# Action refinement as an implementation relation

Arend Rensink[1] and Roberto Gorrieri[2]

[1] Institut für Informatik, University of Hildesheim, Postfach 101363, D–31113 Hildesheim; email: rensink@informatik.uni-hildesheim.de

[2] Dipartimento di Scienze dell'Informazione, University of Bologna, Porta San Donato 5, I–40127 Bologna; email: gorrieri@cs.unibo.it

This work has been partially supported by the Vigoni exchange program and the HCM network EXPRESS ("Expressiveness of Languages for Concurrency").

**Abstract.** We propose a theory of process refinement which relates behavioural descriptions belonging to conceptually different abstraction levels, through a so-called *vertical implementation relation*. The theory is based on action refinement, which permits to relate abstract actions of the specification to concrete computations of the implementation; it is developed in the standard interleaving approach. A number of proof rules is shown to be sound for the particular vertical implementation relation (based on observation congruence) we study in this paper. We give an illustrative example.

**Appeared in: TAPSOFT '97: Theory and Practice of Software Development, M. Bidoit and M. Dauchet (Eds.), pp. 772–786**

## 1 Introduction

There is a long tradition in defining process refinement theories (cf. [9] for an overview), essentially based on the idea that, given two processes $S$ and $I$, $I$ is an implementation of $S$ if $I$ is more deterministic (equivalent) according to the chosen semantics. Still, both $S$ and $I$ belong conceptually to the same abstraction level, as the actions they perform belong to the same alphabet. In the development of software components, however, it is quite often required to compare systems belonging to different abstraction levels. To the best of our knowledge, the only theory that has been developed to this aim is the work on action refinement (e.g., [2, 3, 7, 10, 18, 19, 20]) and interface refinement [4].

Given a refinement function $r$ mapping abstract actions to concrete processes, the developed theories say that the implementation of a specification $S$ is given by the *syntactic substitution* of concrete processes $r(a)$ for actions $a$ in $S$ [2, 3, 11, 16] or by the *semantic substitution* of the model of concrete processes $r(a)$ for actions $a$ in the semantics of $S$ [7, 10, 12, 18]. The basic assumption of these theories is that there is *only one* possible implementation for a given specification; in other words, the action refinement function is used as a *prescriptive tool* to specify the only way abstract actions are to be implemented. Consequences of this are the following:

- The refinement function can be used as an operator of the language, as it defines also a function on processes. Hence, it becomes immediately relevant to investigate the so-called *congruence problem*: find an equivalence relation

such that, if two processes $S_1$ and $S_2$ are equivalent, then also $r(S_1)$ and $r(S_2)$ are equivalent. Dating back to [5], it is clear that it is necessary to move to *non-interleaving* semantics: the parallel execution of actions $a$ and $b$, denoted $a \,|||\, b$, is interleaving equivalent to their sequential simulation $a;b+b;a$; however, if we refine $a$ to the sequence $a_1;a_2$, then we obtain $a_1;a_2 \,|||\, b$ and $a_1;a_2;b+b;a_1;a_2$ which are not equivalent at all. Most of the work in action refinement has been devoted to this problem.

– Because of the strong relation to the syntactical structure of the specification $S$, the implementation $r(S)$ is rigidly defined. One of the typical constraints is that the possible causal relation between two abstract actions is preserved among *all* the actions of the two implementing processes. For instance, if $S = a;b$ and $r(a) = a_1;a_2$, then the only possible implementation is $r(S) = a_1;a_2;b$. As pointed out in [14], this can be a serious drawback, because in general a causal relation at an abstract level *could* be partially forgotten at the concrete one: if only $a_1$ is to be considered a cause for $b$, then $a_1;(a_2 \,|||\, b)$ implements $a;b$ (via $r$). Some investigations of less rigid forms of action refinement can be found in [8, 13, 21]. Still, in all these approaches, specification and refinement function completely determining the implementation.

Our research starts by removing the basic assumption: *more than one* implementation is possible for a given specification. This seems quite natural, even if the implementation of the abstract action is completely specified via $r$; for instance, if an abstract action represents a communication, the way the actual implementing protocol is defined should not be relevant at the high level of the specification. Considering the example above, $a \,|||\, b$ is implemented as $a_1;a_2 \,|||\, b$ (via $r$) in the traditional approach, but we also admit the more sequential process $a_1;a_2;b+b;a_1;a_2$ as a possible implementation. Similarly, we consider $a_1;a_2 \,|||\, b$ a legal implementation for $a;b+b;a$ (via $r$).

As a consequence, the congruence problem simply disappears: since one single specification may admit non-equivalent implementations, *a fortiori* implementations of two equivalent specifications need be equivalent themselves. Furthermore, the syntactic structure needs not to be preserved rigidly.

We advocate the use of *vertical implementation relations* (up to a refinement function), a concept first proposed in [17], as a means to relate processes belonging to conceptually different abstraction levels. They are built on top of an existing *horizontal* implementation relation, called its *basis*, such as those mentioned above, but in addition use the refinement function to set a correspondence between abstract actions and concrete computations. After introductory definitions (Sect. 2), the core of the paper (Sect. 3) discusses a set of properties any vertical implementation relation $\sqsubseteq^r$ (where $r$ is the refinement function considered) should satisfy. They can be divided into two main groups. The first group states the interplay between $\sqsubseteq^r$ and its chosen basis $\leq$: in particular, $\sqsubseteq^{id}$ (vertical implementation under the identity function) collapses to $\leq$, and $\sqsubseteq^r$ and $\leq$ compose, meaning that $\leq \circ \sqsubseteq^r \circ \leq \;= \sqsubseteq^r$. The second group defines a set of congruence-like properties; e.g., if $S_i \sqsubseteq^r I_i$ for $i = 1, 2$, then $S_1 + S_2 \sqsubseteq^r I_1 + I_2$. In

Sect. 4 we then propose a specific vertical implementation relation $\lesssim^r$, with the following main features: it is defined in the standard interleaving approach, its basis is observation congruence $\simeq$ (cf. [15]), and it enjoys all the proof rules for $\sqsubseteq^r$. Finally, in Sect. 5 we apply the resulting theory to an example taken from [4]. Because of space limitations, proofs have been omitted from this paper.

*Evaluation.* The approach to action refinement proposed in this paper is quite new, in the following respects:

- We allow a given abstract specification to have different, incomparable implementations under a given, fixed refinement function. This immediately implies that refinement cannot be treated as an operator; hence the standard congruence problem of traditional action refinement disappears.
- We integrate action refinement with interleaving semantics. The only remotely similar work we are aware of is [6], which establishes restrictions under which interleaving models are still compositional with respect to traditional refinement; and [12], which considers a different type of action refinement, for which interleaving semantics is already compositional.
- We directly compare systems on different levels of abstraction, using the concept of *vertical implementation* that extends the standard notion of "horizontal" implementation relation.
- We give algebraic proof rules for vertical implementation. The only comparable concept in traditional action refinement seems to be its treatment as syntactic substitution, studied by Aceto and Hennessy in [2, 3] and compared with semantic refinement in [11].
- We allow vertical implementation to be *collapsed* to the well-known observational congruence relation, by hiding all the actions that were refined. This is reminiscent of *interface refinement* as in [4]; it makes it possible to mix action refinement with established methods for "horizontal" implementation.

Many of the basic ideas behind the approach of this paper were already present in [12, 17], but the technical material, including the algebraic proof rules and the notion of vertical bisimulation, appear here for the first time.

## 2 Definitions

We assume a universe of action names $\mathbf{U}$, ranged over by $a, b, c$, and an invisible action $\tau \notin \mathbf{U}$. Subsets of $\mathbf{U}$ are denoted $\mathbf{A}, A, \mathbf{C}, C$ (for *abstract* and *concrete* actions, respectively). We denote $A_\tau = A \cup \{\tau\}$ for any $A \subseteq \mathbf{U}$. $\mathbf{U}_\tau$ is ranged over by $\alpha, \beta, \gamma$. In addition we use a set of *process names* $\mathbf{X}$. We define a family of languages $\mathbf{L_A}$, ranged over by $t, u, v, S, I$; $\mathbf{A} \subseteq \mathbf{U}$ is the set of actions that may be used within terms.

$$t ::= \mathbf{0} \mid \mathbf{1} \mid \alpha \mid t;t \mid t+t \mid t \parallel_A t \mid t[\phi] \mid t/A \mid x \mid \mu x.\, t \ .$$

Here, $\alpha \in \mathbf{A}_\tau$, $A \subseteq \mathbf{A}$, $\phi\colon \mathbf{A} \to \mathbf{A}$ and $x \in \mathbf{X}$. Renaming functions $\phi$ are extended when necessary with the mapping $\tau \mapsto \tau$. In addition, we use $t \parallel\!\!\parallel u = t \parallel_\varnothing u$ to

**Table 1.** Structural operational semantics

$$\frac{}{\mathbf{1}\checkmark} \qquad \frac{t\checkmark \quad u\checkmark}{(t+u)\checkmark} \qquad \frac{t\checkmark \quad u\checkmark}{(t;u)\checkmark} \qquad \frac{t\checkmark \quad u\checkmark}{(t \parallel_A u)\checkmark} \qquad \frac{t\checkmark}{(t/A)\checkmark} \qquad \frac{t\checkmark}{t[\phi]\checkmark} \qquad \frac{t\checkmark}{(\mu x.\, t)\checkmark}$$

$$\frac{}{\alpha \xrightarrow{\alpha} \mathbf{1}} \qquad \frac{t \xrightarrow{\alpha} t'}{t+u \xrightarrow{\alpha} t'} \qquad \frac{u \xrightarrow{\alpha} u'}{t+u \xrightarrow{\alpha} u'} \qquad \frac{t \xrightarrow{\alpha} t'}{t;u \xrightarrow{\alpha} t';u} \qquad \frac{t\checkmark \quad u \xrightarrow{\alpha} u'}{t;u \xrightarrow{\alpha} u'}$$

$$\frac{t \xrightarrow{\alpha} t' \quad \alpha \notin A}{t \parallel_A u \xrightarrow{\alpha} t' \parallel_A u} \qquad \frac{u \xrightarrow{\alpha} u' \quad \alpha \notin A}{t \parallel_A u \xrightarrow{\alpha} t \parallel_A u'} \qquad \frac{t \xrightarrow{\alpha} t' \quad u \xrightarrow{\alpha} u' \quad \alpha \in A}{t \parallel_A u \xrightarrow{\alpha} t' \parallel_A u'}$$

$$\frac{t \xrightarrow{\alpha} t' \quad \alpha \notin A}{t/A \xrightarrow{\alpha} t'/A} \qquad \frac{t \xrightarrow{\alpha} t' \quad \alpha \in A}{t/A \xrightarrow{\tau} t'/A} \qquad \frac{t \xrightarrow{\alpha} t'}{t[\phi] \xrightarrow{\phi(\alpha)} t'[\phi]} \qquad \frac{t \xrightarrow{\alpha} t'}{\mu x.\, t \xrightarrow{\alpha} t'[\mu x.\, t/x]}$$

denote synchronisation-less parallelism. We also use $\mathcal{A}(t)$ to denote the set of actions syntactically occurring in $t$ (taking care to define this appropriately for recursive terms.) For the treatment of process names and recursion, we rely on the standard notion of *guardedness*: all recursive terms are assumed to be guarded. A stronger criterion that we will need in the course of the paper is *visible* guardedness, which holds if all process names are guarded by a visible action, and no process name or recursion occurs in the context of hiding.

The language $\mathbf{L_A}$ has an operational semantics expressed by a transition relation $\rightarrow \subseteq \mathbf{L_A} \times \mathbf{A}_\tau \times \mathbf{L_A}$ and a termination predicate $\checkmark \subseteq \mathbf{L_A}$: see Table 1. There are two slightly nonstandard aspects: the semantic rule for recursion reflects the fact that we assume guardedness; and following Aceto and Hennessy [1], a choice is terminated only if *both* operands are terminated. The latter has the following consequence:

**Proposition 1.** *For all $t \in \mathbf{L_A}$, if $t\checkmark$ then $\nexists \alpha \in \mathbf{A}_\tau.\ t \xrightarrow{\alpha}$.*

This plays a crucial role in our definition of vertical bisimulation. The basic, one-step transitions are extended to $\tau$-abstracting transitions in the usual fashion:

$$t \xRightarrow{\alpha_1 \cdots \alpha_n} u \ :\Leftrightarrow\ t \xrightarrow{\tau}{}^* \xrightarrow{\alpha_1} \xrightarrow{\tau}{}^* \cdots \xrightarrow{\tau}{}^* \xrightarrow{\alpha_n} \xrightarrow{\tau}{}^* u$$

An important property of visible guardedness is the following:

**Proposition 2.** *If $t$ is visibly guarded, then for all $\sigma \in \mathbf{A}^*$, there is only a finite number of $t'$ such that $t \xRightarrow{\sigma} t'$.*

In general, a transition system is a tuple $T = \langle L, S, \rightarrow, \checkmark, q \rangle$ where $\rightarrow \subseteq S \times L \times S$ is the transition relation, $q \in S$ is the initial state and $\checkmark \subseteq S$ a *termination predicate*, which is such that $s\checkmark$ implies $\nexists \ell \in L.\ s \xrightarrow{\ell}$. We write $s \xrightarrow{a} s'$ for $(s, a, s') \in \rightarrow$ and $s\checkmark$ for $s \in \checkmark$. We denote the components of $T$ by $L_T$, $S_T$ etc., dropping the index whenever this does not give rise to confusion. Obviously, for every term $t \in \mathbf{L_A}$, the operational semantics gives rise to a transition system with termination $\langle \mathbf{A}_\tau, \mathbf{L_A}, \rightarrow, \checkmark, t \rangle$ where $\rightarrow$ and $\checkmark$ are the smallest predicates satisfying the rules in Table 1.

A widely accepted $\tau$-abstracting interleaving equivalence relation is *observation congruence*; see [15], in our case extended to take termination into account; see also [1]. As the name suggests, the resulting relation is a congruence over $\mathbf{L}$. The definition relies on a function $\hat{\cdot}\colon \mathbf{U}_\tau \to \mathbf{U}^*$ such that $\hat{\tau} = \varepsilon$ and $\hat{a} = a$ for all $a \in \mathbf{U}$.

**Definition 3.** Let $T, U$ be transition systems with termination. A weak bisimulation relation between $T$ and $U$ is a binary relation $\rho \subseteq S_T \times S_U$ such that for all $(s_T, s_U) \in \rho$

1. If $s_T \xrightarrow{\alpha} s'_T$ then $\exists s_U \stackrel{\hat{\alpha}}{\Longrightarrow} s'_U$ such that $(s'_T, s'_U) \in \rho$;
2. If $s_U \xrightarrow{\alpha} s'_U$ then $\exists s_T \stackrel{\hat{\alpha}}{\Longrightarrow} s'_T$ such that $(s'_T, s'_U) \in \rho$;
3. If $s_T\checkmark$ then $\exists s_U \stackrel{\varepsilon}{\Longrightarrow} s'_U$ such that $s'_U\checkmark$.
4. If $s_U\checkmark$ then $\exists s_T \stackrel{\varepsilon}{\Longrightarrow} s'_T$ such that $s'_T\checkmark$.

$T$ and $U$ are called *observation equivalent*, denoted $T \approx U$, if there is a bisimulation relation $\rho$ such that $(q_T, q_U) \in \rho$, and *observation congruent*, denoted $T \simeq U$, if in addition

5. If $q_T \xrightarrow{\tau} s_T$ then $\exists q_U \stackrel{\tau}{\Longrightarrow} s_U$ such that $(s_T, s_U) \in \rho$;
6. If $q_U \xrightarrow{\tau} s_U$ then $\exists q_T \stackrel{\tau}{\Longrightarrow} s_T$ such that $(s_T, s_U) \in \rho$.

*Refinement Functions.* A refinement function maps abstract actions to concrete processes, where the notions of *abstract* and *concrete* are accompanied by a change of alphabet. If $\mathbf{A}$ is the set of abstract actions and $\mathbf{C}$ that of concrete actions, then a refinement function is of the form $r\colon \mathbf{A} \to \mathbf{L}_\mathbf{C}$, with *domain* $\operatorname{dom} r = \mathbf{A}$. To (informally) preserve the *atomicity* of abstract actions, the images of $r$ are constrained to be

- *non-empty*, i.e., $\neg r(a)\checkmark$ for all $a \in \operatorname{dom} r$,
- *eventually terminating*, i.e., $t \stackrel{\sigma}{\Longrightarrow} t'\checkmark$ for any term $t$ reachable from $r(a)$,
- *visible*, i.e., $t \not\xrightarrow{\tau}$ for any term $t$ reachable from $r(a)$.

The resulting fragment of $\mathbf{L}$ is reasonably general, and (as far as we know) includes all refinement functions that we know of as having been proposed in practical examples. For instance, all renaming functions can be regarded as (particular instances) of refinement functions. Some refinement functions actually contain a degree of *confusion*, in the sense that the alphabets of refinements of different abstract actions overlap. Part of the theory developed in this paper relies on the absence of such confusion; this is achieved by imposing further restrictions on the refinement functions.

**Definition 4 (refinement functions).** Let $r\colon \mathbf{A} \to \mathbf{L}_\mathbf{C}$ be arbitrary.

1. $r$ is called *allowable* if for all $a \in \mathbf{A}$, $r(a)$ is non-empty, eventually terminating and visible. The class of allowable refinement functions is denoted $\mathbf{R}_{\mathbf{A},\mathbf{C}}$.
2. $r$ is called *initial-distinct* if for all $a, b \in \mathbf{A}$, $r(a) \xrightarrow{c}$ together with $r(b) \stackrel{\sigma}{\Longrightarrow} t \xrightarrow{c}$ implies $a = b$ and $\sigma = \varepsilon$ (hence $t = r(b)$).
3. $r$ is called *distinct* on $A$ if for all $a, b \in A$, $r(a) \stackrel{\sigma_a}{\Longrightarrow} t_a \xrightarrow{c} t'_a$ together with $r(b) \stackrel{\sigma_b}{\Longrightarrow} t_b \xrightarrow{c} t'_b$ implies $t_a = t_b$ and $t'_a = t'_b$, and if, furthermore, $\sigma_a = \varepsilon$, then $a = b$ and $\sigma_b = \varepsilon$.

These constraints are semantic-based, but it is not difficult to single out syntactic restrictions on terms that ensure them. From now on, we only consider allowable refinement functions. With $\tilde{r}\colon \mathbf{A} \to \mathbf{2^C}$ we denote the function $a \mapsto \mathcal{A}(r(a))$. This is extended pointwise (under overloading of notation) to $\tilde{r}\colon \mathbf{2^A} \to \mathbf{2^C}$.

Another possible source of confusion contained in $r$ consists of an overlap between the actions used in the refinements of a certain set $A \subseteq \mathbf{A}$ and the refinements of $\overline{A} = \mathbf{A} \setminus A$. If such an overlap does not exist, we say that $r$ *preserves* $A$, which is formally defined as follows:

**Definition 5.** $A \subseteq \operatorname{dom} r$ is *preserved* by $r$ if $\tilde{r}(A) \cap \tilde{r}(\overline{A}) = \varnothing$.

Furthermore, we distinguish the *active domain* $\operatorname{adom} r$ and the *identity domain* $\operatorname{idom} r$ of a refinement function $r$, defined as follows:

$$\operatorname{adom} r = \textstyle\bigcup_{r(a)\neq a} (\{a\} \cup (\tilde{r}(a) \cap \operatorname{dom} r))$$
$$\operatorname{idom} r = \operatorname{dom} r \setminus \operatorname{adom} r$$

Hence the active domain is a subset of the domain, consisting of two types of actions: those that are not mapped onto themselves, and those that are used in the image of any action different from themselves. The identity domain, on the other hand, consists only of (but not necessarily of all) actions on which the refinement is the identity function. (Note that $\operatorname{adom} r$ and $\operatorname{idom} r$ are always preserved by $r$, which would *not* have been the case if we had taken the more straightforward definition $\operatorname{adom} r = \{a \mid r(a) \neq a\}$.) We use $id\colon \mathbf{A} \to \mathbf{L_A}$ to denote the identity refinement function on $\mathbf{A}$ (hence $\operatorname{adom} id = \varnothing$). In addition, we use the following constructions on refinement functions:

$$r\backslash A\colon a \mapsto \begin{cases} a & \text{if } a \in A \\ r(a) & \text{otherwise} \end{cases} \qquad r[\phi]\colon a \mapsto r(a)[\phi] \qquad r \circ \phi\colon a \mapsto r(\phi(a))$$

## 3 Proof rules for vertical implementation

We now come to the concept of a *vertical implementation relation* $\sqsubseteq^r$, parametrised w.r.t. a refinement function $r$. $t \sqsubseteq^r u$ is intended to mean that $t$ is an abstract system and $u$ one of its possible implementations, where the correspondence between actions of the former and computations of the latter is set via the refinement function $r$. We regard vertical implementation in combination with a more standard, "flat" or "horizontal" implementation relation (i.e., relating systems at the same abstraction level) such as those studied in, e.g., [9]. This flat implementation relation, sometimes referred to as the *basis* of $\sqsubseteq^r$, is denoted $\leq$. In the following sections, we will actually instantiate $\leq$ to observation congruence $\simeq$.

In order to deal with recursion, we also have to consider *open terms*, i.e., terms with free process names. Let $fn(t)$ denote the free process names in $t$. Unfortunately, we cannot rely on the standard technique to extend relations to open terms, since $x \in fn(t)$ has a different interpretation from $x \in fn(u)$; viz., the latter stands for an *implementation* of the former. Therefore, we require a

**Table 2.** Proof rules for vertical implementation

$$\frac{fn(t) = \{\mathbf{x}\}}{\mathbf{x} \sqsubseteq^{id} \mathbf{x} \vdash t \sqsubseteq^{id} t}\mathrm{R}_1 \qquad \frac{\mathbf{x} \sqsubseteq^{id} \mathbf{x} \vdash t \sqsubseteq^{id} u}{t \leq u}\mathrm{R}_2 \qquad \frac{t \leq t' \quad \Gamma \vdash t' \sqsubseteq^r u' \quad u' \leq u}{\Gamma \vdash t \sqsubseteq^r u}\mathrm{R}_3$$

$$\frac{}{\vdash \mathbf{0} \sqsubseteq^r \mathbf{0}}\mathrm{R}_4 \qquad \frac{}{\vdash \mathbf{1} \sqsubseteq^r \mathbf{1}}\mathrm{R}_5 \qquad \frac{}{\vdash \alpha \sqsubseteq^r r(\alpha)}\mathrm{R}_6 \qquad \frac{\Gamma \vdash t_1 \sqsubseteq^r u_1,\, t_2 \sqsubseteq^r u_2}{\Gamma \vdash t_1 + t_2 \sqsubseteq^r u_1 + u_2}\mathrm{R}_7$$

$$\frac{\Gamma \vdash t_1 \sqsubseteq^r u_1,\, t_2 \sqsubseteq^r u_2}{\Gamma \vdash t_1; t_2 \sqsubseteq^r u_1; u_2}\mathrm{R}_8 \qquad \frac{\Gamma \vdash t \sqsubseteq^r u \quad r \text{ preserves } A}{\Gamma \vdash t/A \sqsubseteq^{r \setminus A} u/\tilde{r}(A)}\mathrm{R}_9$$

$$\frac{\Gamma \vdash t \sqsubseteq^r u \quad \mathrm{adom}\, r \subseteq \mathrm{idom}\, \phi}{\Gamma \vdash t[\phi] \sqsubseteq^r u[\phi]}\mathrm{R}_{10} \qquad \frac{\Gamma \vdash t \sqsubseteq^r u \quad \phi \text{ injective}}{\Gamma \vdash t[\phi] \sqsubseteq^{r[\psi] \circ \phi^{-1}} u[\psi]}\mathrm{R}_{11}$$

$$\frac{\Gamma \vdash t_1 \sqsubseteq^r u_1,\, t_2 \sqsubseteq^r u_2 \quad r \text{ preserves and is distinct on } A}{\Gamma \vdash t_1 \parallel_A t_2 \sqsubseteq^r u_1 \parallel_{\tilde{r}(A)} u_2}\mathrm{R}_{12}$$

$$\frac{}{x \sqsubseteq^r x \vdash x \sqsubseteq^r x}\mathrm{R}_{13} \qquad \frac{\Gamma \vdash t \sqsubseteq^r u}{\Gamma, \Delta \vdash t \sqsubseteq^r u}\mathrm{R}_{14} \qquad \frac{\Gamma, x \sqsubseteq^{r'} x \vdash t \sqsubseteq^r u \quad \Gamma \vdash t' \sqsubseteq^{r'} u'}{\Gamma \vdash t[t'/x] \sqsubseteq^r u[u'/x]}\mathrm{R}_{15}$$

$$\frac{\Gamma, x \sqsubseteq^r x \vdash t \sqsubseteq^r u}{\Gamma \vdash \mu x.\, t \sqsubseteq^r \mu x.\, u}\mathrm{R}_{16}$$

list of *assumptions* about how the free process names are to be interpreted, of the form $\Gamma = x_1 \sqsubseteq^{r_1} x_1, \ldots, x_n \sqsubseteq^{r_n} x_n$ where for all $i$, $x_i$ is a process name and $r_i$ a refinement function, and $x_i \sqsubseteq^{r_i} x_i$ expresses that $x_i$ occurring in $u$ is assumed to be an $r_i$-implementation of $x_i$ occurring in $t$. (We sometimes write $\Gamma = \mathbf{x} \sqsubseteq^{\mathbf{r}} \mathbf{x}$ where $\mathbf{x} = x_1 \cdots x_n$ and $\mathbf{r} = r_1 \cdots r_n$ are *vectors* of variables and refinement functions, respectively.) We then write $\Gamma \vdash t \sqsubseteq^r u$ to indicate that $t \sqsubseteq^r u$ holds whenever appropriate closed terms are substituted for the $x_i$; in other words, $t[\mathbf{t}/\mathbf{x}] \sqsubseteq^r u[\mathbf{u}/\mathbf{x}]$ whenever $\forall i.\ t_i \sqsubseteq^{r_i} u_i$. If $\mathrm{dom}\,\Gamma = \varnothing$, we write $\vdash t \sqsubseteq^u r$ or simply $t \sqsubseteq^r u$.

A number of *proof rules* for $\sqsubseteq^r$ are given in Table 2. We first discuss the case for closed terms; i.e., we assume $\Gamma = \varnothing$ and consider $\mathrm{R}_1$–$\mathrm{R}_{12}$ only.

The first group of properties, consisting of rules $\mathrm{R}_1$–$\mathrm{R}_3$, expresses the basic assumption of working "modulo" the basis $\leq$. Rule $\mathrm{R}_1$ states that every term implements itself as long as no refinement takes place; rule $\mathrm{R}_2$ says that $\sqsubseteq^{id}$, where no actual refinement takes place, implies horizontal implementation; Rule $\mathrm{R}_3$ explains the interplay between horizontal and vertical implementation. Note that, as a consequence, we also have that $t \leq u$ implies $t \sqsubseteq^{id} u$; hence $\leq$ and $\sqsubseteq^{id}$ in fact coincide. Moreover, Rules $\mathrm{R}_1$–$\mathrm{R}_3$ together imply that $\leq$ is a pre-order, which indeed is the standard requirement for (flat) implementation relations.

$\mathrm{R}_4$–$\mathrm{R}_{12}$ essentially express congruence of vertical implementation with respect to the constants and operators of our language. For instance, if the refinement functions in these rules are set to $id$, then these rules collapse to the standard pre-congruence properties of $\leq$. (In other words, $\leq$ needs to be at least

a pre-congruence.)

$R_6$ is the core of the relationship between the refinement function $r$ and the vertical implementation relation. It expresses the basic expectation that $r(a)$ should be an implementation for $a$. $R_7$ and $R_8$ are straightforward congruence rules. $R_9$ is slightly more surprising in that the refinement function "loses" some of its active domain, namely those actions that are hidden. An interesting special case is when *all* actively refined actions are hidden, in which case the vertical implementation collapses to its basis; i.e., if $t \sqsubseteq^r u$ then $t/\operatorname{adom} r \leq u/\tilde{r}(\operatorname{adom} r)$.

Renaming and refinement are similar concepts; indeed there is some interference between the two, due to which no general congruence rule for renaming can be formulated. Instead, we have "standard" congruence ($R_{10}$) if the refinement and renaming functions do not interfere, and another rule ($R_{11}$) which treats renaming as part of the refinement and only works for injective renamings. In $R_{12}$, finally, the synchronisation set $A$ of the specification is refined in the implementation; moreover, there is a restriction on the refinement function, which will be discussed below in more detail.

There are some side conditions in Table 2 whose rationale is not immediately obvious. In particular, the refinement function is constrained to be $A$-preserving in the rule for hiding ($R_9$), and distinct and preserving in the rule for parallel composition ($R_{12}$). We give two examples illustrating what goes wrong if these side conditions are not met. We assume that $\leq$ preserves deadlock freedom, i.e., if $t$ is deadlock-free and $t \leq u$ then $u$ is deadlock-free.

*Example 1.* Assume $\mathbf{A} = \mathbf{C} = \{a, b\}$ and let $r\colon a \mapsto a; b,\ b \mapsto b$. Then the rules of Table 2 allow the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{(R_6)}{a \sqsubseteq^r a; b} \quad \dfrac{(R_6)}{b \sqsubseteq^r b}\ (R_8)}{a; b \sqsubseteq^r a; b; b}\ (R_9)}{(a; b)/a \sqsubseteq^{id} (a; b; b)/a, b}\ (R_2)}{(a; b)/a \leq (a; b; b)/a, b}$$

However, $(a; b)/a$ gives rise to a non-deadlocking term when substituted for $x$ in $x \parallel_b b$, whereas $(a; b; b)/a, b$ does not. This contradicts the requirement that $\leq$ preserves deadlock freedom.

The above problem is caused by the application of $R_9$: we hid $a$ in the specification and the alphabet of its refinement, $\tilde{r}(a)$, in the implementation. The latter includes $b \in \tilde{r}(a)$, which, however, also occurs independently of $a$. In other words, $\{a\}$ is not *preserved* by $r$; hence the side condition of $R_9$ is not met.

The next example shows what goes wrong if the distinctness condition in Rule $R_{12}$ is not met: confusion, in the sense discussed in the justification of Def. 4, may arise if the refinements of two different actions start with the same concrete action.

*Example 2.* Let $r$ be a refinement function with active part $a \mapsto c;a$ and $b \mapsto c;b$. The rules of Table 2 then allow to derive $((a+d)\|_{a,b}(b+d))/a,b \leq ((c;a+d)\|_{a,b,c}(c;b+d))/a,b,c$. The left hand term contains no deadlock, whereas the right hand term has a $\tau$-transition to the deadlocked state $(\mathbf{1};b\,\|_{b,c,d}\,\mathbf{1};d)/b,c,d$.

Now we turn to open terms and non-empty assumption lists. The intuition behind the proof rules discussed so far is not changed essentially. Rules $R_{13}$–$R_{15}$ reflect the intention discussed at the beginning of this section. Rule $R_{16}$ is the usual congruence rule for recursion, adapted to take the assumption list into account. Moreover, this rule is restricted to *visibly guarded recursion*. In contrast to the restrictions discussed above, this is not because the general version is known to be unsound (in fact, we conjecture that it is sound) but because we have been unable to prove it. The difficulties stem from the fact that the standard proof technique of *up-to bisimulation* (cf. [15]) seems inapplicable.

## 4 Vertical bisimulation

We now come to the definition of an actual vertical implementation relation that satisfies the derivation rules of Table 2. We build on the principles of observation congruence. (However, the basic framework in no way depends on this choice, and we feel that any of the $\tau$-abstracting relations studied in, e.g., [9] can, in principle, be used as a basis for vertical implementation.)

Observation congruence is defined using a binary relation that connects states of the specification with states of the implementation. In the case of *vertical* bisimulation, we also have to take into account that in any given state of the implementation, there may be associated refined actions whose execution has not yet terminated. These will be collected in a multiset of *residual* or *pending refinements* that is added as a third component to the bisimulation relation. To be precise, an $r$-residual set is a multiset of non-terminated terms $t$ such that $r(a) \stackrel{\sigma}{\Longrightarrow} t$ for some $a \in \operatorname{dom} r$ and $\sigma \in \mathbf{C}^+$. It is formally represented by a function $R \in [\mathbf{L_C} \to \mathbb{N}]$. We will denote $t \in R$ if $R(t) > 0$. The operational behaviour of a multiset corresponds to the synchronisation-free parallel composition of its elements:

$$R \xrightarrow{\alpha} R' \;:\Leftrightarrow\; \exists t \in R.\, \exists t \xrightarrow{\alpha} t'.\, R' = (R \ominus [t]) \oplus [t']$$

We use the following constructions on residual sets:

$$0\colon u \mapsto 0 \qquad [t]\colon u \mapsto \begin{cases} 1 & \text{if } u = t \text{ and } \neg t\checkmark \\ 0 & \text{otherwise} \end{cases}$$

$$R_1 \oplus R_2\colon u \mapsto R_1(u) + R_2(u) \qquad R_1 \ominus R_2\colon u \mapsto max(R_1(u) - R_2(u), 0)$$

$$R \restriction A\colon u \mapsto \begin{cases} R(u) & \text{if } \mathcal{A}(u) \subseteq A \\ 0 & \text{otherwise.} \end{cases}$$

Note the fact that terminated terms do not contribute to the residual set. We now present our proposal for relating a *specification* $T$ with an *implementation* $U$, where abstract actions of the specification are matched by computations of their refinements.

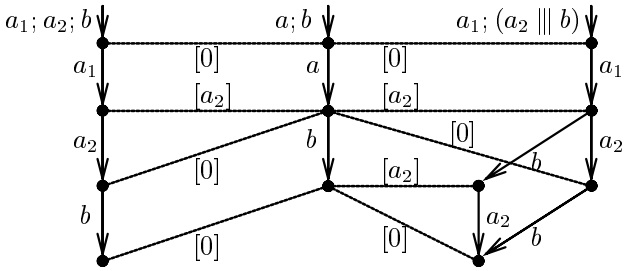
**Definition 6.** Let $T, U$ be transition systems with $L_T = \mathbf{A}_\tau$ and $L_U = \mathbf{C}_\tau$, and let $r \in \mathbf{R}_{\mathbf{A},\mathbf{C}}$ be a refinement function. A *vertical bisimulation relation up to* $r$ is a set $\rho \subseteq S_T \times [\mathbf{L_C} \to \mathbb{N}] \times S_U$ such that for all $\langle s_T, R, s_U \rangle \in \rho$, $R$ is an $r$-residual set and the following properties hold:

1. If $s_T \xrightarrow{\alpha} s'_T$, $R = 0$ and $r(\alpha) \stackrel{\sigma}{\Longrightarrow} v\checkmark$ for $\sigma \in \mathbf{C}^*$ then $\exists s_U \stackrel{\sigma}{\Longrightarrow} s'_U$ such that $\langle s'_T, 0, s'_U \rangle \in \rho$.
2. If $s_U \xrightarrow{\gamma} s'_U$ then either of the following holds:
   (a) $\exists \alpha.\, \exists s_T \stackrel{\hat{\alpha}}{\Longrightarrow} s'_T$ and $\exists r(\alpha) \xrightarrow{\gamma} v$ such that $\langle s'_T, R \oplus [v], s'_U \rangle \in \rho$.
   (b) $\exists s_T \stackrel{\varepsilon}{\Longrightarrow} s'_T$ and $\exists R \xrightarrow{\gamma} R'$ such that $\langle s'_T, R', s'_U \rangle \in \rho$.
3. If $R \xrightarrow{\gamma} R'$ then $\exists s_U \stackrel{\gamma}{\Longrightarrow} s'_U$ such that $\langle s_T, R', s'_U \rangle \in \rho$.
4. If $s_T\checkmark$ and $R = 0$ then $\exists s_U \stackrel{\varepsilon}{\Longrightarrow} s'_U$ such that $s'_U\checkmark$.
5. If $s_U\checkmark$ then $R = 0$ and $\exists s_T \stackrel{\varepsilon}{\Longrightarrow} s'_T$ such that $s'_T\checkmark$.

$T$ and $U$ are *vertically bisimilar up to* $r$, denoted $T \lesssim^r U$, if there is a vertical bisimulation relation $\rho$ with $\langle q_T, 0, q_U \rangle \in \rho$ and

6. If $q_T \xrightarrow{\tau} s_T$ then $\exists q_U \stackrel{\tau}{\Longrightarrow} s_U$ such that $\langle s_T, 0, s_U \rangle \in \rho$;
7. If $q_U \xrightarrow{\tau} s_U$ then $\exists q_T \stackrel{\tau}{\Longrightarrow} s_T$ such that $\langle s_T, 0, s_U \rangle \in \rho$.

Let $r$: $a_1; a_2$. The following shows two examples of vertical bisimulation relations:




The first item of Def. 6 is quite natural: if no residual is active and the specification can do an action $\alpha$, then the implementation can match any terminated trace of the refinement of $\alpha$. (It turns out to be too strong to require that a *single step* of the refinement of $\alpha$ can be matched by the implementation.) The second item considers the case where the moves of the implementation are to be justified. There are two possible justifications: either the low-level action "opens" a new refinement, in which case the specification must be able to do the corresponding abstract action, and the new residual is added to the residual set; or the low-level action continues one of the pending refinements, in which case the specification does not take part except for a possible invisible move. The third item is crucial: any move of the pending refinement set must be matched by the implementation, without the specification moving at all. This implies that pending refinements can be "worked off" in any possible order by the implementation. This can be construed as an operational formulation of *atomicity*: that which is started can always be finished.

Directly from Def. 6, it follows that vertical bisimilarity up to *id* equals observation congruence. Furthermore, the rules in Table 2 are sound for $\lesssim^r$, To formalise this, we write $\mathbf{x} \lesssim^{\mathbf{r}} \mathbf{x} \vDash t \lesssim^r t$ if $\forall i.\, t_i \lesssim^{r_i} u_i$ implies $t[\mathbf{t}/\mathbf{x}] \lesssim^r u[\mathbf{u}/\mathbf{x}]$.

**Theorem 7.** $\_ \vDash \_ \lesssim^r \_$ *satisfies all the rules in Table 2.*

Note that, although Table 2 gives no recipe for deriving implementations from specifications, in many cases, one particular implementation can be obtained through the *syntactic substitution* of all abstract actions by their refinements.

*Abstraction.* In order to strengthen the intuition behind vertical bisimulation, we now show that it can in fact be characterised as a combination of (horizontal) observation congruence and *abstraction*. The abstraction of a transition system $U$ up to a given refinement function consists of "guessing" where the transitions of $U$ originate from, i.e., which abstract action they refine.

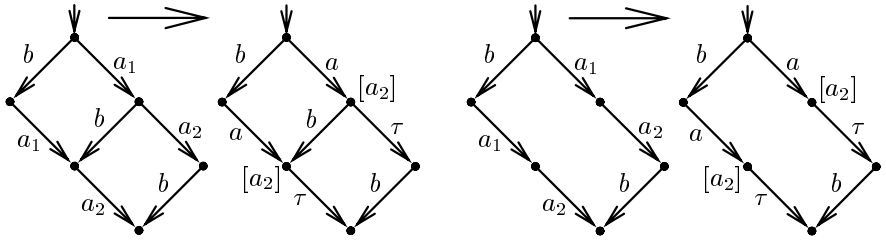
**Definition 8 (abstraction).** Let $U$ be a transition system with $L_U = \mathbf{C}_\tau$, and $r \in \mathbf{R}_{\mathbf{A},\mathbf{C}}$ a refinement function. An *r-abstraction of* $U$ is a transition system $\langle \mathbf{A}_\tau, S, \rightarrow, \checkmark_U \times \{0\}, (q_U, 0)\rangle$, where $(q_U, 0) \in S \subseteq S_U \times [\mathbf{L_C} \to \mathbb{N}]$ and

$$\begin{aligned} \rightarrow \subseteq\ & \{((s,R), \alpha, (s', R \oplus [v])) \mid s \xrightarrow{\gamma} s', r(\alpha) \xrightarrow{\gamma} v\} \\ & \cup \{((s,R), \tau, (s', R')) \mid s \xrightarrow{\gamma} s', R \xrightarrow{\gamma} R'\} \end{aligned}$$

Moreover, the following conditions are required to hold for all $(s, R) \in S$:

1. if $(s,R) \xrightarrow{\alpha} (s', R')$, $R = 0$ and $r(\alpha) \stackrel{\sigma}{\Longrightarrow} v\checkmark$ then $\exists s \stackrel{\sigma}{\Longrightarrow} s''$ s.t. $(s,R) \stackrel{\alpha}{\Longrightarrow} (s'', 0) \approx (s', R')$;
2. if $s \xrightarrow{\gamma} s'$ then either $\exists r(\alpha) \xrightarrow{\gamma} v.(s,R) \xrightarrow{\alpha} (s', R \oplus [v])$ or $\exists R \xrightarrow{\gamma} R'.(s,R) \xrightarrow{\tau} (s', R')$;
3. if $R \xrightarrow{\gamma} R'$ then $\exists s \stackrel{\gamma}{\Longrightarrow} s'$ such that $(s,R) \stackrel{\varepsilon}{\Longrightarrow} (s', R') \approx (s, R)$.

There is a clear correspondence of the conditions above to the simulation properties 1–3 of Def. 6. Two easy examples of abstraction, using the function $r$ with $r\colon a \mapsto a_1; a_2$, are given by the following transition systems (where we only show nonempty residual sets):




**Theorem 9.** $T \lesssim^r U$ *if there exists an r-abstraction* $V$ *of* $U$ *such that* $T \simeq V$.

Although the principle of abstraction strengthens the intuition behind vertical bisimulation, it does not yet offer an easier method of checking vertical bisimulation: the abstraction of a transition system is not always defined, may not be unique when it is defined, and may be non-trivial to construct even when unique. On the other hand, for the subclass of *initial-distinct* refinement functions the problem becomes much easier.

**Proposition 10.** *If $r$ is initial-distinct and $U$ an arbitrary transition system, then modulo $\simeq$ there is at most one $r$-abstraction of $U$.*

We denote this $r$-abstraction of $U$ (if there is one) by $U\Uparrow_r$. If, moreover, $U$ is finite-state, then $U\Uparrow_r$ is also finite-state and can be effectively constructed. Finally, for initial-distinct $r$, the inverse of Th. 9 also holds.

**Theorem 11.** *If $r$ is initial-distinct and $T \lesssim^r U$, then $U\Uparrow_r$ exists and $T \simeq U\Uparrow_r$.*

## 5 Example: Interface Refinement

In this section we apply our theory to a small example taken from Brinksma, Jonsson and Orava [4]. The example concerns a distributed data base that can be queried and updated and an agent responsible for updating the data base; the latter can also do some local actions not involving the data base. An important simplification is that the *state* of the data base is completely abstracted away from. Data base and agent are modelled by the following systems $Data_S$ and $Agent_S$:


The problem considered in the paper is to change the interface between data base and agent, so that the two longer communicate over a single update action; rather, updating consists of two separate stages, in which the update is *requested* and *confirmed*, respectively. In our setting, this can be expressed by a refinement function $r$: $upd \mapsto req; cnf$. Moreover, it is required that in the meantime (between request and confirmation), querying the data base should not be disabled. The solution proposed is to refine data base and agent by $Data_I$ and $Agent_I$ in the above figure.

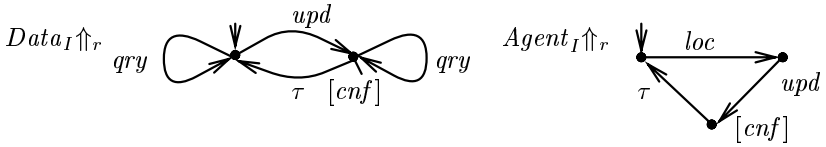
It is seen that, similar to our approach, the proposed implementations differ from the corresponding specifications in the level of abstraction of their alphabets. The correctness criterion employed in the paper circumvents the associated problems by just requiring (horizontal) correctness *after hiding* the relevant actions: i.e., they prove

$$(Data_S \parallel_{upd} Agent_S)/upd \leq (Data_I \parallel_{req,\,cnf} Agent_I)/req, cnf$$

where $\leq$ is a testing preorder. The same result holds in our approach (albeit up to observation equivalence); in that sense, we achieve nothing new. However, our method of establishing this result is quite different.

- The first point is that we can state correctness in a more general manner, *before* hiding the actions that are changed; for in our framework, $Data_S \lesssim^r Data_I$ and $Agent_S \lesssim^r Agent_I$. Moreover, we have an effective way to check

this, through Abstraction Th. 9, by constructing $Data_I\Uparrow_r$ and $Agent_I\Uparrow_r$ and observing $Data_I\Uparrow_r \simeq Data_S$ and $Agent_I\Uparrow_r \simeq Agent_S$:




– The second point is that we can also prove these vertical inequalities *algebraically*, and in fact *derive* $Data_I$ from $Data_S$ and $Agent_I$ from $Agent_S$. (In the approach of [4], such a derivation is possible for $Data$ but not for $Agent$.) For consider the following algebraic specifications:

$$\begin{array}{ll} Data_S = (\mu Q.\ qry; Q) \,|||\, (\mu U.\ upd; U) & Agent_S = \mu A.\ upd; A + loc; A \\ Data_I = (\mu Q.\ qry; Q) \,|||\, (\mu U.\ req; cnf; U) & Agent_I = \mu A.\ req; cnf; A + loc; A \end{array}$$

The correctness of the $Data$-part can be shown as follows:

$$\dfrac{\dfrac{(\mathrm{R}_6)}{\vDash upd \lesssim^r req; cnf}\ (\mathrm{R}_{14}) \quad \dfrac{(\mathrm{R}_{13})}{U \lesssim^r U \vDash U \lesssim^r U}\ (\mathrm{R}_8)}{\dfrac{U \lesssim^r U \vDash upd; U \lesssim^r req; cnf; U}{\vDash \mu U.\ upd; U \lesssim^r \mu U.\ req; cnf; U}\ (\mathrm{R}_{12})}\ (\mathrm{R}_{16})}{\vDash Data_S \lesssim^r Data_I}$$

with $(\mu Q.\ qry; Q)$ above the vertical dots.

The correctness of the $Agent$-part is proved in analogous fashion.

– As a final point, the correctness of the combined system again follows by application of algebraic derivation rules, which allow to prove:

$$\vdash (Data_S \parallel_{upd} Agent_S)/upd \simeq (Data_I \parallel_{req,cnf} Agent_I)/req, cnf$$

Note that we can as easily derive another, incomparable but equally correct implementation for $Data_S$ by first rewriting its specification to the observationally congruent $\mu D.\ qry; D + upd; D$, and applying syntactic substitution to that term. This yields $Data'_I = \mu D.\ qry; D + req; cnf; D$, where the $qry$-action is not possible in between $req$ and $cnf$.

Using the "traditional" approach to action refinement, where refinement is treated as an operator, one can also show that $Data_I$ implements $Data_S$ and $Agent_I$ implements $Agent_S$. In fact, the implementations can even be derived algebraically: [11] gives conditions under which syntactic substitution coincides with semantic refinement, and it so happens that these conditions are satisfied in the present example. Still, in comparison to the traditional approach, vertical implementation offers the following advantages:

- Vertical implementation is based on an interleaving semantics, which means that the results are equally valid when expressed using the transition systems in which the original problem was posed as using the corresponding language description. Not so for traditional action refinement, where a more "precise" specification has to be given than can be done using transition systems: either a term or a more expressive semantic model. That more precise specification will then allow *either* $Data_I$ *or* $Data'_I$ as an implementation (or possibly yet something different); under no circumstances will it allow both. In other words, in the traditional approach, the *design decision* is taken at an earlier stage, namely as soon as the refinement function is given.
- More importantly, vertical implementation "collapses" back to horizontal implementation: having derived $Data_I$ and $Agent_I$, we can compose them, hide the interface actions and get a system that is correct in the well-known, standard interleaving sense. This means that our notion of vertical implementation can be integrated into existing interleaving-based design methods. There is no similar concept in the traditional approach to action refinement.

A problem in the context of action refinement that we have mentioned in the Introduction but ignored thereafter is that traditional refinement is too *strict*: it forces all abstract causalities to be inherited in the implementation. To some degree, we have solved this problem by "closing up to observation congruence," so that apparent abstract causalities may sometimes be turned into independencies. In fact, vertical bisimulation allows a bit more than that, since $\lesssim^r$ already satisfies the following rule:

$$\frac{\Gamma \vdash a \sqsubseteq^r u_1; u_2,\ t \sqsubseteq^r v \quad \neg u_1\checkmark}{\Gamma \vdash a; t \sqsubseteq^r u_1; (u_2 \,|||\, v)}$$

This states that activities that on an abstract level were specified completely after $a$, may in the implementation overlap the "tail" of the implementation of $a$. However, to be really useful, the following rule would be preferable:

$$\frac{\Gamma \vdash a \sqsubseteq^r u_1; u_2,\ t \sqsubseteq^r v_1; v_2 \quad \neg u_1\checkmark}{\Gamma \vdash a; t \sqsubseteq^r u_1; (u_2 \,|||\, v_1); v_2}$$

which expresses that the *start* of the implementation of $t$ may overlap with the *tail* of the implementation of $a$. This latter rule unfortunately does *not* hold for $\lesssim^r$. We intend to study this issue in the future.

## References


1. L. Aceto and M. Hennessy. Termination, deadlock, and divergence. *J. ACM*, 39(1):147–187, Jan. 1992.
2. L. Aceto and M. Hennessy. Towards action-refinement in process algebras. *I&C*, 103:204–269, 1993.

3. L. Aceto and M. Hennessy. Adding action refinement to a finite process algebra. *I&C*, 115:179–247, 1994.
4. E. Brinksma, B. Jonsson, and F. Orava. Refining interfaces of communicating systems. In Abramsky and Maibaum, eds., *TAPSOFT '91, Volume 2*, vol. 494 of *LNCS*, pp. 297–312. Springer, 1991.
5. L. Castellano, G. De Michelis, and L. Pomello. Concurrency vs. interleaving: An instructive example. *Bull. EATCS*, 31:12–15, 1987.
6. I. Czaja, R. J. van Glabbeek, and U. Goltz. Interleaving semantics and action refinement with atomic choice. In Rozenberg, ed., *Advances in Petri Nets 1992*, vol. 609 of *LNCS*, pp. 89–109. Springer, 1992.
7. P. Degano and R. Gorrieri. A causal operational semantics of action refinement. *I&C*, 122:97–119, 1995.
8. P. Degano, R. Gorrieri, and G. Rosolini. A categorical view of process refinement. In De Bakker, De Roever, and Rozenberg, eds., *Semantics: Foundations and Applications*, vol. 666 of *LNCS*, pp. 138–153. Springer, 1992.
9. R. J. van Glabbeek. The linear time – branching time spectrum II: The semantics of sequential systems with silent moves. In Best, ed., *Concur '93*, vol. 715 of *LNCS*, pp. 66–81. Springer, 1993.
10. R. J. van Glabbeek and U. Goltz. Refinement of actions in causality based models. In De Bakker, De Roever, and Rozenberg, eds., *Stepwise Refinement of Distributed Systems — Models, Formalisms, Correctness*, vol. 430 of *LNCS*, pp. 267–300. Springer, 1990.
11. U. Goltz, R. Gorrieri, and A. Rensink. Comparing syntactic and semantic action refinement. *I&C*, 125(2):118–143, Mar. 1996.
12. R. Gorrieri. A hierarchy of system descriptions via atomic linear refinement. *Fund. Informaticae*, 16:289–336, 1992.
13. M. Huhn. Action refinement and property inheritance in systems of sequential agents. In Montanari and Sassone, eds., *Concur '96: Concurrency Theory*, vol. 1119 of *LNCS*, pp. 639–654. Springer, 1996.
14. W. Janssen, M. Poel, and J. Zwiers. Actions systems and action refinement in the development of parallel systems. In Baeten and Groote, eds., *Concur '91*, vol. 527 of *LNCS*, pp. 298–316. Springer, 1991.
15. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
16. M. Nielsen, U. Engberg, and K. G. Larsen. Fully abstract models for a process language with refinement. In De Bakker, De Roever, and Rozenberg, eds., *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, vol. 354 of *LNCS*, pp. 523–549. Springer, 1989.
17. A. Rensink. Methodological aspects of action refinement. In Olderog, ed., *Programming Concepts, Methods and Calculi*, pp. 227–246. IFIP, 1994.
18. A. Rensink. An event-based SOS for a language with refinement. In Desel, ed., *Structures in Concurrency Theory*, pp. 294–309. Springer, 1995.
19. W. Vogler. Failures semantics based on interval semiwords is a congruence for refinement. *Distributed Computing*, 4:139–162, 1991.
20. W. Vogler. Bisimulation and action refinement. *TCS*, 114:173–200, 1993.
21. H. Wehrheim. Parametric action refinement. In Olderog, ed., *Programming Concepts, Methods and Calculi*, pp. 247–266. IFIP, 1994.

This article was processed using the LaTeX macro package with LLNCS style