Fundamenta Informaticae XX (2001) 1–25
IOS Press

Comparing Refinements for Failure and Bisimulation Semantics *

Rik Eshuis†
Dept. INF
University of Twente
Enschede, the Netherlands
eshuis@cs.utwente.nl

Maarten M. Fokkinga
Dept. INF
University of Twente
Enschede, The Netherlands
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 failure 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, labelled transition systems, bisimulation semantics, failure semantics, decorated traces

*Address for correspondence: Dept. INF, University of Twente, P.O.Box 217, NL-7500 AE Enschede, The Netherlands

†Partially supported by NWO/SION, grant nr. 612-62-02 (DAEMON)

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 and will strongly direct our investigations. 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.)

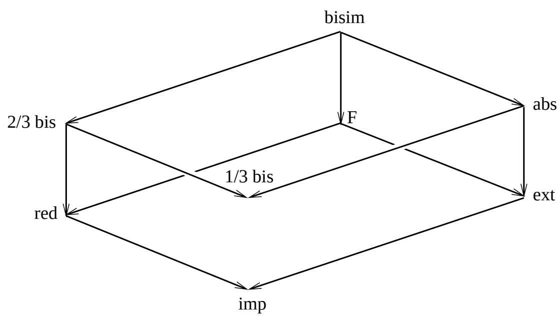
Both kinds of transformation apply to a single notion of substitutability. A specification is refined by an implementation if usage of the system according to the specification is still valid – and has the same observable results – if actually the system is built according to the implementation.

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 bisimulation semantics and failure semantics. In bisimulation semantics, 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 failure semantics, a process is a *failure system* (FS), consisting of traces "decorated" with refusals, and processes are identified if their traces and refusals are the same. A failure is a pair of trace and refusal. Consequently, refinement between processes is defined in terms of 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. In section 5 we will study refinements for other semantics of processes.

Results. Although refinements on LTSs and FSs are defined quite differently, they are strongly related. In particular, there exist pairs of LTS and FS refinements that have (almost) the same properties. Every LTS refinement implies its FS counterpart. (This is well known for several pairs of refinements we discuss, but not for all of them.) Conversely, and that is what we consider the main result of this paper: every FS 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 FS 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. Refinements at the left ($\frac{2}{3}$*bis* and *red*) allow for a reduction of non-



Symbol	Abbreviation	Defined by
bisim	bisimulation	[12],[10]
$\frac{2}{3}bis$	$\frac{2}{3}$-bisimulation	[8]
	ready simulation	[8]
abs	*abs*-bisimulation	
$\frac{1}{3}bis$	$\frac{1}{3}$-bisimulation	[4]
	forward simulation	[17]
F	failure equivalence	[7]
	testing equivalence	[1]
red	reduction	[7],[1]
ext	extension	[1]
imp	implementation	[1]
$X \longrightarrow Y$	X strictly implies Y	

Figure 1. Some refinements in failure and bisimulation semantics

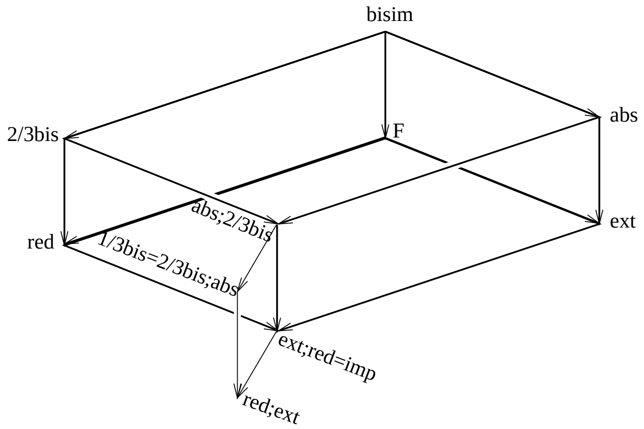
determinism; refinements at the right (*abs* and *ext*) allow for an extension of the functionality. Refinement *abs* has not yet been studied in the literature. The well-known items in the figure have already been related to each other by the 'strictly implies' relation $\rightarrow$ by other people. Van Glabbeek [5] classified the left back face: $bisim, \frac{2}{3}bis, F, red$. Brinksma *et al.* [1] classified the bottom face, consisting of the refinements for failure semantics: F, red, ext, imp. Finally, Derrick *et al.* [4] 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 [6]. The significance of this result is the insight it provides into the nature of refinemements for different semantic domains.

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 these results are summarised in Figure 2.

Structure of the paper. The remainder of this paper is organised as follows. In section 2 we present the LTS and FS 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 FSs and LTSs and to present their



Properties of the refinements:

Congruence and trace safe:
$bisim, \frac{2}{3}bis, red, F$

Refusal safe:
all except $\frac{1}{3}bis$ and $red;ext$

Pre-order:
all except $abs;\frac{2}{3}bis$ and imp

Figure 2. Classification of refinements in failure and bisimulation semantics

properties. In section 4 we classify the refinements and discuss how LTS refinements relate to their FS counterpart. In section 5 we study refinements and canonical processes for other process semantics. The paper ends with a short section on related work.

The proofs are in the appendix.

2. Processes

In this section, we give two definitions of the notion of process: the LTS variant (a 'labelled transition system'), and the FS variant (also known as 'failure system').

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

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

One of the start states is chosen as initial state. The definition of an LTS is a generalisation of the original one (see e.g. [11]), 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:

$$s \xrightarrow{a} s' \quad\Leftrightarrow\quad (s, a, s') \in (\rightarrow)$$
$$s \xrightarrow{a} \quad\Leftrightarrow\quad \exists\, s' \bullet s \xrightarrow{a} s'$$
$$s \stackrel{a}{\not\rightarrow} \quad\Leftrightarrow\quad \nexists s' \bullet s \xrightarrow{a} s'$$

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:

$$\begin{array}{lcl} \mathit{initials}(s) & = & \{a \in \mathit{Act} \mid s \xrightarrow{a}\} \\ \mathit{deadlock}(s) & \Leftrightarrow & \mathit{initials}(s) = \varnothing \end{array}$$

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 give rise to traces. A *trace* is a sequence $t \in \mathit{Act}^*$; we denote the empty trace by ϵ and trace concatenation by juxtaposition. Relation $\rightarrow$ is extended to traces; for $t = a_1 a_2 \ldots a_n$:

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

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

$$\mathit{Tr}_P \quad = \quad \{t \in \mathit{Act}^* \mid \exists\, s \in \mathit{State} \bullet \mathit{start} \ni\xrightarrow{t} s\}$$

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

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 $t \in \mathit{Tr}$ is a set of actions such that all actions in the set can be refused after the process has executed t. The *refusal set* $\mathit{Ref}(t)$ belonging to a trace $t \in \mathit{Tr}$ consists of all refusals of t:

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

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

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:

$$\mathit{det}_P \quad \Leftrightarrow \quad \forall\, s, s' \in \mathit{State};\ t \in \mathit{Tr} \bullet \mathit{start} \ni\xrightarrow{t} s \wedge \mathit{start} \ni\xrightarrow{t} s' \Rightarrow \mathit{initials}(s) = \mathit{initials}(s')$$

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

Failure systems. A failure system (FS) may be viewed as an abstraction of an LTS, in which only the external properties are kept; it is the triple: $(\mathit{Act}, \mathit{Tr}, \mathit{Ref})$. Function Ref "decorates" each trace t with its refusal set $\mathit{Ref}(t)$. We define: any triple $(\mathit{Act}, \mathit{Tr}, \mathit{Ref})$, where $\mathit{Tr} \in \mathit{Act}^*$ and $\mathit{Ref} \in \mathit{Tr} \longrightarrow \mathbb{P}\,\mathit{Act}$, is an FS, provided that Tr is nonempty and prefix-closed, Ref is nonempty and closed under inclusion, and actions leading outside Tr, must be refused:

$$\begin{array}{lr} \epsilon \in \mathit{Tr} & [\text{non-emptiness } \mathit{Tr}] \\ tt' \in \mathit{Tr} \Rightarrow t \in \mathit{Tr} & [\text{prefix-closedness } \mathit{Tr}] \\ t \in \mathit{Tr} \Rightarrow \varnothing \in \mathit{Ref}(t) & [\text{non-emptiness } \mathit{Ref}] \\ t \in \mathit{Tr} \wedge R \in \mathit{Ref}(t) \wedge R' \subseteq R \Rightarrow R' \in \mathit{Ref}(t) & [\text{subset closedness } \mathit{Ref}] \end{array}$$

$$t \in Tr \wedge ta \notin Tr \wedge R \in Ref(t) \Rightarrow R \cup \{a\} \in Ref(t) \qquad \text{[consistency } Ref\text{]}$$

For FSs 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:

$$\begin{array}{lcl} initials(t) & = & \{a \mid ta \in Tr\} \\ deadlock(t) & \Leftrightarrow & initials(t) = \varnothing \end{array}$$

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

$$det_P \Leftrightarrow \forall\, t \in Tr;\, R \in Ref(t) \bullet R \cap initials(t) = \varnothing$$

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

From LTS to FS. 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 FSs, mentioned above. Hence, P determines an FS, denoted $fs(P) = (Act, Tr_P, Ref_P)$. Now the question arises whether the notions of initials, deadlock and determinism for $fs(P)$ are consistent with those for P. The following proposition asserts that this is the case.

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

1. $(\cup_{s \in State \mid start \ni \xrightarrow{t} s} initials_P(s)) = initials_Q(t)$
2. $(\forall_{s \in State \mid start \ni \xrightarrow{t} s}\ deadlock_P(s)) \Leftrightarrow deadlock_Q(t)$
3. $det_P \Leftrightarrow det_Q$

From FS to LTS. In the previous paragraph, we saw that every LTS P can be transformed into an FS $fs(P)$. Conversely, an FS $P = (Act, Tr, Ref)$ 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:

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

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

Proposition 2. Let P be an FS. Then $P = fs(lts(P))$.

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

Proposition 3. Let P be an FS and $Q = lts(P)$. Then $det_P \Leftrightarrow det_Q$.

$$\frac{}{a \xrightarrow{a} \textbf{stop}} \qquad \frac{}{a.P \xrightarrow{a} P} \qquad \frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'} \qquad \frac{Q \xrightarrow{a} Q'}{P + Q \xrightarrow{a} Q'}$$

$$\frac{P \xrightarrow{a} P' \quad a \notin A}{P \,||_A\, Q \xrightarrow{a} P' \,||_A\, Q} \qquad \frac{Q \xrightarrow{a} Q' \quad a \notin A}{P \,||_A\, Q \xrightarrow{a} P \,||_A\, Q'} \qquad \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q' \quad a \in A}{P \,||_A\, Q \xrightarrow{a} P' \,||_A\, Q'}$$

Table 1. Structural operational semantics for the process notation

Process notation. In order to denote (many, but certainly not all) finite LTSs and FSs, 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 (.), choice (+) and interleaving with communication ($||_A$) as operators to compose processes. The following grammar describes these process terms:

$$P ::= a \mid a.P \mid P + P \mid P \,||_A\, P$$

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 single start state. The transition relation is the least relation satisfying the implications, written as inference rules, in table 1. Since each LTS determines an FS, a term also denotes an FS. In the sequel, we shall blur the distinction between processes and process terms: we shall use a process term P for the LTS/FS 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 [14]). 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:

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

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:

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

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:

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

We consider the property of refusal safety so important, that we shall guarantee it when defining a refinement, whenever there is a simple way to do so (as in all refinements for FS). 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 FSs, and next refinements for LTSs.

3.1. FS refinements

For FSs, each refinement $\sqsubseteq$ is defined in terms of inclusions between the components Act, Tr, and Ref of S and I. Regarding Act, it is natural to require that:

(1) $Act_S \;=\; Act_I$

since otherwise the traces and refusals cannot be compared. Regarding Ref, we have already said that we wish to require refusal safety:

(2) $\forall\, t : Tr_s \cap Tr_I \bullet Ref_I(t) \subseteq Ref_S(t)$

Regarding Tr, there are precisely four ways to combine inclusions between Tr_S and Tr_I:

(3) $Tr_S \;=\; Tr_I$

(4) $Tr_S \;\supseteq\; Tr_I$

(5) $Tr_S \;\subseteq\; Tr_I$

(6) $true$

Thus we define four refinements:

$$S =_F I \quad\equiv\quad (1) \wedge (2) \wedge (3) \qquad \text{Failure equivalence}$$

$$
\begin{array}{llll}
S \sqsubseteq_{red} I & \equiv & (1) \wedge (2) \wedge (4) & \text{Reduction} \\
S \sqsubseteq_{ext} I & \equiv & (1) \wedge (2) \wedge (5) & \text{Extension} \\
S \sqsubseteq_{imp} I & \equiv & (1) \wedge (2) \wedge (6) & \text{Implementation}
\end{array}
$$

We shall now briefly discuss them in order.

Failure equivalence. This equivalence was first introduced by Hoare [7]. It plays the role of identity in CSP. Our formulation is the same as the one presented by Brinksma *et al.* [1] under the name *testing equivalence* for LOTOS. Regarding the name 'failure' equivalence, note that in CSP a failure is a pair (t, R), where $R \in Ref(t)$; in the original definition these pairs were used. Properties of this relation are summarised in table 2. When used as a refinement, the most distinguishing property of $=_F$ is that it is symmetric. That means that specification and implementation cannot be distinguished: they are all equal (i.e., express the same observable behaviour). Here is an example: Let $S = a.b.c + a.b.d$ and $I = a.(b.c + b.d)$; then $S =_F I$.

Property	$=_F$	$\sqsubseteq_{red}$	$\sqsubseteq_{ext}$	$\sqsubseteq_{imp}$	$=_{bis}$	$\sqsubseteq_{\frac{2}{3}bis}$	$\sqsubseteq_{abs}$	$\sqsubseteq_{\frac{1}{3}bis}$	$\sqsubseteq_{red\text{-}ext}$	$\sqsubseteq_{abs\text{-}\frac{2}{3}bis}$
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 FSs and LTSs

Reduction refinement. This refinement was first defined for use within CSP [7]. It is the only refinement that is used in CSP. However, the other definitions of refinement are useful for CSP too. Reduction is also used in LOTOS [1], of which the definition coincides with ours. Properties of the 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. Here is an example: Let $S = a.b.c + a.b.d$ and $I = a.b.c$; then $S \sqsubseteq_{red} I$. Note that $Ref_I(ab) \subset Ref_S(ab)$.

Extension refinement. This refinement was first defined by Brinksma *et al.* [1] 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. Note that the implementation can exhibit more behaviour than the specification. The implementation deadlocks less often than the specification for the traces that the 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.* [1]. If $Tr_S \subset Tr_I$ the following counterexample holds: Let $P = a.b$, $S = c$, and

$I = a + c$; then $S \sqsubseteq_{ext} I$, 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, the 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 refinement. This refinement was first used in LOTOS [1], and is also known under the name *conformance*. The 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 $I = a.(b + c)$; then $S \sqsubseteq_{imp} T \sqsubseteq_{imp} I$ 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 (for 'Abstract' and 'Concrete') denote arbitrary states of S and I, respectively.

For LTSs, the refinements are defined in terms of the capability of 'doing an a-step', the main ingredient in the definition of LTSs. More precisely, each refinement $\sqsubseteq$ is defined in terms of the existence of a 'simulation relation' R between states of S and I. We call a relation R a *simulation* relation if, informally, R-related states are comparable in the capability of doing a-steps; formally, if:

$$(7) \quad A \,\underline{R}\, C \wedge A \xrightarrow{a} A' \boxed{\wedge\, C \xrightarrow{a} C''}^{1} \quad \Rightarrow \quad \exists\, C' \bullet C \xrightarrow{a} C' \boxed{\wedge\, A' \,\underline{R}\, C'}^{2}, \quad \text{and}$$

$$(8) \quad A \,\underline{R}\, C \wedge C \xrightarrow{a} C' \boxed{\wedge\, A \xrightarrow{a} A''}^{3} \quad \Rightarrow \quad \exists\, A' \bullet A \xrightarrow{a} A' \boxed{\wedge\, A' \,\underline{R}\, C'}^{4}$$

The boxed parts 1 through 4 can be kept or omitted, thus giving rise to 16 variants. However, some possibilities are ruled out by our intuitive and informal requirements about refinements, namely 'reduction of nondeterminism' and 'functional extension' as discussed in Section 1. Before discussing these, let us first explain implication $(7)^{-1-2}$, that is, (7) with both boxes deleted. It says that all actions that can be done in A can also be done in C (so the implementation does not deadlock when the specification does not). Similarly, requirement $(8)^{+3+4}$ says that for each initial action a of both A and C, the successor states of C when a is executed, are again related with some successor states of A.

Adding or keeping box 1 weakens the implication so much that it conflicts with the intuition about refinements; the presence of the boxed part 1 allows for introducing deadlock into the implementation: even though in S the a-step is possible out of state A, this step is not *required* to exist in the simulating state C in I. Also, deleting box 4 weakens the implication too much; the absense of the boxed part 4 allows for the introduction of nondeterminism into the implementation: there can be several a-steps out of C whose successor states are not related in any way to the successors of a-steps in A. Thus we are left with only adding or deleting boxes 2 and 3, giving four variants in total:

R is a *bisimulation* relation	$\Leftrightarrow$	$\forall A, A', A'', C, C', C'' \bullet (7)^{-1+2} \wedge (8)^{-3+4}$
R is a $\frac{2}{3}$-*bisimulation* relation	$\Leftrightarrow$	$\forall A, A', A'', C, C', C'' \bullet (7)^{-1-2} \wedge (8)^{-3+4}$
R is an *abs-bisimulation* relation	$\Leftrightarrow$	$\forall A, A', A'', C, C', C'' \bullet (7)^{-1+2} \wedge (8)^{+3+4}$
R is a $\frac{1}{3}$-*bisimulation* relation	$\Leftrightarrow$	$\forall A, A', A'', C, C', C'' \bullet (7)^{-1-2} \wedge (8)^{+3+4}$

Actually, the relations should be qualified as *forward* 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 predecessor of C simulate those of A. We do not deal with backward simulation in this paper.

We shall now define and briefly discuss the four refinements, one for each of the above simulation relations. An obvious common requirement 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 equivalence. We define $S =_{bis} I$ if: there exists a bisimulation relation that is total on both $start_S$ and $start_I$. The equivalence is one of the strongest known equivalences for processes. It was first defined by Park [12], as an improvement on the equivalences used until then by Milner for CCS. Note that we do not consider internal actions. This equivalence now forms the cornerstone of CCS [10][11]; it defines identity on CCS processes. Properties of the equivalence are shown in table 2. Like failure equivalence, this refinement is symmetric. Here is an example: Let $S = a.b.c + a.b.d$ and $I = a.(b.c + b.d)$; then $S \neq_{bis} I$. Here is another example: Let $S = a.b$ and $I = a.b + a.b$; then $S =_{bis} I$.

$\frac{2}{3}$-*bisimulation refinement.* We define $S \sqsubseteq_{\frac{2}{3}bis} I$ if: there exists a $\frac{2}{3}$-bisimulation relation that is total on $start_I$. The refinement was introduced by Larsen and Skou [8]. It is also known as *ready simulation*. Independently, He Jifeng introduced an equivalent definition [6]. Properties of this refinement are summarised in table 2. As reduction, the refinement 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 as shown by the following counterexample by Van Glabbeek [5]: Let $S = a.b.c + a.(b.c + b.d)$ and $I = a.(b.c + b.d)$; then $S \sqsubseteq_{\frac{2}{3}bis} I \sqsubseteq_{\frac{2}{3}bis} S$, but $S \neq_{bis} I$.

Abs-bisimulation refinement. We define $S \sqsubseteq_{abs} I$ if: there exists an abs-bisimulation relation that is total on $start_S$ and $start_I$. The refinement had been conceived for use in Paul [13], a language to express behavioural aspects of object-oriented designs. The refinement has not been actually used in Paul, since after all, failure semantics seemed more appropriate. 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 $a \notin initials(A)$. In this way, new traces can be added to S. The refinement is not a congruence and not a partial order. The following counterexample ([15]) shows that this relation is not anti-symmetric: Let $S = a.(a + a.b.c)$ and $I = a.(a + a.b + a.b.c)$; then $S \sqsubseteq_{abs} I \sqsubseteq_{abs} S$ but $S \neq_{bis} I$.

$\frac{1}{3}$-bisimulation refinement, Z forward simulation refinement. We define $S \sqsubseteq_{\frac{1}{3}bis} I$ if: there exists a $\frac{1}{3}$-bisimulation relation that is total on $start_I$. The refinement is used in Z under the name forward simulation [17]. Translation of Z refinement to processes gives rise to $\frac{1}{3}$-bisimulation. This latter refinement is defined by Derrick *et al.* [4] and is an adaptation of the $\frac{2}{3}$-bisimulation (ready simulation) that we discussed above. 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 $I = a.(b + c.e)$; then 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. The refinement is not a partial order. For anti-symmetry, the following is a counterexample ([15]): Let $S = a.(a + a.b.c)$ and $I = a.(a + a.b + a.b.c)$; then $S \sqsubseteq_{\frac{1}{3}bis} I \sqsubseteq_{\frac{1}{3}bis} S$ but $S \neq_{bis} I$. Just like abs-bisimulation, $\frac{1}{3}$-bisimulation is not a congruence as shown by the following counterexample: 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. The latter 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 FS 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 theorem 8.a-c, the new refinements, $\sqsubseteq_x$ and $\sqsubseteq_y$, would be defined as follows:

$$S \sqsubseteq_x I \quad \Leftrightarrow \quad lts(S) \sqsubseteq_{\frac{1}{3}bis} lts(I)$$
$$S \sqsubseteq_{imp} I \quad \Leftrightarrow \quad lts(S) \sqsubseteq_y lts(I)$$

However, these definitions give not much insight. Our attempt to eliminate the LTS concepts in the definition of $\sqsubseteq_x$ in favour of FS concepts led to the following definition for x, now renamed into red-ext:

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

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

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

Proposition 5.

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

Regarding the requested refinement $\sqsubseteq_y$, the preceding successful approach for $\sqsubseteq_x$ leads us to the following definition for y, now renamed into abs-$\frac{2}{3}$bis, and the following proposition:

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

Proposition 6.

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

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

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

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

Theorem 8.

$$\begin{array}{lll}
a. & S =_F I & \Leftrightarrow \quad lts(S) =_{bis} lts(I) \\
b. & S \sqsubseteq_{red} I & \Leftrightarrow \quad lts(S) \sqsubseteq_{\frac{2}{3}bis} lts(I) \\
c. & S \sqsubseteq_{ext} I & \Leftrightarrow \quad lts(S) \sqsubseteq_{abs} lts(I) \\
d. & S \sqsubseteq_{red\text{-}ext} I & \Leftrightarrow \quad lts(S) \sqsubseteq_{\frac{1}{3}bis} lts(I) \\
e. & S \sqsubseteq_{imp} I & \Leftrightarrow \quad lts(S) \sqsubseteq_{abs\text{-}\frac{2}{3}bis} lts(I)
\end{array}$$

Proof:
The $\Leftarrow$-parts are all similar and follow from Theorems 4 and 7 and Proposition 2; for (b) it reads:

$$\begin{array}{lr}
lts(S) \sqsubseteq_{\frac{2}{3}bis} lts(I) & \\
\Rightarrow lts(S) \sqsubseteq_{red} lts(I) & [\text{Thm } 4] \\
\Leftrightarrow fs(lts(S)) \sqsubseteq_{red} fs(lts(I)) & [\text{def } red, lts] \\
\Leftrightarrow S =_F fs(lts(S)) \sqsubseteq_{red} fs(lts(I)) =_F I & [\text{Prop } 2] \\
\Leftrightarrow S \sqsubseteq_{red} I & [\text{on FSs, } =_F \text{ is } =]
\end{array}$$

For the $\Rightarrow$-parts of cases a-c, we give a relation R for the desired conclusions, respectively, using the auxiliary notion $oldinit_{XY}(t) = \{a \mid ta \in Tr_X \wedge ta \notin Tr_Y\}$:

$$\begin{array}{ll}
a. & R = \{((t, R_A), (t, R_C)) \in States_S \times States_I \mid R_A = R_C\} \\
b. & R = \{((t, R_A), (t, R_C)) \in States_S \times States_I \mid R_A = R_C \cup oldinit_{SI}(t)\} \\
c. & R = \{((t, R_A), (t, R_C)) \in States_S \times States_I \mid R_A \supseteq R_C\}
\end{array}$$

We now show case b: relation R is a $\frac{2}{3}$-bisimulation relation for $lts(S) \sqsubseteq_{\frac{2}{3}bis} lts(I)$. The reasoning for cases a and c is easier, and therefore omitted.

First, we prove rule 1 of the definition: $A \ \underline{R} \ C \wedge A \xrightarrow{a} \ \Rightarrow C \xrightarrow{a}$. On account of $A \ \underline{R} \ C$ the arbitrary states A and C have the form (t, R_A) and (t, R_C) with the same t. Now:

$$
\begin{array}{ll}
(t, R_A) \xrightarrow{a} & \\
\Rightarrow a \notin R_A \wedge ta \in Tr_S & [\text{def } lts, \rightarrow] \\
\Leftrightarrow a \notin R_A \wedge ta \in Tr_S \wedge a \notin oldinit_{SI}(t) & [oldinit_{SI}(t) \subseteq R_A] \\
\Leftrightarrow a \notin R_A \wedge ta \in Tr_S \wedge (ta \notin Tr_S \vee ta \in Tr_I) & [\text{def } oldinit_{SI}(t)] \\
\Leftrightarrow a \notin R_A \wedge ta \in Tr_S \wedge ta \in Tr_I & [\text{logic}] \\
\Rightarrow a \notin R_C \wedge ta \in Tr_I & [R_C \subseteq R_A] \\
\Leftrightarrow (t, R_C) \xrightarrow{a} & [\text{def } lts, \rightarrow]
\end{array}
$$

Next, we prove rule 2 of the definition: $A \ \underline{R} \ C \wedge C \xrightarrow{a} C' \Rightarrow \exists A' \bullet A \xrightarrow{a} A' \wedge A' \ \underline{R} \ C'$.

$$
\begin{array}{ll}
(t, R_A) \ \underline{R} \ (t, R_C) \wedge (t, R_C) \xrightarrow{a} (ta, R'_C) & \\
\Rightarrow ta \in Tr_I \wedge R'_C \in Ref_I(ta) & [\text{def } lts, \rightarrow] \\
\Leftrightarrow ta \in Tr_I \wedge R'_C \in Ref_I(ta) \wedge ta \in Tr_S & [Tr_I \subseteq Tr_S] \\
\Leftrightarrow ta \in Tr_I \wedge R'_C \cup oldinit_{SI}(ta) \in Ref_I(ta) \wedge ta \in Tr_S & [\text{consistency } Ref_I(ta), \text{def } oldinit_{SI}] \\
\Rightarrow R'_C \cup oldinit_{SI}(ta) \in Ref_S(ta) \wedge ta \in Tr_S & [Ref_I(ta) \subseteq Ref_S(ta)] \\
\Leftrightarrow (t, R_A) \xrightarrow{a} (ta, R'_C \cup oldinit_{SI}(ta)) & [\text{def } lts, \rightarrow] \\
\Rightarrow \exists A' \bullet (t, R_A) \xrightarrow{a} A' \ \wedge \ A' \ \underline{R} \ (ta, R'_C) & [\text{def } R]
\end{array}
$$

For case e, we argue as follows.

$$
\begin{array}{ll}
S \sqsubseteq_{imp} I & \\
\Leftrightarrow S \sqsubseteq_{ext} \sqsubseteq_{red} I & [\text{Prop } 5] \\
\Leftrightarrow \exists P \bullet S \sqsubseteq_{ext} P \sqsubseteq_{red} I & [\text{relational composition}] \\
\Leftrightarrow \exists P \bullet lts(S) \sqsubseteq_{abs} lts(P) \sqsubseteq_{\frac{2}{3}bis} lts(I) & [\text{Thm 8.b,c}] \\
\Leftrightarrow lts(S) \sqsubseteq_{abs} \sqsubseteq_{\frac{2}{3}bis} lts(I) & [\text{relational composition}] \\
\Leftrightarrow lts(S) \sqsubseteq_{abs\text{-}\frac{2}{3}bis} lts(I) & [\text{def } \sqsubseteq_{abs\text{-}\frac{2}{3}bis}\ ]
\end{array}
$$

Case d is proven by similar reasoning. □

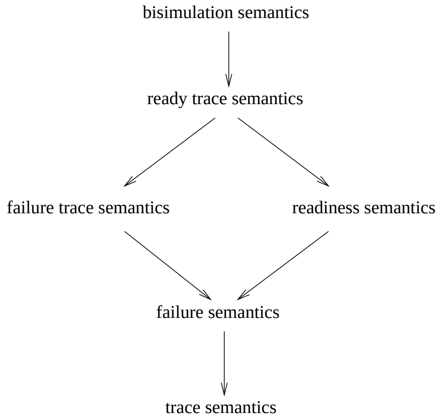


Figure 3. Semantic domains

5. Generalisations

Up till now we considered two semantics for processes, namely LTSs and FSs. The LTSs are used in the domain of bisimulation semantics and the FSs in the domain of failure semantics. Figure 3 shows some other well-known semantics for processes (see e.g. [5]). The question arises whether our investigation and results can be extended to these other semantics. In order to do so, we will first define suitable systems (like FS for failure semantics); thereafter we return to a discussion of refinements.

Readiness semantics. An *acceptance* A of a process after trace t is a set of actions that can be accepted after t is executed; the *acceptance set* $Acc(t)$ is the set of possible acceptances. Formally, for LTS P and $t \in Tr$:

$$A \in Acc(t) \Leftrightarrow \exists\, s \bullet start \ni\!\xrightarrow{t} s \wedge A = initials(s)$$

Now we define a ready system (RS) to be any triple (Act, Tr, Acc) provided that the following constraints hold: Tr nonempty and prefix-closed, every trace has a nonempty acceptance set, and every action that is acceptable should be in the trace set (and vice versa):

$$\epsilon \in Tr \qquad \text{[nonemptyness } Tr]$$
$$tt' \in Tr \Rightarrow t \in Tr \qquad \text{[prefix-closedness } Tr]$$
$$t \in Tr \Rightarrow Acc(t) \neq \varnothing \qquad \text{[nonemptyness } Acc]$$
$$ta \in Tr \Rightarrow \exists\, A \in Acc(t) \bullet a \in A \qquad \text{[consistency } Tr]$$
$$a \in A \in Acc(t) \Rightarrow ta \in Tr \qquad \text{[consistency } Acc]$$

It is easy to check that for LTS P, the triple $rs(P) = (Act_P, Tr_P, Acc_P)$ is an RS. Conversely, given an RS $P = (Act, Tr, Acc)$, the *canonical lts for* P, denoted $lts(P)$, is defined by:

$$\begin{array}{lcl} State & = & \{(t, A) \mid t \in Tr \wedge A \in Acc(t)\} \\ (t, A) \xrightarrow{a} (t', A') & \Leftrightarrow & (t, A) \in State \wedge a \in A \wedge ta = t' \wedge (t', A') \in State \end{array}$$

$$start \quad = \quad \{(\epsilon, A) \mid A \in Acc(\epsilon)\}$$

The following proposition gives some assurance that the characterization is correct.

Proposition 9. Let P be an RS. Then $P = rs(lts(P))$.

Failure trace, ready trace semantics. In the same way as for readiness semantics above (and failure semantics in Section 2), it is not too hard to come up with a definition of a failure trace system FTS, and mappings

$$FTS \underset{fts}{\overset{lts}{\rightleftarrows}} LTS \qquad \text{such that} \qquad fts \circ lts = id$$

and similarly for a ready trace system RTS. Specifically, choose the set *State* of the canonical LTS for a given FTS P to be the set of failure traces of P; similarly, the set *State* of the canonical LTS for a given RTS P is the set of ready traces of P.

The characteristic properties of FTS and RTS are quite easily derived from FS and RS, respectively. We omit further details, since we shall not actually use them.

Refinements. Next, we try to define refinements in the three semantics. Having identified in Section 3 five refinements for both bisimulation and failure semantics, the natural conjecture is that five similar refinements exist for each of the new semantics. Since two of the refinements are compositions of two other refinements, we only need to consider three refinements: equality ($=_F$,$=_{bis}$), reduction of nondeterminism ($\sqsubseteq_{red}, \sqsubseteq_{\frac{2}{3}bis}$), and extension of behaviour ($\sqsubseteq_{ext}, \sqsubseteq_{abs}$).

It turns out that some of these refinements are hard to define in terms of the ingredients of the systems (RS, FTS, RTS). For example, a refinement supporting reduction of nondeterminism is easy to define for FTSs (as inclusion between failure traces), but we see no way to do so for a refinement supporting extension of behaviour. On the other hand, for RSs and RTSs, it is easy to define a refinement supporting extension of behaviour, but we see no way to define a refinement supporting reduction of nondeterminism in terms of the ingredients of those systems.

Now we can apply the notion of canonical process: give the definitions as suggested by Theorem 8. For example, a refinement r supporting reduction of nondeterminism may be defined by

$$P \sqsubseteq_r Q \quad \Leftrightarrow \quad lts(P) \sqsubseteq_{\frac{2}{3}bis} lts(Q)$$

So, rather than a "direct" definition, we may define the refinements indirectly using the notion of canonical process.

Summarising, we have shown in this section two benefits of using canonical processes: first, it gives us some assurance that chosen charactarisation of semantics are valid, and second, it helps us in defining refinements that are hard to define otherwise.

6. Related work

Some work by Cleaveland and Hennessy [2] 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 kind of bisimulation between these two transformed transition systems.

(Testing relations were introduced by De Nicola and Hennessy [3]; they prove that one testing relation, the must preorder, is equivalent to reduction for concrete processes.) There are three differences with our work. First, there is a difference between our canonical process and their transformed transition system, because these transformed transition systems are deterministic (whereas canonical processes can be nondeterministic) and the states are labelled with extra information (whereas states of the canonical process are unlabelled). Second, the bisimulation they use is stronger than ordinary bisimulation (that we use), since not all pairs of states may be in the bisimulation relation, but only those that bear the same labels. This stronger definition is necessary because the transformed systems are deterministic: in the transformation nondeterministic information in the branching structure of the process is lost (and put in the labels of the states). Third, our work is more general since it applies to FSs, and therefore also to LTSs, since every LTS determines an FS.

Next, 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 FSs (failure systems; traces and refusals), and the generalisations to RSs, FTSs, and RTSs.

Important overviews are given by Derrick *et al.* [4], Brinksma *et al.* [1], and Van Glabbeek [5]. Derrick *et al.* [4] define and compare various refinements for LOTOS and Z. All relations are defined on processes, as in this paper. Brinksma *et al.* [1] define and compare testing equivalence (the same as failure equivalence), reduction, extension and implementation, all of which are based on failure semantics [7]. Van Glabbeek [5] studies and compares various equivalences, commonly used in process algebra.

Another interesting overview is given by Lynch and Vaandrager [9]. 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 serious shortcoming for use in practical program development.

The notion of a canonical form is due to He Jifeng [6]. 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 proves that a combination of forward and backward $\frac{2}{3}$-bisimulation provides a complete proof method with respect to reduction. An 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.

Acknowledgement. We are grateful to the reviewers for comments that led to a considerable improvement in the presentation, and to the extension with Section 5. We thank Eerke Boiten for spotting a flaw in a previous version of this paper.

References

[1] E. Brinksma, G. Scollo, and C. Steenbergen. LOTOS specifications, their implementations and their tests. In G. Bochmann and B. Sarikaya, editors, *Protocol Specification, Testing and Verification VI*, pages 349–360. North-Holland, 1987.

[2] R. Cleaveland and M. Hennessy. Testing equivalence as a bisimulation equivalence. *Formal Aspects of Computing*, 5(1):1–20, 1993.

[3] R. de Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1-2):83–133, November 1984.

[4] J. Derrick, H. Bowman, E.A. Boiten, and M. Steen. Comparing LOTOS and Z refinement relations. In *FORTE/PSTV'96*, pages 501–516. Chapman & Hall, 1996.

[5] R. J. van Glabbeek. The linear time-branching time spectrum (extended abstract). In J. C. M. Baeten and J. W. Klop, editors, *CONCUR '90*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297, Amsterdam, The Netherlands, 1990. Springer-Verlag.

[6] J. He. Process simulation and refinement. *Formal Aspects of Computing*, 1(3):229–241, 1989.

[7] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[8] K. G. Larsen and A. Skou. Bisimulation through probabilistic testing (preliminary report). In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 344–352. ACM Press, 1989.

[9] N.A. Lynch and F.W. Vaandrager. Forward and backward simulations, I: Untimed systems. *Information and Computation*, 121(2):214–233, September 1995.

[10] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25(3):267–310, July 1983.

[11] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[12] D. Park. Concurrency and automata on infinite sequences. In Peter Deussen, editor, *5th GI-Conference*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183, Berlin, Heidelberg, and New York, 1981. Springer-Verlag.

[13] R. van Rein and M.M. Fokkinga. Protocol assuring universal language. In Paolo Ciancarini, Alessandro Fantechi, and Robert Gorrieri, editors, *Formal Methods for Open Object-based Distributed Systems*, pages 241–258. Kluwer Academic Publishers, 1999.

[14] J.M. Spivey. *The Z notation — second edition*. Prentice Hall, 1992.

[15] E. Strijbos and M.M. Fokkinga. Personal communication, 1998.

[16] F.W. Vaandrager. On the relationship between process algebra and input/output automata (extended abstract). In *Proceedings* 6^{th} *Annual Symposium on Logic in Computer Science,* Amsterdam, pages 387–398. IEEE Computer Society Press, 1991.

[17] J. Woodcock and J. Davies. *Using Z — Specification, Refinement and Proof.* Prentice Hall, 1996.

Appendix — the Proofs

Proof of Proposition 1. By unfolding the definitions and simple set-theoretic reasoning. For (i):

$$
\begin{aligned}
& \cup_{s \in State \mid start \ni\xrightarrow{t} s} initials_P(s) & \\
&= \cup_{s \in State \mid start \ni\xrightarrow{t} s} \{a \mid s \xrightarrow{a}\} & \text{[def } initials_P\text{]} \\
&= \{a \mid start \ni\xrightarrow{t} s \wedge s \xrightarrow{a}\} & \text{[set theory]} \\
&= \{a \mid ta \in Tr_P\} & \text{[def } Tr_P\text{]} \\
&= initials_Q(t) & \text{[def } initials_Q\text{]}
\end{aligned}
$$

The reasoning for (ii) is similar, and therefore omitted. For (iii) we argue as follows.

$$
\begin{aligned}
& det_P & \\
&\Leftrightarrow \forall\, s, s' \in State;\ t \in Act^* \bullet & \\
&\qquad start \ni\xrightarrow{t} s \wedge start \ni\xrightarrow{t} s' \Rightarrow initials_P(s) = initials_P(s') & \text{[def } det_P\text{]} \\
&\Leftrightarrow \forall\, s \in State;\ t \in Tr_P;\ R \in Ref_P(t) \bullet start \ni\xrightarrow{t} s \Rightarrow R \cap initials_P(s) = \varnothing & \text{[def } Ref\text{]} \\
&\Leftrightarrow \forall\, t \in Tr_P;\ R \in Ref_P(t) \bullet R \cap \bigcup_{s \mid start \ni\xrightarrow{t} s} initials_P(s) = \varnothing & \text{[logic]} \\
&\Leftrightarrow \forall\, t \in Tr_P;\ R \in Ref_P(t) \bullet R \cap initials_Q(t) = \varnothing & \text{[prop 1.1]} \\
&\Leftrightarrow det_Q & \text{[def } det_Q\text{]}
\end{aligned}
$$

□

Proof of Proposition 2. Put $P' = lts(P)$ and $Q = fs(P')$. We decorate all components (except $\rightarrow$) of P' with a prime $'$. The following lemma is easily provable by induction on the structure of t:

Lemma 10. For $t, t' \in Tr$ and $R' \in Ref(t')$: $start' \ni\xrightarrow{t} (t', R') \quad \Leftrightarrow \quad t = t'$

Now continuing the proof of Proposition 2, it is clear that $Act_Q = Act' = Act$. We prove for an arbitrary trace t that:

$$
\begin{array}{lllll}
t \in Tr & \Leftrightarrow & t \in Tr' & \Leftrightarrow & t \in Tr_Q \\
R \in Ref(t) & \Leftrightarrow & R \in Ref'(t) & \Leftrightarrow & R \in Ref_Q(t),
\end{array}
$$

where, in the latter line, the components of the former line are assumed to hold. The rightmost equivalences are true by construction of Q ($Tr_Q = Tr'$ and $Ref_Q = Ref'$), so it suffices to show the leftmost equivalences only.

For arbitrary t we argue, for the upper-left equivalence:

$$
\begin{aligned}
& t \in Tr & \\
&\Leftrightarrow \exists\, R \bullet (t, R) \in State' & \text{[def } lts,\ Ref(t) \text{ nonempty]}
\end{aligned}
$$

$\Leftrightarrow \exists\, R \bullet start' \ni\!\xrightarrow{t} (t, R)$ [lemma 10]
$\Leftrightarrow t \in Tr'$ [def Tr']

And for the lower-left equivalence by circular implication, tacitly using that $t \in Tr \wedge t \in Tr'$:

$R \in Ref(t)$
$\Rightarrow (t, R) \in State'$ [def lts]
$\Rightarrow \forall\, a \in R \bullet (t, R)\overset{a}{\not\to}$ [def lts]
$\Rightarrow R \in Ref'(t)$ [def Ref']
$\Rightarrow \exists\, R' \bullet start' \ni\!\xrightarrow{t} (t, R') \wedge \forall\, a \in R \bullet (t, R')\overset{a}{\not\to}$ [def Ref', lemma 10]
$\Rightarrow \exists\, R' \bullet start' \ni\!\xrightarrow{t} (t, R') \wedge \forall\, a \in R \bullet ta \notin Tr \vee a \in R'$ [def lts]
$\Rightarrow \exists\, R' \bullet R' \cup \{a \mid ta \notin Tr\} \in Ref(t) \wedge R \subseteq R' \cup \{a \mid ta \notin Tr\}$ [consistency Ref]
$\Rightarrow R \in Ref(t)$ [Ref subset closed]

□

Proof of Proposition 3. Immediately from Proposition 1 and 2:
$det_P \Leftrightarrow det_{fs(lts(P))} \Leftrightarrow det_{lts(P)}$ □

Proof of Properties of FS refinements in table 2. Rather than discussing all the refinements in detail, we only consider *red*. The proofs for the other refinements are similar.

- $\sqsubseteq_{red}$ is a partial order:

  Reflexivity, transitivity and anti-symmetry immediately follow from the definition and the reflexivity, transitivity and anti-symmetry of $\subseteq$.
- $\sqsubseteq_{red}$ is a congruence for $+$, ., and $||_A$:

  See [1]. The result also follows from the fact that the s.o.s. rules of our process notation satisfy the De Simone format [16].
- $\sqsubseteq_{red}$ is safe w.r.t. refusals:

  $S \sqsubseteq_{red} I$
  $\Leftrightarrow Tr_I \subseteq Tr_S \wedge \forall\, t \in Tr_S \cap Tr_I \bullet Ref_I(t) \subseteq Ref_S(t)$ [def $\sqsubseteq_{red}$]
  $\Rightarrow \forall\, t \in Tr_S \cap Tr_I \bullet Ref_I(t) \subseteq Ref_S(t)$ [logic]

  $\sqsubseteq_{red}$ is safe with respect to traces:

  $S \sqsubseteq_{red} I$
  $\Leftrightarrow Tr_I \subseteq Tr_S \wedge \forall\, t \in Tr_S \cap Tr_S \bullet Ref_I(t) \subseteq Ref_S(t)$ [def $\sqsubseteq_{red}$]
  $\Rightarrow Tr_I \subseteq Tr_S$ [logic]

□

Proof of Properties of LTS refinements in table 2. Rather than discussing all the refinements in detail, we only consider $\frac{1}{3}bis$.

- $\sqsubseteq_{\frac{1}{3}bis}$ is a pre-order, not a partial order:
  - Reflexivity. We have to prove that $S \sqsubseteq_{\frac{1}{3}bis} S$. Let $R = \{(s,s) \mid s \in States_S\}$. Relation R is a $\frac{1}{3}$-bisimulation relation. Furthermore $\{(s,s) \mid s \in start\} \subseteq R$.
  - Transitivity. Given $S \sqsubseteq_{\frac{1}{3}bis} T \sqsubseteq_{\frac{1}{3}bis} I$ with $\frac{1}{3}$-bisimulation relations R_1 and R_2, respectively. Then $R = R_1 \mathbin{;} R_2$ is a $\frac{1}{3}$-bisimulation relation for $S \sqsubseteq_{\frac{1}{3}bis} I$. Furthermore, R is total on $start_I$.
  - Anti-symmetry. $\sqsubseteq_{\frac{1}{3}bis}$ is not anti-symmetric. For a counterexample take $S = a.(a + a.b.c)$ and $I = a.(a + a.b + a.b.c)$ [15]. Now $S \sqsubseteq_{\frac{1}{3}bis} I \sqsubseteq_{\frac{1}{3}bis} S$ but $S \neq_{bis} I$.
- $\sqsubseteq_{\frac{1}{3}bis}$ is a congruence for . and $||_A$, but not for $+$:
  - Case $+$: For a counterexample, take $P = c.d$, $S = a.b$, $I = a.b + c$. Now $S \sqsubseteq_{\frac{1}{3}bis} I$ but $P + S \not\sqsubseteq_{\frac{1}{3}bis} P + I$.
  - Case '.': Let $S \sqsubseteq_{\frac{1}{3}bis} I$ with $\frac{1}{3}$-bisimulation relation R_1. Take $R = \{(a.S, a.I)\} \cup R_1$. Then R is a $\frac{1}{3}$-bisimulation relation for $S \sqsubseteq_{\frac{1}{3}bis} I$, containing $(a.S, a.I)$.
  - Case $||_A$: Let R_1 be a $\frac{1}{3}$-bisimulation relation for $S \sqsubseteq_{\frac{1}{3}bis} I$. Take $R = \{(P \;||_A\; X, P \;||_A\; Y) \mid \exists\, t \in Tr \bullet S \xrightarrow{t} X \wedge I \xrightarrow{t} Y \wedge X \;\underline{R_1}\; Y\}$. Then R is a $\frac{1}{3}$-bisimulation relation for $S \sqsubseteq_{\frac{1}{3}bis} I$, containing $(P \;||_A\; S, P \;||_A\; I)$.
- $\sqsubseteq_{\frac{1}{3}bis}$ is not safe w.r.t. refusals and traces:
  - Refusal safety. For a counterexample, take $S = a.b + a.c.d$ and $I = a.(b + c.e)$. Now $S \sqsubseteq_{\frac{1}{3}bis} I$, but I is not refusal safe, since after trace ac, I always refuses d whereas S always accepts d.
  - Trace safety. For a counterexample, take $S = a.b$ and $I = a.b + c$. Now $S \sqsubseteq_{\frac{1}{3}bis} I$ but $c \notin Tr_S$.

$\square$

Proof of Theorem 4. We consider first the top face of figure 1, then the bottom face, and finally the vertical edges:

a. bisimulation strictly implies $\frac{2}{3}$-bisimulation.
b. bisimulation strictly implies abs-bisimulation.
c. $\frac{2}{3}$-bisimulation strictly implies $\frac{1}{3}$-bisimulation.
d. abs-bisimulation strictly implies $\frac{1}{3}$-bisimulation.
e. failure equivalence strictly implies reduction.
f. failure equivalence strictly implies extension.

g. reduction strictly implies implementation.
h. extension strictly implies implementation.
i. bisimulation strictly implies failure equivalence.
j. $\frac{2}{3}$-bisimulation strictly implies reduction.
k. abs-bisimulation strictly implies extension.
l. $\frac{1}{3}$-bisimulation and implementation are *unrelated*.

a: in [5]. Note that our processes have a start set rather that a simple start state, and that therefore there is a clause "R is total on $start_I$ and $start_S$" in the definition of bisimulation. The proof in [5] can be easily adapted to this generalisation. Similar remarks apply in the sequel.

b: $\Rightarrow$-part. Every bisimulation relation R is also an abs-bisimulation relation, since rule 1 and 2 for bisimulation imply rule 1 and 2 for abs-bisimulation.
$\not\Leftarrow$-part. Let $S = a.b$ and $I = a.b + c$. Now $S \neq_{bis} I$, but $S \sqsubseteq_{abs} I$.

c: in [4].

d: $\Rightarrow$-part. Every abs-bisimulation relation R is also an $\frac{1}{3}$-bisimulation relation, since rule 1 and 2 for abs-bisimulation imply rule 1 and 2 for $\frac{1}{3}$-bisimulation.
$\not\Leftarrow$-part. Let $S = a.b + a.c$ and $I = a.b + d$. Now $S \not\sqsubseteq_{abs} I$, but $S \sqsubseteq_{\frac{1}{3}bis} I$.

e,f,g,h: in [1].

i: in [5].

j: in [5], [6].

k: $\Rightarrow$-part. Let R be an abs-bisimulation relation between S and I, total on $start_S$ and on $start_I$. To show that $Tr_S \subseteq Tr_I$, we prove the stronger claim that for all $t \in Act^*$ and $A \in States_S$:

$$start_S \ni\!\xrightarrow{t} A \quad \Rightarrow \quad \exists\, C \bullet start_I \ni\!\xrightarrow{t} C \wedge A\ \underline{R}\ C$$

We use induction on the structure of t. The case $t = \epsilon$ is simple (R is total on $start_S$). For the case $t = t'a$ we argue as follows:

$$\begin{array}{ll}
start_S \ni\!\xrightarrow{t'a} A & \\
\Leftrightarrow \exists\, A' \bullet start_S \ni\!\xrightarrow{t'} A' \xrightarrow{a} A & [\text{def} \rightarrow] \\
\Rightarrow \exists\, A', C' \bullet start_I \ni\!\xrightarrow{t'} C' \wedge A'\ \underline{R}\ C' \wedge A' \xrightarrow{a} A & [\text{ind hyp}] \\
\Rightarrow \exists\, C', C \bullet start_I \ni\!\xrightarrow{t'} C' \xrightarrow{a} C \wedge A\ \underline{R}\ C & [\text{abs-bis } R\text{, rule 1}] \\
\Leftrightarrow \exists\, C \bullet start_I \ni\!\xrightarrow{t'a} C \wedge A\ \underline{R}\ C & [\text{def} \rightarrow]
\end{array}$$

Using this result, the claim $\forall\, t \in Tr_S \cap Tr_I \bullet Ref_I(t) \subseteq Ref_S(t)$ immediately follows from the following two observations:

- $A\ \underline{R}\ C \Rightarrow initials_S(A) \subseteq initials_I(C)$ [abs-bis R, rule 1]
- $Ref(t) = \{R \mid \exists\, s \bullet start \ni\!\xrightarrow{t} s \wedge R \cap initials(s) = \varnothing\}$

$\not\Leftarrow$-part. Let $S = a.b.c + a.b.d$ and $I = a.(b.c + b.d)$. Now $S \sqsubseteq_{ext} I$, but $S \not\sqsubseteq_{abs} I$.

l: $\not\Rightarrow$-part. Let $S = a.b + a.c.d$ and $I = a.(b + c.e)$. Now $S \sqsubseteq_{\frac{1}{3}bis} I$, but $S \not\sqsubseteq_{imp} I$, because after trace ac S always accepts d, whereas I always refuses d.

$\not\Leftarrow$-part. Let $S = a.b.c + a.b.d$ and $I = a.(b.c + b.d)$. Now $S \sqsubseteq_{imp} I$, but $S \not\sqsubseteq_{\frac{1}{3}bis} I$.

□

Proof of Properties of red-ext in table 2. We only prove reflexivity and transitivity here. Reflexivity follows immediately from the reflexivity of $\sqsubseteq_{red}$ and $\sqsubseteq_{ext}$. Transitivity follows immediately from transitivity of $\sqsubseteq_{red}$ and $\sqsubseteq_{ext}$, and the following lemma.

Lemma 11.

$$S \sqsubseteq_{ext}\sqsubseteq_{red} I \quad \Rightarrow \quad S \sqsubseteq_{red}\sqsubseteq_{ext} I$$

To prove the lemma, assume $S \sqsubseteq_{ext}\sqsubseteq_{red} I$. For later use, we first observe:

$$\forall\, t \in Tr_S \cap Tr_I \bullet Ref_I(t) \subseteq Ref_S(t) \qquad (*)$$

Now define $P = (Act_P, Tr_P, Ref_P)$, by:

$$\begin{aligned} Act_P &= Act_S (= Act_I) \\ Tr_P &= Tr_S \cap Tr_I \\ Ref_P(t) &= Ref_I(t) \cup \{\{a\} \cup R \mid a \in initials_P(t) \wedge R \in Ref_I(t)\} \end{aligned}$$

for all $t \in Tr_P$. It is easy to verify that P is an FS. We shall now show that $S \sqsubseteq_{red} P \sqsubseteq_{ext} I$, thus completing the proof of the lemma.

Part $P \sqsubseteq_{ext} I$ is easy and therefore omitted. Regarding part $S \sqsubseteq_{red} P$, the condition $Tr_P \subseteq Tr_S$ follows immediately from the definition of Tr_P. It remains to show $R \in Ref_P(t) \Rightarrow R \in Ref_S(t)$ for all $t \in Tr$. First we observe:

$R \in Ref_P(t)$

$\Leftrightarrow$ [def Ref_P, Tr_P]

1. $R \in Ref_I(t) \vee$
2. $(\exists\, a \notin initials_I(t);\, R' \in Ref_I(t) \bullet R = R' \cup \{a\}) \vee$
3. $(\exists\, a \notin initials_S(t);\, R' \in Ref_I(t) \bullet R = R' \cup \{a\})$.

Now, in case 1 we have $R \in Ref_S(t)$ by (*). In case 2 we have $R = R' \cup \{a\} \in Ref_I(t)$ by consistency of Ref_I, so $R \in Ref_S(t)$ by (*). In case 3 we have $R' \in Ref_S(t)$ by (*), so $R = R' \cup \{a\} \in Ref_S(t)$ by consistency of Ref_S. □

Proof of Proposition 5. $\Rightarrow$-part. Assume $S \sqsubseteq_{imp} I$. We define a process P such that $S \sqsubseteq_{ext} P \sqsubseteq_{red} I$. Let $P = (Act_P, Tr_P, Ref_P)$, where:

$$\begin{aligned} Act_P &= Act_S (= Act_I) \\ Tr_P &= Tr_S \cup Tr_I \\ Ref_P(t) &= Ref_I(t), && \text{if } t \in Tr_I \\ &= Ref_S(t), && \text{otherwise} \end{aligned}$$

It follows immediately from the definitions of $\sqsubseteq_{ext}$ and $\sqsubseteq_{red}$ that $S \sqsubseteq_{ext} P \sqsubseteq_{red} I$.

$\Leftarrow$-part. See observation (*) in the proof of lemma 11. □

Proof of Proposition 6. $\Rightarrow$-part. Assume $S \sqsubseteq_{\frac{1}{3}bis} I$ with $\frac{1}{3}$-bisimulation relation R. We construct a process P, such that $S \sqsubseteq_{\frac{2}{3}bis} P \sqsubseteq_{abs} I$:

$$\begin{array}{lcl} Act_P & = & Act_S \\ States_P & = & States_S \\ s \xrightarrow{a}_P s' & \Leftrightarrow & s \xrightarrow{a}_S s' \wedge s, s' \in \mathrm{dom}\, R \\ start_P & = & start_S \cap \mathrm{dom}\, R \end{array}$$

Now, $S \sqsubseteq_{\frac{2}{3}bis} P$ follows from the fact that $\{(x, x) \mid x \in \mathrm{dom}\, R\}$ is a $\frac{2}{3}$-bisimulation relation, total on $start_P$. And $P \sqsubseteq_{abs} I$ follows from the fact that R is an abs-bisimulation relation total on $start_P$ and $start_I$.
$\Leftarrow$-part. Assume $S \sqsubseteq_{\frac{2}{3}bis} \sqsubseteq_{abs} I$. Let P be an LTS such that $S \sqsubseteq_{\frac{2}{3}bis} P \sqsubseteq_{abs} I$, with $\frac{1}{3}$-bisimulation relation R_1 and abs-bisimulation relation R_2, respectively. Then $R = R_1 \,;\, R_2$ is a $\frac{1}{3}$-bisimulation relation for $S \sqsubseteq_{\frac{1}{3}bis} I$, total on $start_I$. $\square$

Proof of Properties of abs-$\frac{2}{3}$bis in table 2. We only discuss reflexivity and transitivity. Reflexivity follows from the reflexivity of abs and $\frac{2}{3}$-bisimulation. To show that $\sqsubseteq_{abs\text{-}\frac{2}{3}bis}$ is not transitive, let $S = a.b + a.c.d$, $I = a.(b + c.e)$ and $P = a.b$. Then $S \sqsubseteq_{abs\text{-}\frac{2}{3}bis} P \sqsubseteq_{abs\text{-}\frac{2}{3}bis} I$, but $S \not\sqsubseteq_{abs\text{-}\frac{2}{3}bis} I$. $\square$

Proof of Theorem 7. We consider the arrows as follows:

a. $\frac{1}{3}$-bisimulation strictly implies red-ext.
b. implementation strictly implies red-ext.
c. abs-$\frac{2}{3}$-bisimulation strictly implies implementation.
d. $\frac{2}{3}$-bisimulation strictly implies abs-$\frac{2}{3}$-bisimulation.
e. abs-bisimulation strictly implies abs-$\frac{2}{3}$-bisimulation.
f. abs-$\frac{2}{3}$-bisimulation strictly implies $\frac{1}{3}$-bisimulation.

a: $\Rightarrow$-part. From $S \sqsubseteq_{\frac{1}{3}bis} I$ it follows by proposition 6 that $S \sqsubseteq_{\frac{2}{3}bis} \sqsubseteq_{abs} I$, and so, by theorem 4 (j,k), $S \sqsubseteq_{red} \sqsubseteq_{ext} I$.

$\not\Leftarrow$-part. Let $S = a.b.c + a.b.d$ and $I = a.(b.c + b.d)$. Now $S \sqsubseteq_{red\text{-}ext} I$, but $S \not\sqsubseteq_{imp} I$.

b: $\Rightarrow$-part. See [1], or observation (*) in lemma 11.

$\not\Leftarrow$-part. Although Brinksma *et al.* [1] claim the opposite, $S \sqsubseteq_{red\text{-}ext} I \not\Rightarrow S \sqsubseteq_{imp} I$. For a counterexample, let $S = a.b + a.c.d$ and $I = a.(b + c.e)$. Then $S \sqsubseteq_{red\text{-}ext} I$, but $S \not\sqsubseteq_{imp} I$.

c: Follows immediately from propositions 4 (k,j) and 5.

d: $\Rightarrow$-part. Assume $S \sqsubseteq_{\frac{2}{3}bis} I$. Then $S \sqsubseteq_{abs} S \sqsubseteq_{\frac{2}{3}bis} I$.

$\not\Leftarrow$. Take $S = a.b$ and $I = a.b + d$. Now $S \sqsubseteq_{abs\text{-}\frac{2}{3}bis} I$, but $S \not\sqsubseteq_{\frac{2}{3}bis} I$.

e: By similar reasoning as for case d.

f: $\Rightarrow$-part. Assume $S \sqsubseteq_{abs\text{-}\frac{2}{3}bis} I$ and take P such that $S \sqsubseteq_{abs} P \sqsubseteq_{\frac{2}{3}bis} I$ with abs-bisimulation relation R_1 and $\frac{2}{3}$-bisimulation relation R_2, respectively. It is easy to verify that $R = R_1 ⨾ R_2$ is a $\frac{1}{3}$-bisimulation relation for $S \sqsubseteq_{\frac{1}{3}bis}$, total on $start_I$.

$\not\Leftarrow$-part. Let $S = a.b + a.c.d$ and $I = a.(b + c.e)$. Now $S \sqsubseteq_{\frac{1}{3}bis} I$, but $S \not\sqsubseteq_{abs\text{-}\frac{2}{3}bis} I$.

□

Proof of Proposition 9. Put $P' = lts(P)$ and $Q = rs(P')$. We decorate all components (except $\rightarrow$) of P' with a prime $'$. The following lemma is easily provable by induction on the structure of t:

Lemma 12. For $t, t' \in Tr$ and $A \in Acc(t')$: $start' \ni\xrightarrow{t} (t', A) \quad \Leftrightarrow \quad t = t'$

Now continuing the proof of Proposition 9, it is clear that $Act_Q = Act' = Act$. We prove for an arbitrary trace t that:

$$\begin{array}{lclcl} t \in Tr & \Leftrightarrow & t \in Tr' & \Leftrightarrow & t \in Tr_Q \\ R \in Ref(t) & \Leftrightarrow & R \in Ref'(t) & \Leftrightarrow & R \in Ref_Q(t), \end{array}$$

where, in the latter line, the components of the former line are assumed to hold. The rightmost equivalences are true by construction of Q ($Tr_Q = Tr'$ and $Ref_Q = Ref'$), so it suffices to show the leftmost equivalences only.

For arbitrary t we argue, for the upper-left equivalence:

$$\begin{array}{ll} t \in Tr & \\ \Leftrightarrow \exists\, A \bullet (t, A) \in State' & [\text{def } lts,\ Acc(t) \text{ nonempty}] \\ \Leftrightarrow \exists\, A, t' \bullet start' \ni\xrightarrow{t} (t', A) & [\text{lemma 12}] \\ \Leftrightarrow t \in Tr' & [\text{def } Tr'] \end{array}$$

And for the lower-left equivalence, tacitly using that $t \in Tr \wedge t \in Tr'$:

$$\begin{array}{ll} A \in Acc(t) & \\ \Leftrightarrow (t, A) \in State' & [\text{def } lts] \\ \Leftrightarrow \exists\, t', A' \bullet start' \ni\xrightarrow{t} (t', A') \wedge A = A' & [\text{lemma 12}] \\ \Leftrightarrow \exists\, t', A' \bullet start' \ni\xrightarrow{t} (t', A') \wedge initials(t, A') = A & [\text{def } lts, initials] \\ \Leftrightarrow A \in Acc'(t) & [\text{def } Acc'] \end{array}$$

□