Dependency-Based Action Refinement

Arend Rensink and Heike Wehrheim*

Institut für Informatik, University of Hildesheim
Postfach 101363, D–31113 Hildesheim, Germany
{rensink,wehrheim}@informatik.uni-hildesheim.de

Abstract. *Action refinement* in process algebras has been widely studied in the last few years as a means to support top-down design of systems. A specific notion of refinement arises when a *dependency relation* on the actions (in the Mazurkiewicz sense) is used to control the inheritance of orderings from the abstract level. In this paper we present a rather simple *operational semantics* for dependency-based action refinement. We show the consistency of the operational with a (previously published) denotational semantics. We moreover show that bisimulation is a *congruence* for dependency-based refinement. Finally, we give an illustrative example.

1 Introduction

Action refinement in process algebras [1, 2, 18, 11, 8, 9] has been proposed to support hierarchical design of reactive systems. Starting with an abstract specification, step-by-step more concrete specifications are constructed by refining actions into concrete processes. Thus the complexity of the design process is reduced and furthermore verification can be facilitated [13]. Standard action refinement uses a *strong* concept of inheritance of causally related abstract actions: if two actions sequentially follow each other on an abstract level, their implementations (as described by the refinement) are also strictly sequential. However, this has turned out to be too restrictive in practice: it might as well be the case that some parts of the implementation of sequential actions overlap in time. As proposed by [14], one way to overcome this problem is to combine action refinement with a dependency relation on actions (in the sense of Mazurkiewicz [16]): the ordering among abstract actions is only inherited to dependent parts of their refinements. Dependency-based action refinement allows an overlapping (concurrent execution) of *independent* parts of the implementation of sequential abstract actions. Other approaches to design by refinement allowing such an overlap, but not based on dependencies, can be found in [10, 21, 23].

In general, two approaches to the definition of an action refinement operator can be found in the literature. On the one hand, action refinement can be defined as *syntactic substitution* on terms [1, 2, 18], on the other hand as *semantic substitution* in an appropriate denotational model [11, 3, 15, 8, 25, 3].

* The research reported in this paper was partially supported by the Human Capital and Mobility Cooperation Network "EXPRESS" (Expressiveness of Languages for Concurrency).

For dependency-based refinement, only the latter approach has been followed yet [14]. In general, the two approaches do not coincide (for a comparison see [12]). *Operational semantics* for action refinement, being consistent with a denotational model, have been defined in [6, 9, 22], however all of them generate transition systems with an enhanced labelling of transition (not just plain actions), adding e.g. causes or event names. The reason for this is essentially that plain transition systems can in general only be distinguished up to *bisimulation* and bisimulation is not a congruence for standard action refinement [7]. Bisimulation over augmented transition systems (as used for giving operational semantics to action refinement) coincides with equivalences (ST-bisimulation, history preserving bisimulation, event isomorphism) which are invariant under refinement.

In this paper we extend a process algebra with action dependencies (as presented in [24]) with an operator for action refinement and develop an operational semantics for it. The basic idea is to reduce refinement to *weak sequential composition* (which is dependency-based sequential composition allowing an overlapping of independent parts of the operands): if the abstract system can perform a sequence of action $a_1 a_2 \ldots$ then the refinement can perform $r(a_1) \cdot r(a_2) \cdot \ldots$ ($r(a)$ is the refinement of an action a according to a refinement function r, $\cdot$ denotes weak sequential composition). This allows independent parts of refinements of sequentially ordered actions to be executed concurrently. The transition relation of the resulting labelled transition system is solely labelled with actions.

The operational semantics is shown to be consistent with the denotational semantics of dependency-based action refinement as introduced in [26]. We furthermore show that bisimulation is a congruence for action refinement. Finally we illustrate the practical applicability of our approach by an example.

2 Definitions

We start with a brief repetition of the operational and denotational semantics for the process algebra $\mathbf{L}$ as developed in [24]. Thereafter, $\mathbf{L}$ will be extended by an action refinement operator.

We assume a distributed alphabet $\langle \mathbf{A}, I \rangle$, where $\mathbf{A}$ is a set of actions and $I \subseteq \mathbf{A} \times \mathbf{A}$ is a symmetric and irreflexive *independence relation.* $\mathbf{A}$ is ranged over by a, b, c. We denote $a \; D \; b$ iff $\neg(a \; I \; b)$, and $[A]_D := \{b \; D \; a \mid a \in A\}$. D is called the dependency relation. Intuitively, actions are dependent if they share some common resource, which can for instance be a shared variable, a database item or a printer. We also use a set of process variables $\mathbf{X}$ to allow for the definition of infinite behaviour. The language $\mathbf{L}$ is generated by the following productions:

$$B ::= \mathbf{0}_P \mid a \mid X \mid B \cdot B \mid B + B \mid B \parallel_A B \mid \mu X.B$$

where $a \in \mathbf{A}$, $A, P \subseteq \mathbf{A}$ and $X \in \mathbf{X}$. A term is called *closed* iff all variables are in the scope of a fixpoint operator. In the sequel we only consider closed terms. We let $B\{B'/X\}$ stand for the substitution of all occurrences of the variable X in B

by B'. Furthermore, we denote by $\alpha(B)$ the set of all actions which syntactically occur in B.

The operators have the following informal meaning: $\mathbf{0}_P$ describes an empty process (below we comment on the set P), a a process executing the action a, $\cdot$ is *weak* sequential composition, $+$ denotes a CCS-like choice (with a non-standard interaction with sequential composition), $\|_A$ a TCSP-like parallel composition with synchronisation on actions in A and $\mu X.B$ defines recursion. We only consider *guarded* recursion, where a term B is considered guarded if all variables X only occur in subterms of the form $a \cdot X$ with $[a]_D = \mathbf{A}$.

The information about the dependencies of actions is exploited to give a semantics to terms which ensures that independent actions never have to wait for one another to proceed. This results in a *weak* form of sequential composition which allows actions from the second operand that are independent of the first to be executed immediately. To capture the intended behaviour of weak sequential composition, a particular notion of *termination* is needed. It is not sufficient anymore to solely distinguish between successful termination and deadlock; instead, a process is said to be terminated *wrt. a set of actions P* (intuitively if it does not want to execute anything on which actions out of P depend). In a sequential composition, the second component can perform an action a if the first component is terminated wrt. a (e.g. $a \cdot b$, $a\ I\ b$, can execute b since a is terminated wrt. b). This notion of termination (called *permission*) is also reflected in the constants representing empty processes: for any $P \subseteq \mathbf{A}$, $\mathbf{0}_P$ is an empty process terminated wrt. P (thus $\mathbf{0}_{\{a\}} \cdot a$ allows execution of a). We let $\mathbf{1} := \mathbf{0}_\mathbf{A}$ stand for the completely terminated process.

In connection with weak sequential composition, the semantics of choice is also non-standard. Choices can be resolved by actions not participating in the choice but sequentially following it. As an example take $B = (a+b) \cdot c$, $a\ D\ c$, $b\ I\ c$. Since b and c are independent, c can also be executed first; afterwards, however, a is not possible anymore, since otherwise the specified order between dependent actions, a and c, is not met.

In Table 1 we give a structured operational semantics for $\mathbf{L}$. Besides a transition relation ($\rightarrow$) we also include a *permission relation* ($\cdot\cdot\!\rightarrow$) to describe the new form of termination. In contrast to the usual termination predicate, permission is a binary relation, since permission may change processes due to the resolution of choices (see above). This is reflected by the permission rule for choices: if only one operand of the choice permits an action, the term changes with the permission. The standard semantic model of labelled transition systems can be extended to permissions.

Definition 1. A *transition-permission-system* (tps for short) is a tuple $T = \langle S, \rightarrow, \cdot\cdot\!\rightarrow \rangle$ with

- S a set of *states*,
- $\rightarrow \subseteq S \times \mathbf{A} \times S$ a *transition relation*,
- $\cdot\cdot\!\rightarrow \subseteq S \times \mathbf{A} \times S$ a *permission relation*.

In particular, the rules in Table 1 define relations $\rightarrow$ and $\cdot\cdot\!\rightarrow$ over $\mathbf{L}$ that give rise to a transition-permission system $(\mathbf{L}, \rightarrow, \cdot\cdot\!\rightarrow)$.

$$\frac{a \in P}{\mathbf{0}_P \stackrel{a}{\dashrightarrow} \mathbf{0}_P} \qquad \frac{a\ I\ b}{b \stackrel{a}{\dashrightarrow} b} \qquad \frac{}{a \xrightarrow{a} \mathbf{1}}$$

$$\frac{B_1 \stackrel{a}{\dashrightarrow} B' \quad B_2 \stackrel{a}{\not\dashrightarrow}}{B_1 + B_2 \stackrel{a}{\dashrightarrow} B'} \qquad \frac{B_2 \stackrel{a}{\dashrightarrow} B' \quad B_1 \stackrel{a}{\not\dashrightarrow}}{B_1 + B_2 \stackrel{a}{\dashrightarrow} B'} \qquad \frac{B_1 \xrightarrow{a} B'}{B_1 + B_2 \xrightarrow{a} B'} \qquad \frac{B_2 \xrightarrow{a} B'}{B_1 + B_2 \xrightarrow{a} B'}$$

$$\frac{B_1 \stackrel{a}{\dashrightarrow} B_1' \quad B_2 \stackrel{a}{\dashrightarrow} B_2'}{B_1 \cdot B_2 \stackrel{a}{\dashrightarrow} B_1' \cdot B_2'} \qquad \frac{B_1 \xrightarrow{a} B_1'}{B_1 \cdot B_2 \xrightarrow{a} B_1' \cdot B_2} \qquad \frac{B_1 \stackrel{a}{\dashrightarrow} B_1' \quad B_2 \xrightarrow{a} B_2'}{B_1 \cdot B_2 \xrightarrow{a} B_1' \cdot B_2'}$$

$$\frac{B_1 \stackrel{a}{\dashrightarrow} B_1' \quad B_2 \stackrel{a}{\dashrightarrow} B_2'}{B_1 \parallel_A B_2 \stackrel{a}{\dashrightarrow} B_1' \parallel_A B_2'} \qquad \frac{B_1 \xrightarrow{a} B_1', a \notin A}{B_1 \parallel_A B_2 \xrightarrow{a} B_1' \parallel_A B_2} \qquad \frac{B_2 \xrightarrow{a} B_2', a \notin A}{B_1 \parallel_A B_2 \xrightarrow{a} B_1 \parallel_A B_2'}$$

$$\frac{B_1 \xrightarrow{a} B_1' \quad B_2 \xrightarrow{a} B_2', a \in A}{B_1 \parallel_A B_2 \xrightarrow{a} B_1' \parallel_A B_2'} \qquad \frac{B\{\mu X.B/X\} \stackrel{a}{\dashrightarrow} B'}{\mu X.B \stackrel{a}{\dashrightarrow} B'} \qquad \frac{B\{\mu X.B/X\} \xrightarrow{a} B'}{\mu X.B \xrightarrow{a} B'}$$

Table 1. Operational semantics for **L**

In a transition-permission system T, a state s' is called *reachable* from a state s if $(s, s') \in (\bigcup_a \xrightarrow{a} \cup \stackrel{a}{\dashrightarrow})^*$. Furthermore, s is called *fully terminated* if $s \stackrel{a}{\dashrightarrow} s$ for all $a \in \mathbf{A}$, and *terminating* if for all s' reachable from s, there is a fully terminated s'' reachable from s'. The usual notion of (strong) bisimilarity of [19, 17] on labelled transition systems can be adapted to our setting.

Definition 2. Let $T = \langle S, \rightarrow, \dashrightarrow \rangle$ be a tps. *Bisimilariy over* T is the largest symmetrical relation $\sim \subseteq S \times S$ such that whenever $s_1 \sim s_2$ then

1. $s_1 \xrightarrow{a} s_1'$ implies $\exists s_2' : s_2 \xrightarrow{a} s_2'$ and $s_1' \sim s_2'$;
2. $s_1 \stackrel{a}{\dashrightarrow} s_1'$ implies $\exists s_2' : s_2 \stackrel{a}{\dashrightarrow} s_2'$ and $s_1' \sim s_2'$.

Denotational semantics. We now recall the event-based denotational model for **L** developed in [26], with a slight deviation to smoothen the definition of action refinement, later on. In this model, a process is represented by its set of maximal runs (this is the deviation from [26], where all runs where included, not just the maximal ones). Each run consists of a set of *events* that have happened (where an event corresponds to the execution of an action), together with their causal ordering and a labelling function indicating which actions have been executed. Furthermore, each run contains *termination information*, in the form of a set containing the actions with respect to which the run is terminated. The causal ordering has to be consistent with the dependency relation: only dependent actions may (and must) be ordered. For the construction of runs we assume a universe of events $\mathbf{E}$, ranged over by d, e.

Definition 3. A run is a tuple $u = \langle E, <, \ell, \checkmark \rangle$ where

- $E \subseteq \mathbf{E}$ is a (finite or infinite) set of events;
- $< \subseteq E \times E$ is an acyclic ordering on E;

- $\ell: E \to \mathbf{A}$ is a labelling function, such that $\ell(d)\ D\ \ell(e)$ iff $d \leq e$ or $e \leq d$;
- $\surd \subseteq \mathbf{A}$ is a *termination set.*

The class of runs is denoted $\mathbf{R}$. We denote the elements of a run u by E_u, $<_u$, ℓ_u and $\surd_u$. $\leq$ will denote the transitive closure of $<$. The independence relation I is extended to E_u through ℓ_u such that $d\ D_u\ e$ iff $\ell_u(d)\ D\ \ell_u(e)$. A run can be depicted by a graph; e.g., $\boxed{{}_1a \to {}_2b}, \mathbf{A}$ denotes a run consisting of two ordered events, 1 and 2 labelled with a and b, respectively, and termination set is $\mathbf{A}$. A run u is called a *prefix* of a run v if the following holds:

$$u \sqsubseteq v :\Leftrightarrow (E_u \subseteq E_v) \wedge (<_u = <_v \cap (E_u \times E_v)) \wedge \surd_u = \surd_v \setminus [\ell_v(E_v \setminus E_u)]_D \ .$$

If $\mathcal{P} \subseteq \mathbf{R}$, we denote $E_\mathcal{P} = \bigcup_{u \in \mathcal{P}} E_u$ and $\ell_\mathcal{P} = \bigcup_{u \in \mathcal{P}} \ell_u$. $\mathcal{P}$ is called *labelling consistent* if $\ell_\mathcal{P}$ is a function. Our denotational models will be nonempty, labelling consistent sets of runs, called *families of runs.* The class of families of runs is denoted $\mathbf{M}$. They are an extension of the *families of posets* proposed in [20]. Only the $\sqsubseteq$-maximal elements of a family of runs are considered significant. This is expressed by taking two models, $\mathcal{P}$ and $\mathcal{Q}$, as equivalent if there is a bijection $\phi: E_\mathcal{P} \to E_\mathcal{Q}$ such that $\phi^*(v) \in \max_\sqsubseteq \mathcal{Q}$ for all $u \in \max_\sqsubseteq \mathcal{P}$ (where ϕ^* denotes the natural pointwise extension of ϕ to $\mathbf{R}$).

In [26] we have presented a denotational semantics for $\mathbf{L}$ that can easily be adapted to the above families of runs, giving rise to a mapping $[\![\cdot]\!]: \mathbf{L} \to \mathbf{M}$. Due to lack of space, we omit the model constructions for the standard operators. As an example, assume $\mathbf{A} = \{a, b, c\}$ with $a\ D\ c$, $a\ D\ b$ and $b\ I\ c$. A family of runs modelling the term $(a + b) \cdot (b \parallel_\varnothing c)$ is given by $\boxed{{}_0a \to {}_2b,\ {}_0a \to {}_3c}, \mathbf{A}$ and $\boxed{{}_1b \to {}_2b \quad {}_3c}, \mathbf{A}$. Without proof, we state the following property:

Proposition 4. *B is terminating iff $\surd_u = \mathbf{A}$ for all $u \in \max_\sqsubseteq [\![B]\!]$.*

Families of runs distinguish concurrent from interleaving behaviour; moreover, the branching structure of the behaviour can be reconstructed via the names of events. Thus, the model is quite expressive; much more so, in fact, than necessary for our present purposes. In this paper, we interpret families of runs up to bisimulation, by regarding them as transition-permission systems in the sense of Definition 1:

(1) A transition corresponds to the occurrence of an event; the label of the transition is the event label. For an event to occur it must be *enabled* in a run, i.e., have no causal predecessors; afterwards, it can be discarded. Hence, if $e \in \min E_u$, the *remainder* of u after e is given by

$$u \setminus e := \langle E \setminus \{e\}, < \cap ((E \setminus \{e\}) \times (E \setminus \{e\})), \ell \restriction (E \setminus \{e\}), \surd_u \rangle$$

(2) A permission corresponds to the occurrence of a *future* action a. For any given run u, such a future action a can only be allowed if it is independent of all actions of u and u is terminated w.r.t. a. We say that u *permits* a iff $a \in \surd_u$ and $\forall e \in E_u.\ \ell_u(e)\ I\ a$. The following relations are considered to hold iff the right hand sides are nonempty:

$$\mathcal{P} \xrightarrow{\ell_\mathcal{P}(e)} \{u \setminus e \mid u \in \mathcal{P}, e \in \min E_u\}$$
$$\mathcal{P} \overset{a}{\dashrightarrow} \{u \in \mathcal{P} \mid u \text{ permits } a\}$$

This gives rise to a transition-permission system $(\mathbf{M}, \rightarrow, \dashrightarrow)$. We then have the following consistency result (cf. [24]):

Theorem 5. *For arbitrary terms $B \in \mathbf{L}$, $B \sim [\![B]\!]$ (in the transition-permission system $\mathbf{L} \cup \mathbf{M}$).*

3 Action Refinement

We now extend our language with the *action refinement* operator discussed in the introduction. Syntactically, the extension is given by the term $B[r]$, where r stands for a *refinement function* $r\colon \mathbf{A} \rightarrow \mathbf{L}$. In contrast to standard action refinement in causality-based models [11], in our setting, the inheritance of abstract orderings by the concrete actions of the refinement is driven by the dependencies between the latter. Therefore, the refinement of ordered abstract actions may result in sets of events which partially overlap in their execution. This is demonstrated in the following example.

Example 1. Let $B = a \cdot b$ with $a\ D\ b$; hence $[\![B]\!] = \{\boxed{{}_1a \rightarrow {}_2b}, \mathbf{A}\}$. Let $r\colon a \mapsto a_1 \cdot a_2$ with $a_1\ D\ a_2$, and $b \mapsto b_1 \cdot b_2$ with $b_1\ D\ b_2$, such that $a_1\ D\ b_1$ and $a_2\ D\ b_2$ but $a_2\ I\ b_1$ and $a_1\ I\ b_2$. The only allowed execution of $B[r]$ by standard refinement would be $a_1a_2b_1b_2$, where the entire refinement of b has to wait for a to complete. With dependency-based refinement we get the following run, which also allows an overlapping execution $a_1b_1a_2b_2$:

$$\boxed{\begin{array}{ccc} {}_{(1,1)}a_1 & \rightarrow & {}_{(1,2)}a_2 \\ \downarrow & & \downarrow \\ {}_{(2,1)}b_1 & \rightarrow & {}_{(2,2)}b_2 \end{array}}, \mathbf{A}$$

In the following we extend both the operational and the denotational semantics of $\mathbf{L}$ to capture this type of dependency-based refinement. To achive this, we have to impose two restrictions on refinement functions:
(1) All images have to be *terminating* (see Section 2). This is a quite natural restriction, given the fact that abstractly, an action is atomic, i.e., cannot deadlock during its execution.
(2) The refinement has to *preserve dependency and independency.* This property is called *D-consistency* below. Preservation of dependency, in our case, means that if $a\ D\ b$ then a also has to depend on the *initial* actions of $r(b)$. Preservation of independency means that if $a\ I\ b$ then a is independent of *all* actions of $r(b)$. Formally, r is called terminating if $r(a)$ is terminating for all $a \in \mathbf{A}$, and D-consistent if

$$a\ D\ b \Longrightarrow r(a) \overset{b}{\dashrightarrow}\!\!\!\!\!/\ \ \wedge \forall r(a) \xrightarrow{a'} : a'\ D\ b$$
$$a\ I\ b \Longrightarrow \forall a' \in \alpha(r(a)) : a'\ I\ b$$

Operational semantics. We develop SOS rules for refinement. With respect to the transitions of $B[r]$, it is clear that we need at least a rule of the following form: from $B \xrightarrow{a} B'$ and $r(a) \xrightarrow{b} C$, conclude $B[r] \xrightarrow{b} B''$ for some term B''. The interesting part is the choice of B''. It should capture the following points: (1) The refinement of a should be able to proceed, i.e., B'' should somehow contain

C; (2) B'' should resolve all the choices in B in the same way as in B'; (3) B'' should be able to start the refinements of all a-independent actions allowed by B' and (4) B'' should still contain the function r. A rule which captures all these aspects is:

$$\frac{B \xrightarrow{a} B',\ r(a) \xrightarrow{b} C}{B[r] \xrightarrow{b} C \cdot B'[r]} \tag{1}$$

Hence, B'' equals the sequential composition of C with $B'[r]$. To come back to Example 1, the overlapping execution of the refinement of a and b is thus derivable:

$$B[r] \xrightarrow{a_1} a_2 \cdot b[r] \xrightarrow{b_1} a_2 \cdot b_2 \cdot \mathbf{1}[r] \xrightarrow{a_2} \mathbf{1} \cdot b_2 \cdot \mathbf{1}[r] \xrightarrow{b_2} \mathbf{1} \cdot \mathbf{1} \cdot \mathbf{1}[r]$$

The rule for permissions is straightforward, and reflects the intuition behind D-consistency: If the abstract system B permits an action a, then the refined system permits it as well.

$$\frac{B \overset{a}{\dashrightarrow} B'}{B[r] \overset{a}{\dashrightarrow} B'[r]} \tag{2}$$

The operational semantics of **L**-plus-refinement, therefore, is determined by Table 1 augmented by Equations (1) and (2). It is noteworthy that these operational refinement rules are simpler by far than the ones obtained in other approaches, in particular [9] and [22]. For instance, we no not rely on auxiliary operators of any kind.

An immediate question concerns the congruence of our semantic equivalence, bisimulation. This is proved by showing that our operational rules obey a certain format, which must at least allow negative premises. The format we choose is GSOS [4]. In order to apply this to our setting, with two kinds of transition relations, we can extend $\mathbf{A}$ by a set $\overline{\mathbf{A}}$ and define $\overset{a}{\dashrightarrow}$ to be $\xrightarrow{\bar{a}}$.

Theorem 6. *$\sim$ is a congruence for refinement.*

It is clear that the congruence property does not depend on the termination or D-consistency of r, since these requirements are not expressed in the operational rules in any way. On the other hand, for refinement functions that are not D-consistent, the operationally derived behaviour may deviate from the expected.

Example 2. Consider $B = a \cdot b$ with $a\ D\ b$, and a D-inconsistent refinement in which some initial action b_1 from $r(b)$ is independent of the refinement of a; e.g., $r : a \mapsto a', b \mapsto b_1 \cdot b_2$ such that $a'\ I\ b_1$ and $a'\ D\ b_2$. Intuitively, since ab_1b_2 is an execution of $B[r]$ and b_1 is independent of a, $B[r]$ should also be able to start with b_1. However, this cannot be inferred operationally: the only initial action of B is a, and therefore $B[r]$ cannot start with an action not coming from $r(a)$.

We will strengthen and formalise the concept of "intuitive correctness" for the operational semantics by investigating a denotational characterisation of the refinement operator, and showing that the resulting models coincide, at least for D-consistent refinement functions.

Denotational semantics. For the denotational semantics of $B[r]$, we first derive a semantic function $[\![r]\!]\colon \mathbf{A} \to \mathbf{M}$ according to $[\![r]\!]\colon a \mapsto [\![r(a)]\!]$, and then show how to apply such a function to an arbitrary model $\mathcal{P} \in \mathbf{M}$. The latter concerns a pointwise extension of the refinements of single runs.

- Given a model $\mathcal{P} \in \mathbf{M}$ and a semantic refinement function $\mathcal{R}\colon \mathbf{A} \to \mathbf{M}$, a *witness* is a function $w\colon E_{\mathcal{P}} \to \mathbf{R}$ such that $w(e) \in \mathcal{R}(\ell_{\mathcal{P}}(e))$ and $\surd_{w(e)} = \mathbf{A}$ for all $e \in E_{\mathcal{P}}$.
- Given a run $u \in \mathcal{P}$ and witness $w\colon E_{\mathcal{P}} \to \mathbf{R}$, the refinement of u by w replaces all $e \in E_u$ by their w-images, and orders the resulting events, *insofar they are dependent*, according to the ordering of u. This is defined formally below.
- The refinement of $\mathcal{P}$ by $\mathcal{R}$ is defined as the set of all $\mathcal{P}$-runs refined by all $\mathcal{R}$-witnesses: $\mathcal{R}(\mathcal{P}) = \{w(u) \mid w \text{ is a } \mathcal{R}\text{-witness}, u \in \mathcal{P}\}$.

Definition 7. Let u be a run and $w\colon E \to \mathbf{R}$ a function with $E_u \subseteq E$. The *refinement of u by w*, denoted $w(u)$, is defined as:

- $E_{w(u)} := \bigcup_{e \in E_u} \{e\} \times E_{w(e)}$;
- $(d, d') <_{w(u)} (e, e') \;:\Leftrightarrow\; (d = e \wedge d' <_{w(e)} e') \vee (d <_u e \wedge d' \; D_{w(u)} \; e')$;
- $\ell_{w(u)}\colon (e, e') \mapsto \ell_{w(e)}(e')$ for all $(e, e') \in E_{w(u)}$;
- $\surd_{w(u)} := \surd_u$.

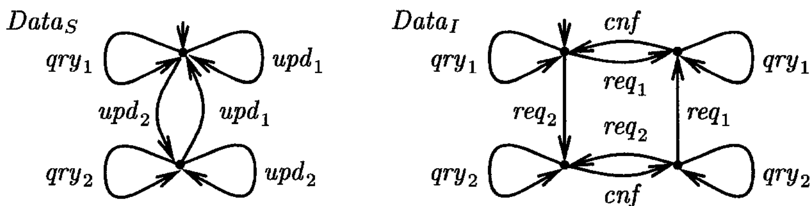
We can now extend the semantic mapping $[\![\cdot]\!]$ to the language with refinement, through the rule $[\![B[r]]\!] := [\![r]\!]([\![B]\!])$. The consistency result of Theorem 5 can now be extended to the full language.

Theorem 8. *For arbitrary B and terminating, D-consistent r,* $B[r] \sim [\![B[r]]\!]$.

For the proof see [27]. Here, the requirement of D-consistency is crucial: The denotational semantics can describe the intuitively expected behaviour of Example 2. This shows that the operational and denotational interpretations may differ outside the class of D-consistent refinements. Moreover, for D-inconsistent refinements, bisimulation may fail to be a congruence. For more details see [27].

4 Example: Data base access

In this section we apply our theory to a small example inspired by Brinksma, Jonsson and Orava [5]. The example concerns a distributed data base that can be queried and updated. We assume that there are only two possible data, which we denote 1 and 2. The data base specification is modelled by the transition system $Data_S$ in the following figure:



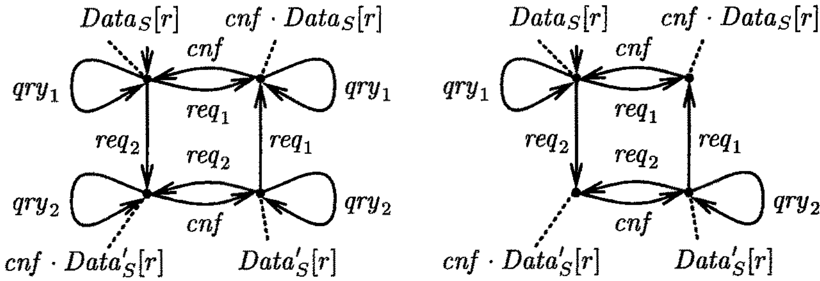
The problem considered in the paper is to change the interface of the data base, so that updating consists not of a single action but of two separate stages, in

which the update is *requested* and *confirmed*, respectively. In our setting, this can be expressed by a refinement function $r\colon upd_i \mapsto req_i; cnf$. Moreover, it is required that in the meantime (between request and confirmation), querying the data base should still be possible. This results in the behaviour $Data_I$ above.

In our approach, this implementation can be obtained algebraically through an application of the refinement operator. The overlap between qry_i and cnf is obtained by setting the dependencies appropriately: $qry_i\ D\ req_j$ but $qry_i\ I\ cnf$.

$$Data_S = \mu D_1.\ qry_1 \cdot D_1 + upd_1 \cdot D_1 + upd_2 \cdot \mu D_2.\ qry_2 \cdot D_2 + upd_1 \cdot D_1 + upd_2 \cdot D_2$$
$$Data'_S = \mu D_2.\ qry_2 \cdot D_2 + upd_2 \cdot D_2 + upd_1 \cdot \mu D_1.\ qry_1 \cdot D_1 + upd_2 \cdot D_2 + upd_1 \cdot D_1$$

The operational behaviour of $Data_I = Data_S[r]$ is given by the left hand transition system in the following figure. The right hand system shows the case where $qry_i\ D\ cnf$ instead, in which case the next query must wait for the second phase of the updating to finish.



References

1. L. Aceto and M. Hennessy. Towards action-refinement in process algebras. *Information and Computation*, 103(2):204–269, 1993.
2. L. Aceto and M. Hennessy. Adding action refinement to a finite process algebra. *Information and Computation*, 115(2):179–247, 1994.
3. E. Best, R. Devillers, and J. Esparza. General refinement and recursion operators for the Petri box calculus. In Enjalbert, Finkel, and Wagner, eds., *STACS 93*, vol. 665 of *LNCS*, pp. 130–140. Springer, 1993.
4. B. Bloom, S. Istrail, and A. R. Meyer. Bisimulation can't be traced. *Journal of the ACM*, 42(1):232–268, Jan. 1995.
5. 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.
6. N. Busi, R. van Glabbeek, and R. Gorrieri. Axiomatising ST bisimulation equivalence. In Olderog, ed., *Programming Concepts, Methods and Calculi*, vol. A–56 of *IFIP Transactions*, pp. 169–188. IFIP, 1994.
7. L. Castellano, G. De Michelis, and L. Pomello. Concurrency vs. interleaving: An instructive example. *Bull. Eur. Ass. Theoret. Comput. Sci.*, 31:12–15, 1987. Note.
8. P. Darondeau and P. Degano. Refinement of actions in event structures and causal trees. *Theoretical Computer Science*, 118:21–48, 1993.

9. P. Degano and R. Gorrieri. A causal operational semantics of action refinement. *Information and Computation*, 122(1):97–119, 1995.
10. 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.
11. R. van Glabbeek and U. Goltz. Equivalences and refinement. In Guessarian, ed., *18éme Ecole de Printemps d'Informatique Théorique Semantique du Parallelisme*, vol. 469 of *LNCS*, 1990.
12. U. Goltz, R. Gorrieri, and A. Rensink. Comparing syntactic and semantic action refinement. *Information and Computation*, 125(2):118–143, 1996.
13. M. Huhn. Action refinement and property inheritance in systems of sequential agents. In Montanari and Sassone, eds., *Concur'96*, 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. L. Jategaonkar and A. Meyer. Testing equivalences for Petri nets with action refinement. In Cleaveland, ed., *Concur '92*, vol. 630 of *LNCS*, pp. 17–31. Springer, 1992.
16. A. Mazurkiewicz. Basic notions of trace theory. 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. 285–363. Springer, 1989.
17. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
18. 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.
19. D. Park. Concurrency and automata on infinite sequences. In Deussen, ed., *Proceedings 5th GI Conference*, vol. 104 of *LNCS*, pp. 167–183. Springer, 1981.
20. A. Rensink. Posets for configurations! In Cleaveland, ed., *Concur '92*, vol. 630 of *LNCS*, pp. 269–285. Springer, 1992.
21. A. Rensink. Methodological aspects of action refinement. In Olderog, ed., *Programming concepts, methods and calculi*, vol. A–56 of *IFIP Transactions*. IFIP, 1994.
22. A. Rensink. An event-based SOS for a language with refinement. In Desel, ed., *Structures in Concurrency Theory*, Workshops in Computing, pp. 294–309. Springer, 1995.
23. A. Rensink and R. Gorrieri. Action refinement as an implementation relation. In Bidoit and Dauchet, eds., *TAPSOFT '97: Theory and Practice of Software Development*, vol. 1214 of *LNCS*, pp. 772–786. Springer, 1997.
24. A. Rensink and H. Wehrheim. Weak sequential composition in process algebras. In Jonsson and Parrow, eds., *Concur '94: Concurrency Theory*, vol. 836 of *LNCS*, pp. 226–241. Springer, 1994.
25. W. Vogler. Failure semantics based on interval semiwords is a congruence for refinement. *Distributed Computing*, 4:139–162, 1991.
26. H. Wehrheim. Parametric action refinement. In Olderog, ed., *IFIP Transactions: Programming Concepts, Methods and Calculi*, pp. 247–266. Elsevier, 1994.
27. H. Wehrheim. *Specifying Reactive Systems with Action Dependencies: Modelling and Hierarchical Design*. PhD thesis, University of Hildesheim, 1996.