DTSs, and next refinements for LTSs.

3.1 DTS refinements

For DTSs, each refinement \sqsubseteq is defined in terms of inclusions between traces of S and I . However traces do not discern two processes that have different deadlock behaviour. Therefore we require an additional inclusion between the refusal sets of I and S , assuring that for common traces process I deadlocks less often than S :

$$\forall t : Tr_S \cap Tr_I \bullet Ref_S(t) \subseteq Ref_I(t)$$

(We consistently write $Tr_S \cap Tr_I$ even though one of them may include the other.) Furthermore, the assumption is made that both S and I have the *same* observable actions ($Act_S = Act_I$), otherwise the refusals of S and I would be incomparable.

Failure equivalence. Failure equivalence, $=_F$, was first introduced by Hoare [Hoa85]. This equivalence plays the role of identity in CSP. We give a different formulation here, first given by Brinksma *et al.* [BSS87] under the name *testing equivalence* for LOTOS.

Failure equivalence is defined as follows:

$$S =_F I \Leftrightarrow Tr_S = Tr_I \wedge \forall t \in Tr_S \cap Tr_I \bullet Ref_S(t) = Ref_I(t)$$

Two processes are equal if they have the same traces, and furthermore, they refuse the same actions after every trace.

In CSP, a failure is a pair (t, R) , where $R \in Ref(t)$. In the original definition of failure equivalence, this equivalence was defined in terms of these pairs, hence the name failure equivalence. Properties of this relation are summarised in table 2. When used as a refinement, the most distinguishing property of $=_F$ is that

Property	$=_F$	\sqsubseteq_{red}	\sqsubseteq_{ext}	\sqsubseteq_{imp}	$=_{bis}$	$\sqsubseteq_{\frac{2}{3}bis}$	\sqsubseteq_{abs}	$\sqsubseteq_{\frac{1}{3}bis}$	$\sqsubseteq_{red-ext}$	$\sqsubseteq_{abs-\frac{2}{3}bis}$
Reflexivity	x	x	x	x	x	x	x	x	x	x
Transitivity	x	x	x		x	x	x	x	x	
Symmetry	x			x						
Anti-symmetry		x	x							
Congruence	x	x			x	x				
Refusal safety	x	x	x	x	x	x	x			x
Trace safety	x	x			x	x				

Table 2. Properties of refinements for DTSs and LTSs

it is symmetric. That means that specification and implementation cannot be distinguished: they are all equal (i.e., express the same observable behaviour).

Example. Let $S = a.b.c + a.b.d$ and $I = a.(b.c + b.d)$. Now $S =_F I$.

Reduction. Reduction, \sqsubseteq_{red} , was first defined for use within CSP [Hoa85]. This is the only refinement that is used in CSP. As we will see, however, other useful definitions of refinement exist for CSP. Reduction is also used in LOTOS [BSS87], of which we use the formulation.

Reduction is defined as follows:

$$Tr_S \sqsubseteq_{red} Tr_I \Leftrightarrow Tr_I \subseteq Tr_S \wedge \forall t \in Tr_S \cap Tr_I \bullet Ref_I(t) \subseteq Ref_S(t)$$

The implementation cannot exhibit more behaviour than the specification. Furthermore, the implementation cannot refuse more than the specification. It can

however refuse less. Therefore, the implementation is more predictable than the specification, and thus better.

Properties of this refinement are shown in table 2. Reduction can be used to reduce nondeterminism in the specification. The implementation does not have to contain all traces of specification: the traces that contain an action that sometimes can be refused may be left out. Reduction can only be used for complete specifications: partial specifications cannot be completed by refinement, since no new traces can be added.

Example. Let $S = a.b.c + a.b.d$ and $I = a.b.c$. Now $S \sqsubseteq_{red} I$. Note that $Ref_I(ab) \subset Ref_S(ab)$.

Extension. Extension, \sqsubseteq_{ext} , was first defined by Brinksma *et al.* [BSS87] for use in LOTOS. It is the complement of reduction. Extension was defined because when the specification is a partial specification, the implementation must extend the specification to comply with other constraints, not mentioned in the partial specification.

Extension is defined as follows:

$$S \sqsubseteq_{ext} I \Leftrightarrow Tr_S \subseteq Tr_I \wedge \forall t \in Tr_S \cap Tr_I \bullet Ref_I(t) \subseteq Ref_S(t)$$

The implementation can exhibit more behaviour than the specification. The implementation deadlocks less often than the specification for the traces that implementation and specification have in common, since the implementation cannot refuse more than the specification for those traces.

Properties of this refinement are summarised in table 2. The most striking property of this refinement is that extension is not a congruence. Extension is only a congruence if $Tr_S = Tr_I$, according to Brinksma *et al.* [BSS87]. If $Tr_S \subset Tr_I$ the following counterexample holds. Let $P = a.b$, $S = c$, and $I = a + c$. Now $S \sqsubseteq_{ext} I$, but $P + S \not\sqsubseteq_{ext} P + I$, because after executing trace a , process $P + I$ can refuse all actions, thus it deadlocks, whereas $P + S$ always accepts b after executing a , so here no deadlock occurs.

Furthermore, this refinement is not safe with respect to trace safety, because it is possible to introduce new traces in the implementation (necessarily because the intended use is that S is viewed as a partial specification that must be completed eventually).

Implementation. Implementation, \sqsubseteq_{imp} , was first used in LOTOS [BSS87]. This refinement is also known under the name *conformance*.

Implementation is defined as follows:

$$S \sqsubseteq_{imp} I \Leftrightarrow \forall t \in Tr_S \cap Tr_I \bullet Ref_I(t) \subseteq Ref_S(t)$$

This refinement only states that the implementation cannot refuse actions that the specification accepts, for the traces that they have in common. It may seem strange at first sight that no subset/superset constraint between Tr_S and Tr_I is given. This is because the intended use is that, at one hand, new traces can

be added, just as with extension. On the other hand, nondeterminism can be reduced, just as with reduction, so traces containing actions that can be refused, may be removed from the specification.

Properties of this refinement are shown in table 2. Most striking is that it is not transitive. The following example illustrates this. Let $S = a.b + a.c.d$, $T = a.b$ and $I = a.(b + c)$. Now $S \sqsubseteq_{imp} T \sqsubseteq_{imp} I$ but $S \not\sqsubseteq_{imp} I$, since after trace ac , I refuses always action d whereas S always accepts this action. The fact that this refinement is not a pre-order makes this relation not applicable in the design phase. It is therefore surprising that nevertheless this refinement is used in a design language like LOTOS.

3.2 LTS refinements

Throughout the remainder of the paper, A and C (from ‘Abstract’ and ‘Concrete’) denote arbitrary states of S and I , respectively.

For LTSs, each refinement \sqsubseteq is defined in terms of relations R between states of S and I . For example, states A and C are related by R if all actions that can be done in A can also be done in C (so the implementation does not deadlock if the specification does not), and for any initial actions a of both A and C , the successor states of C if a is executed, are again related with (some or all of) the successor states of A . We then say that state C *simulates* state A and call relation R a *simulation witness*. More precisely, the witness is called a *forward simulation* since every move forwards of the implementation has to be matched by the specification. The counterpart of forward simulation is *backward simulation*: then ‘ C simulates A ’ implies that the predecessors of C simulate those of A . An obvious requirement for any forward simulation witness R is that every start state of I should simulate a start state of S : relation R should be total on $start_I$, that is, at least all states in $start_I$ occur in R .

Bisimulation. Bisimulation, $=_{bis}$, is one of the strongest possible equivalences for processes. It was first defined by Park [Par81], as an improvement on the equivalences used until then by Milner for CCS. This equivalence now forms the cornerstone of CCS [Mil89]. It defines identity on CCS processes.

To define *bisimulation*, we need the auxiliary notion of a bisimulation witness. A *bisimulation witness* between S and I is a relation R on the states of the labelled transition systems, satisfying the following property:

Whenever $A \underline{R} C$ then:

1. $A \xrightarrow{a} A' \Rightarrow \exists C' \bullet C \xrightarrow{a} C' \wedge A' \underline{R} C'$
2. $C \xrightarrow{a} C' \Rightarrow \exists A' \bullet A \xrightarrow{a} A' \wedge A' \underline{R} C'$

Now we define: $S =_{bis} I$ if there exists a bisimulation witness that is total on both $start_S$ and $start_I$.

Properties of this relation are shown in table 2. Like failure equivalence, this refinement has as most striking property that it is symmetric.

Example. (1) Let $S = a.b.c + a.b.d$ and $I = a.(b.c + b.d)$; then $S \neq_{bis} I$. **(2)** Let $S = a.b$ and $I = a.b + a.b$; then $S =_{bis} I$.

$\frac{2}{3}$ -bisimulation. Refinement $\frac{2}{3}$ -bisimulation, $\sqsubseteq_{\frac{2}{3}bis}$, was introduced by Larsen and Skou [LS89]. It is also known as *ready simulation*. Independently, He Jifeng introduced an equivalent definition [He89].

To define $\frac{2}{3}$ -bisimulation, we need the auxiliary notion of a $\frac{2}{3}$ -bisimulation witness. A $\frac{2}{3}$ -bisimulation witness between S and I is a relation R on the states of the labelled transition systems, satisfying:

Whenever $A \underline{R} C$ then:

1. $A \xrightarrow{a} C \xrightarrow{a}$
2. $C \xrightarrow{a} C' \Rightarrow \exists A' \bullet A \xrightarrow{a} A' \wedge A' \underline{R} C'$

Now we define: $S \sqsubseteq_{\frac{2}{3}bis} I$ if there exists a $\frac{2}{3}$ -bisimulation witness that is total on $start_I$.

Properties of this refinement are summarised in table 2. As reduction, this relation makes it possible to reduce nondeterminism. It is however not possible to extend a specification (in case of partial specifications). This refinement is a congruence. It is not a partial order, because it is not anti-symmetric. The following counterexample (from [Gla90]) shows that. Let $S = a.b.c + a.(b.c + b.d)$ and let $I = a.(b.c + b.d)$. Now $S \sqsubseteq_{\frac{2}{3}bis} I \sqsubseteq_{\frac{2}{3}bis} S$, but $S \neq_{bis} I$.

Abs-bisimulation. Abs-bisimulation, \sqsubseteq_{abs} , was conceived by Van Rein and Fokkinga for use in Paul [RF99], a language to express behavioral aspects of object-oriented designs. This refinement has never been published, since failure semantics seemed more appropriate for Paul.

Before we define *abs-bisimulation*, we need the auxiliary notion of an abs-bisimulation witness. An *abs-bisimulation witness* between S and I is a relation R on the states of the labelled transition systems, satisfying:

Whenever $A \underline{R} C$ then:

1. $A \xrightarrow{a} A' \Rightarrow \exists C' \bullet C \xrightarrow{a} C' \wedge A' \underline{R} C'$
2. $A \xrightarrow{a} \wedge C \xrightarrow{a} C' \Rightarrow \exists A' \bullet A \xrightarrow{a} A' \wedge A' \underline{R} C'$

We define: $S \sqsubseteq_{abs} I$ if there exists an abs-bisimulation witness that is total on $start_S$ and $start_I$.

Properties of this refinement are summarised in table 2. Nondeterminism cannot be reduced by the implementation: the implementation is as nondeterministic as the specification. It is however possible to extend the specification by adding a ‘new’ action a to a state C , where $a \notin initials(A)$. This way, new traces can be added to S .

This refinement is not a congruence and not a partial order. The following counterexample ([SF98]) shows that this relation is not anti-symmetric. Let $S = a.(a + a.b.c)$ and $I = a.(a + a.b + a.b.c)$. Now $S \sqsubseteq_{abs} I \sqsubseteq_{abs} S$ but $S \neq_{bis} I$.

$\frac{1}{3}$ -bisimulation, Z forward simulation. Refinement $\frac{1}{3}$ -bisimulation, $\sqsubseteq_{\frac{1}{3}bis}$, is used in Z under the name forward simulation [WD96]. Translation of Z refinement to processes gives rise to $\frac{1}{3}$ -bisimulation. This latter relation is defined in [DBBS96] and is an adaptation of the $\frac{2}{3}$ -bisimulation (ready simulation) that we discussed above.

Before we define $\frac{1}{3}$ -bisimulation, we need the auxiliary notion of a $\frac{1}{3}$ -bisimulation witness. A $\frac{1}{3}$ -bisimulation witness between S and I is a relation R on the states of the labelled transition systems, satisfying:

Whenever $A \underline{R} C$ then:

1. $A \xrightarrow{a} \Rightarrow C \xrightarrow{a}$
2. $A \xrightarrow{a} \wedge C \xrightarrow{a} C' \Rightarrow \exists A' \bullet A \xrightarrow{a} A' \wedge A' \underline{R} C'$.

We define: $S \sqsubseteq_{\frac{1}{3}bis} I$ if there exists a $\frac{1}{3}$ -bisimulation witness that is total on $start_I$.

Properties of this refinement are summarised in table 2. Remarkable is that refusal safety does not hold. The following counterexample shows this: take $S = a.b + a.c.d$ and $I = a.(b + c.e)$. Clearly, $S \sqsubseteq_{\frac{1}{3}bis} I$. But I is not safe with respect to refusals, because after trace ac process I always refuses d whereas S always accepts d .

Nondeterminism may be reduced by the implementation, just as with $\frac{1}{3}$ -bisimulation, and ‘new’ actions (and therefore new traces) can be added, just as with abs-bisimulation.

This refinement is not a partial order. For anti-symmetry, the following is a counterexample ([SF98]). Let $S = a.(a + a.b.c)$ and $I = a.(a + a.b + a.b.c)$. Now $S \sqsubseteq_{\frac{1}{3}bis} I \sqsubseteq_{\frac{1}{3}bis} S$ but $S \neq_{bis} I$.

Just like abs-bisimulation, $\frac{1}{3}$ -bisimulation is not a congruence as shown by the following counter example: Let $S = a.b$, $I = a.b + c.d$, $P = c.e$. Then $S \sqsubseteq_{\frac{1}{3}bis} I$, but $S + P \not\sqsubseteq_{\frac{1}{3}bis} I + P$, because $I + P$ has more non-determinism (traces cd and ce) than $S + P$ (trace ce).

Remark. This result, that trace safety follows from congruence, can be explained intuitively. If I strictly extends S then it is possible that I makes assumptions that are somehow contradictory with the assumptions of the context. So, in general it is not allowed to extend a specification if the refinement is a congruence.

4 Interrelationship between refinements

Having defined various refinements, and investigated their individual properties, it is time to investigate their interrelationship. We first consider implications between refinements, and then equivalences between DTS refinements and their LTS counterparts.

Theorem 4. *All the arrows in figure 1 denote strict implications.*

Two new refinements. In view of the missing edge between $\frac{1}{3}$ -bisimulation and implementation, we have searched for other refinements that do give rise to a complete cube. Inspired by the forthcoming proposition 8.a-c, the new refinements, \sqsubseteq_x and \sqsubseteq_y , would be defined as follows:

$$\begin{aligned} S \sqsubseteq_x I &\Leftrightarrow \text{lhs}(S) \sqsubseteq_{\frac{1}{3}\text{bis}} \text{lhs}(I) \\ S \sqsubseteq_{\text{imp}} I &\Leftrightarrow \text{lhs}(S) \sqsubseteq_y \text{lhs}(I) \end{aligned}$$

However, these definitions give not much insight. Attempts to eliminate the LTS concepts in the definition of \sqsubseteq_x in favour of DTS concepts lead to the following definition, called red-ext:

$$S \sqsubseteq_{\text{red-ext}} I \Leftrightarrow S \sqsubseteq_{\text{red}} \sqsubseteq_{\text{ext}} I$$

Here we use juxtaposition to denote relational composition; that is, $(\sqsubseteq_{\text{red}} \sqsubseteq_{\text{ext}}) = (\sqsubseteq_{\text{red}}) \circ (\sqsubseteq_{\text{ext}})$.

The individual properties of red-ext are summarised in table 2, while part of theorem 7 asserts the success of our attempt. The form of the definition leads us to the following alternative characterisation of implementation:

Proposition 5.

$$S \sqsubseteq_{\text{imp}} I \Leftrightarrow S \sqsubseteq_{\text{ext}} \sqsubseteq_{\text{red}} I$$

Regarding the requested refinement \sqsubseteq_y , the preceding successful approach for \sqsubseteq_x leads us to the following definition and proposition:

$$S \sqsubseteq_{\text{abs-}\frac{2}{3}\text{bis}} I \Leftrightarrow S \sqsubseteq_{\text{abs}} \sqsubseteq_{\frac{2}{3}\text{bis}} I$$

Proposition 6.

$$S \sqsubseteq_{\frac{1}{3}\text{bis}} I \Leftrightarrow S \sqsubseteq_{\frac{2}{3}\text{bis}} \sqsubseteq_{\text{abs}} I$$

The individual properties of $\text{abs-}\frac{2}{3}\text{bis}$ are summarised in table 2, while part of theorem 7 asserts the success of our attempt.

Theorem 7. *All the arrows to and from $\text{abs-}\frac{2}{3}\text{bis}$ and red-ext in figure 2 denote strict implications.*

Theorem 7 shows that each LTS refinement implies a DTS refinement. The converse implications however are not true, but they *do* hold if the LTS refinement is taken on the canonical form of the compared DTSs.

Theorem 8.

- a. $S =_F I \Leftrightarrow \text{lhs}(S) =_{\text{bis}} \text{lhs}(I)$
- b. $S \sqsubseteq_{\text{red}} I \Leftrightarrow \text{lhs}(S) \sqsubseteq_{\frac{2}{3}\text{bis}} \text{lhs}(I)$
- c. $S \sqsubseteq_{\text{ext}} I \Leftrightarrow \text{lhs}(S) \sqsubseteq_{\text{abs}} \text{lhs}(I)$
- d. $S \sqsubseteq_{\text{red-ext}} I \Leftrightarrow \text{lhs}(S) \sqsubseteq_{\frac{1}{3}\text{bis}} \text{lhs}(I)$
- e. $S \sqsubseteq_{\text{imp}} I \Leftrightarrow \text{lhs}(S) \sqsubseteq_{\text{abs-}\frac{2}{3}\text{bis}} \text{lhs}(I)$

As said in the introduction, the significance of the theorem is that every DTS refinement can be proven by proving its LTS counterpart. This means, no traces and refusals have to be computed, only the canonical form of the compared DTSs, which is hopefully a less costly operation. Furthermore, this result makes tools for labelled transition systems applicable for decorated traces systems.

5 Related work

There are several papers that present overviews of refinements. None of them however gives an overview as we did with comparisons between the various refinements for LTSs (labelled transition systems) and DTSs (decorated trace systems; traces and refusals).

Important overviews are given by Bowman *et al.* [DBBS96], Brinksma *et al.* [BSS87], and Van Glabbeek [Gla90]. Bowman *et al.* [DBBS96] define and compare various refinements for LOTOS and Z. All relations are defined on processes, as in this paper. Brinksma *et al.* [BSS87] define and compare testing equivalence (the same as failure equivalence), reduction, extension and implementation, all of which are based on failure semantics [Hoa85]. Van Glabbeek [Gla90] studies and compares various equivalences, commonly used in process algebra.

Another interesting overview is given by Lynch and Vaandrager [LV95]. They discuss refinements for automata. These refinements can easily be translated to the context of LTSs, since an LTS is an automaton. The relations however do not allow reduction of nondeterminism, which is a very serious shortcoming.

The notion of a canonical form is due to He Jifeng [He89]. His work inspired us to prove the equivalence between a refinement in failure semantics and a refinement in bisimulation semantics using canonical forms. He Jifeng shows that a combination of forward and backward $\frac{2}{3}$ -bisimulation provides a complete proof method with respect to reduction. Advantage of our approach is that no backward simulation is needed to prove a refinement in failure semantics. Furthermore, our approach encompasses more refinements than just reduction and $\frac{2}{3}$ -bisimulation.

Finally, some work by Cleaveland and Hennessy [CH93] is similar in spirit to ours. They give a procedure to decide a testing relation for LTSs by transforming the transition systems of the two compared processes and deciding a certain kind of bisimulation between these two transformed transition systems. Our work, on the other hand, is more general since it applies to DTSs, and therefore also to LTSs, since every LTS determines a DTS.

References

- [BSS87] E. Brinksma, G. Scollo, and C. Steenbergen. LOTOS specifications, their implementations and tests. In *Protocol Specification, Testing and Verification VI*, 1987.
- [CH93] R. Cleaveland and M. Hennessy. Testing equivalence as a bisimulation equivalence. *Formal Aspects of Computing*, 5(1):1–20, 1993.

- [DBBS96] J. Derrick, H. Bowman, E.A. Boiten, and M. Steen. Comparing LOTOS and Z refinements. In *Formal Description Techniques IX*, 1996.
- [EF99] R. Eshuis and M.M. Fokkinga. Comparing refinements for failure and bisimulation semantics — Proofs. Technical report, Faculty of Computing Science, Enschede, the Netherlands, 1999. Obtainable from <ftp://ftp.cs.utwente.nl/pub/doc/MAICS/99-WAIT99TheProofs.ps.Z> .
- [Gla90] R. van Glabbeek. The linear time – branching time spectrum. In *Proceedings Concur'90*. Springer-Verlag, 1990.
- [He89] J. He. Process simulation and refinement. *Formal Aspects of Computing*, 1(3):229–241, 1989.
- [Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [LS89] K. Larsen and A. Skou. Bisimulation through probabilistic testing. In *Proceeding Principles Of Programming Languages*, 1989.
- [LV95] N.A. Lynch and F.W. Vaandrager. Forward and backward simulations, I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.
- [Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
- [Par81] D. Park. Concurrency and automata on infinite sequences. In *Proceedings 5th GI Conference*, 1981.
- [RF99] R. van Rein and M.M. Fokkinga. Protocol Assuring Universal Language. *Proceedings of the 1999 conference on Formal Methods for Distributed Object Oriented Systems*, 1999.
- [SF98] E. Strijbos and M.M. Fokkinga. Personal communication, 1998.
- [Spi92] J.M. Spivey. *The Z notation — second edition*. Prentice Hall, 1992.
- [WD96] J. Woodcock and J. Davies. *Using Z — Specification, Refinement and Proof*. Prentice Hall, 1996.