Acta Informatica 35, 131–165 (1998)

Acta Informatica
© Springer-Verlag 1998

Refinement of fair action systems

Ralph J.R. Back, Qiwen Xu[⋆]

Department of Computer Science, Åbo Akademi, Lemminkainenkatu 14, FIN-20520 Turku, Finland (e-mail: backrj@ra.abo.fi)

Received: 16 March 1995 / 16 April 1997

Abstract. An action system is a framework for describing parallel or distributed systems, for which the refinement calculus offers a formalisation of the stepwise development method. Fairness is an important notion in modelling parallel or distributed systems, and this paper investigates a calculus for refinement of fair action systems. Simulations, which are proof techniques for refinement, are extended to verify fair action systems. Our work differs from others' in that the additional condition concerning fairness is expressed through termination of related iteration statements. For this purpose, existing proof rules for termination are extended. In the tradition of the refinement calculus, our approach to fairness is based on techniques developed mainly for sequential programming.

1. Introduction

An *action system* is a framework for describing parallel or distributed systems. It focuses on specifying the logical behaviours of the systems by a collection of *actions*. Actions are expressed in familiar sequential programming notations, and are executed atomically. The action systems formalism was proposed by Back and Kurki-Suonio [5]. Similar action-based formalisms have later been used by several other researchers, e.g., Chandy and Misra in UNITY [10], and Lamport in TLA [18].

The *refinement calculus* is a formalisation of the stepwise refinement method of program construction. It was originally designed for derivation of sequential programs by Back [2, 3]. Afterwards it has been studied and extended by a number of researchers (see [21, 22] among others) for the same purpose. More recently, Back and Sere [6] extended the refinement calculus to the design of action systems with respect to *total correctness*. *Reactive* refinement of action

[⋆] Current address: UNU/IIST, P.O. Box 3058, Macau (e-mail: qxu@iist.unu.edu) systems was investigated by Back [4] basing on the techniques for *data refinement* [2, 7].

This paper extends the refinement calculus to deal with action systems that contain *fairness*. Fairness is an important notion in modelling parallel or distributed systems, and was studied in the early work of Back and Kurki-Suonio [5] on action systems. Fairness was also investigated in [4], where explicit coding was employed to model fairness. We aim to handle fairness in the refinement calculus more directly in this paper.

We are interested in refinement which preserves reactive properties. Therefore, we need to record computation sequences in the semantics. A lower level system refines a higher level one, if for any lower level computation, there is a higher level computation which presents identical observations; in this case, the higher level computation is said to *approximate* the lower level one. Refinement holds between fair action systems, if for any fair lower level computation, there is a fair higher level computation which approximates it. *Simulation* offers an effective technique to reduce the verification of the complete action systems to the verification of individual actions (together with some other easily checkable conditions). Simulations of fair action systems are obtained by extending those of unfair systems. In related work on refinement of fair systems, the additional proof conditions concerning fairness are typically expressed in a temporal logic [18, 16, 17]. However, introducing a temporal logic into the refinement calculus would put an extra burden of learning a new notation on users. Moreover, although fairness can be neatly defined using temporal modalities, verifying it still resorts to the more basic Hoare logic style reasoning. We observe that the concerned proof obligations can instead be expressed by termination of related fair iteration statements. Therefore, as in the case of ordinary refinement, our approach to fairness is based on the basic techniques developed mainly for sequential programming.

This paper is organised as follows. Section 2 describes the basics of refinement calculus and action systems. For the sake of presentation, we first restrict ourselves to weak fairness: in Sections 3 and 4, we investigate proof methods for termination and total correctness of weakly fair iteration statements, and refinement of weakly fair action systems. Strong fairness is studied in Section 5. Throughout the paper, a number of toy examples are used to illustrate various proof rules. Section 6 is devoted to a more advanced case study in which a mutual exclusion algorithm is developed. In these sections, we study forward simulation. In Section 7, the dual notion of backward simulation is extended for verifying fair action systems. The last section is a brief discussion.

2. Preliminaries

2.1. Refinement calculus

The basic domains of refinement calculus arise by pointwise extension from the boolean lattice. The truth values

$$\mathsf{Bool} = \{\mathsf{T}, \mathsf{F}\}$$

form a complete lattice under the *implication order*

$$\mathsf{F} \leq \mathsf{T} \qquad \mathsf{T} \leq \mathsf{T} \qquad \mathsf{F} \leq \mathsf{F}$$

Complement $\neg$, meet $\wedge$ and join $\vee$ are respectively negation, conjunction and disjunction.

Let Σ be a set of states. A *predicate over* Σ is a function $p : \Sigma \to \mathsf{Bool}$ which assigns a truth value to each state. The set of predicates over Σ

$$\mathsf{Pred}(\Sigma) \overset{\text{def}}{=} \Sigma \to \mathsf{Bool}$$

also forms a complete lattice under the order obtained from boolean implication by pointwise extension: for $p, q \in \mathsf{Pred}(\Sigma)$

$$p \leq q \qquad \text{iff} \qquad (\forall \sigma \in \Sigma.\ p\,\sigma \leq q\,\sigma)$$

Complement $\neg$, meet $\wedge$ and join $\vee$ are defined pointwisely too, e.g., $(p \wedge q)\,\sigma \overset{\text{def}}{=} (p\,\sigma \wedge q\,\sigma)$. The identically false predicate false is the bottom, and the identically true predicate true is the top, of the predicate lattice. The derived combinator $\Rightarrow$ is defined as usual, i.e., $(p \Rightarrow q) \overset{\text{def}}{=} (\neg p \vee q)$. In this paper, we assume that binding power of various operators decreases along the usual order

functional application
$\neg$
$\vee,\ \wedge$
$\Rightarrow$

Pointwise extension of predicates gives us predicate transformers, which are functions of the type

$$\mathsf{Ptran}(\Sigma, \Gamma) \overset{\text{def}}{=} \mathsf{Pred}(\Gamma) \to \mathsf{Pred}(\Sigma)$$

where Σ and Γ are two state spaces. A program statement A is identified with the *weakest precondition* predicate transformer [12], which maps a postcondition q to a precondition $A\,q$ that describes the set of initial states from which A is guaranteed to terminate in states satisfying q. The definition of the predicate transformer regards a program which does not always produce the desired result as bad as one which does not produce the result at all. In particular, a program which cannot guarantee termination is identified with one which is totally nonterminating.

For S, T: $\mathsf{Ptran}(\Sigma, \Gamma)$, the *refinement order* is

$$S \leq T \overset{\text{def}}{=} (\forall q \in \mathsf{Pred}(\Gamma).S\ q \leq T\ q)$$

Intuitively, T refines S if and only if the former guarantees any postcondition that the latter does. Under this order, predicate transformers form a complete lattice: the bottom is abort which maps any postcondition to false, and the top is magic which maps any postcondition to true; meet and join are again defined pointwisely, e.g., $(S \wedge T)q \overset{\text{def}}{=} (S\ q \wedge T\ q)$. Statement abort never guarantees anything.

Statement magic is *miraculous*, for it promises to achieve any postcondition; therefore, it is an imaginary statement, useful only in formal calculations. Meet and join model *demonic* and *angelic* choices respectively.

A statement S is (positively) *conjunctive*, if for any nonempty set of predicates $\{q_i | i \in I\}$,

$$S(\forall i \in I . q_i) \overset{\text{def}}{=} (\forall i \in I . S\ q_i)$$

We call a conjunctive statement an action. The *non-miraculous domain* or *guard* of action A is defined by

$$gA \overset{\text{def}}{=} \neg A\,\mathsf{false}$$

An action A is enabled in any state where gA holds. The *termination domain* of A is defined by

$$tA \overset{\text{def}}{=} A\,\mathsf{true}$$

Actions can be combined by sequential program operators to form a compound action. We only mention a few structures that are particularly useful. The *sequential composition* is define as usual

$$(A_1; A_2)q \overset{\text{def}}{=} A_1(A_2 q)$$

It has skip (formally defined as skip $q \overset{\text{def}}{=} q$) as unit. A *guarded command* is defined as

$$(b \to A)q \overset{\text{def}}{=} b \Rightarrow Aq$$

and it follows that

$$g(b \to A) = b \wedge gA \qquad \text{and} \qquad t(b \to A) = b \Rightarrow tA$$

It is easy to see that, as special cases, $\mathsf{true} \to A = A$ and $\mathsf{false} \to A = \mathsf{magic}$. Nondeterministic choice is defined as

$$A_1 \,[]\, \ldots \,[]\, A_m \overset{\text{def}}{=} A_1 \wedge \ldots \wedge A_m$$

Its guard and termination domain can be derived respectively

$$g(A_1 \,[]\, \ldots \,[]\, A_m) = gA_1 \vee \ldots \vee gA_m$$
$$t(A_1 \,[]\, \ldots \,[]\, A_m) = tA_1 \wedge \ldots \wedge tA_m$$

Sometimes it is convenient to use a Hoare triple to indicate a total correctness formula: program S is *totally correct* with respect to precondition p and postcondition q, denoted as $\{p\}\ S\ \{q\}$, is defined by

$$\{p\}\ S\ \{q\} \overset{\text{def}}{=} p \leq S\ q$$

It follows that

$$\{p\}\ b \to S\ \{q\} = \{p \wedge b\}\ S\ \{q\}$$

2.2. (Unfair) action systems

When the semantics of a concurrent system is based on *interleaving* observations, it is equivalent to a nondeterministic sequential system with respect to logical behaviours. An action system $\mathscr{A}$ is a statement of the form

$$[\![\ \mathsf{var}\ x \bullet\ p;\ \mathsf{do}\ A\ \mathsf{od}\ ]\!] : z$$

where, x and z are the tuples of local and global variables respectively, p is the initialisation condition, and A is an action, perhaps a compound one. The action is executed repeatedly, and each iteration is not interrupted, i.e., A is treated as an atomic statement. The initialisation condition is often omitted if it is the constant predicate true.

To allow a concurrent system to be modelled within the action system format, a parallel composition of two action systems must be mapped into another action system. This is straightforward. Two action systems can be composed in parallel if they have the same global but disjoint local variables. Let

$$\mathscr{A}_i \overset{\text{def}}{=} [\![\ \mathsf{var}\ x_i \bullet\ p_i;\ \mathsf{do}\ A_i\ \mathsf{od}\ ]\!] : z$$

then

$$\mathscr{A}_1 \parallel \mathscr{A}_2 \overset{\text{def}}{=} [\![\ \mathsf{var}\ x_1, x_2 \bullet\ p_1 \wedge p_2;\ \mathsf{do}\ A_1\ [\,]\ A_2\ \mathsf{od}\ ]\!] : z$$

The resulting action system has the same global variables, while its local variables are the union, its initialisation condition the conjunction and its action the choice of the respective parts in the two component systems. Sometimes it is necessary to *hide* some global variables, especially after two systems are composed in parallel. This is indicated by

$$[\![\ \mathsf{var}\ l \bullet\ \mathscr{A}\ ]\!]$$

where variables in l are made local.

2.3. Semantics and refinement order of action systems

A *computation* of an action system $\mathscr{A} = [\![\ \mathsf{var}\ x \bullet\ p;\ \mathsf{do}\ A\ \mathsf{od}\ ]\!] : z$ is a finite or infinite sequence of states generated by an execution of the action system. The first state of the computation satisfies the initial condition p, and any state transition is performed by action A (no matter whether A is compound or not, the transition corresponds to the complete execution of it). A finite computation is either *terminated* or *aborted*; the former happens when A is no longer *enabled* and the latter occurs when A is nonterminating. The two kinds of finite computations are distinguished

A computation *induces* a trace, in which local states and all the *stuttering* transitions, which are steps that do not change global states, are deleted. When all the transitions in an infinite computation are stuttering from certain point onwards, the induced trace is considered to be aborted. Aborted computations also induce aborted traces. The semantics of an action system $\mathscr{A}$ is defined as the set of its traces $tr(\mathscr{A})$; this captures the *observable* behaviours of the system.

Action system $\mathscr{C}$ is said to refine action system $\mathscr{A}$, denoted by $\mathscr{A} \sqsubseteq \mathscr{C}$, if for any trace σ of $\mathscr{C}$ either

$$(\exists \sigma' | \sigma' \in tr(\mathscr{A}) . \sigma' \preceq \sigma \wedge aborted(\sigma')), \text{ or}$$
$$(\exists \sigma' | \sigma' \in tr(\mathscr{A}) . \sigma' = \sigma \wedge nonaborted(\sigma))$$

where $\preceq$ is the prefix relation (including equality). For any traces σ and σ' satisfying the above condition, we say that σ' approximates σ; a higher level computation approximates a lower level one if the induced traces have this relation.

2.4. Proving refinement

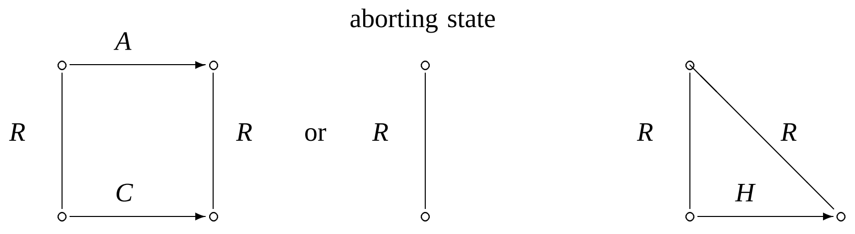
It is difficult to use the semantical definition to directly prove refinement. The effective verification method, known as simulation, reduces the verification of action systems to that of individual actions. Assume that the tuples of abstract and concrete variables are a and c, and the data refinement relation is $R(a, c, z)$ (where z is the tuple of global variables used in both the higher and lower level systems). Simulation is usually divided into so called *forward simulation* and *backward simulation*.

Action A is *forward simulated* by action C under data refinement relation R

$$A \le_R^f C \overset{\text{def}}{=} R(a, c, z) \wedge Aq(a, z) \le C(\lambda(c, z).(\exists a . R(a, c, z) \wedge q(a, z)))(c, z)$$

In this paper, we take the convention that universal quantification is implicitly understood. In the above definition, p, a, c and z are implicitly quantified. Although here the various variables are bounded and consequently it is mathematically correct to use any names for them, we prefer notations consistent with ordinary programming practice, and always use a, c and z to denote the tuples of abstract, concrete and global variables respectively.

The definition of forward simulation implies that for two states coupled by R, if C is aborting in the concrete state, then A is also aborting in the abstract state, and if C is not aborting and there is a transition from C, then either A is aborting in the abstract state or there is also a transition from A such that the two states after the transitions are still coupled by R. Recall that a transition is stuttering if it does not change any global variables. Therefore, the transition from skip is always stuttering. Moreover, if skip $\le_R^f H$, then H is not aborting in any concrete state that is related to an abstract state and its transition is stuttering, and in fact, the next concrete state is related to the same abstract state.



$A \le_R^f C$ $\qquad$ $\mathsf{skip} \le_R^f H$

For two actions $b \to A$ and $d \to C$ where $gA = gC = \mathsf{true}$,

$$b \to A \le_R^f d \to C$$

holds if and only if

$$(i)\ d\,(c,z) \wedge R\,(a,c,z) \;\le\; b\,(a,z)$$
$$(ii)\ R\,(a,c,z) \wedge d\,(c,z) \wedge Aq\,(a,z)$$
$$\le\; C\,(\lambda(c,z).(\exists a.R(a,c,z) \wedge q(a,z)))(c,z)$$

One particularly useful situation in refinement is that the actions of the lower level system form two groups; one group corresponds to actions of the higher level system, and the other group contains only stuttering actions. Therefore, in this paper, we are mainly interested in higher and lower level action systems which respectively have the following forms:

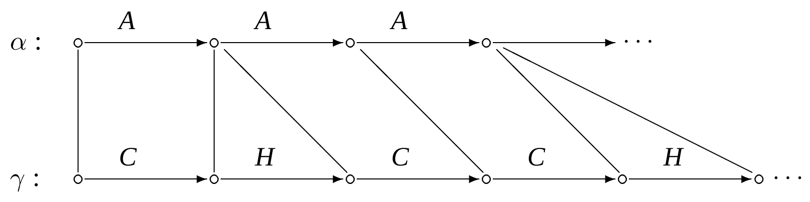
$$\mathscr{A} \overset{\text{def}}{=} [\![\, \mathsf{var}\ a \bullet p; \mathsf{do}\ A\ \mathsf{od}\ ]\!] : z$$
$$\mathscr{C} \overset{\text{def}}{=} [\![\, \mathsf{var}\ c \bullet q; \mathsf{do}\ C\ [\,]\ H\ \mathsf{od}\ ]\!] : z$$

Forward and backward simulations of action systems present respectively a forward way and a backward way of constructing a higher level computation for a given lower level computation such that the former approximates the latter. The existence of such constructions shows that simulations are sound as proof rules of refinement. We next review the forward simulation techniques for unfair action systems. The dual notion of backward simulation will be studied later.

Forward simulation of (unfair) action systems. Data refinement relation $R(a,c,z)$ is said to be a forward simulation between $\mathscr{A}$ and $\mathscr{C}$, denoted by $\mathscr{A} \le_R^f \mathscr{C}$, if

(i) Initialisation: $q(c,z) \le (\exists a.R(a,c,z) \wedge p(a,z))$
(ii) Main actions: $A \le_R^f C$
(iii) Stuttering actions $\mathsf{skip} \le_R^f H$
(iv) Exit condition: $R \wedge \neg(gC \vee gH) \le \neg tA \vee \neg gA$
(v) Internal convergence: $R \wedge tA \le t(\mathsf{do}\ H\ \mathsf{od})$

For any computation $\gamma = \gamma_0\gamma_1...\gamma_i...$ of the lower level system, based on the above conditions, we construct below a computation α of the higher level system such that it approximates γ. The first condition says that there exists an abstract state α_0 which satisfies the initialisation condition of the higher level system and it is related by R with γ_0; α_0 can then be chosen as the first state of α. If γ_0 is the only element of γ or A is aborting in α_0, then let α be the computation containing α_0 one element. Otherwise, depending on whether (γ_0, γ_1) is a transition of C or H, by the second and the third conditions, either there is an abstract state α_1 such that (α_0, α_1) is a transition of A and α_1 is related to γ_1 by R or α_0 is related to γ_1 by R. We repeat this process from γ_1 until either an abstract state in which A is aborting is found or all the concrete states are traversed. The fourth condition guarantees that the terminated state of γ is related either to an aborting or a terminated abstract state. The fifth condition ensures that there are not infinitely many consecutive stuttering transitions in γ unless A is aborting in the last abstract state.



2.5. Fair action systems

The main component of a fair action system is a fair iteration statement. We consider the fair iteration statement of the form

$$\mathscr{A} = \mathsf{do}\ A_1\ [\!]\ A_2\ [\!]\ \ldots\ [\!]\ A_n\ \mathsf{od}$$

where A_i may be associated with fairness assumptions. We consider both *weak* fairness and *strong* fairness, and denote by $WF(\mathscr{A}) \subseteq \{A_1, \ldots, A_n\}$ and $SF(\mathscr{A}) \subseteq \{A_1, \ldots, A_n\}$ the sets of actions associated with weak fairness and strong fairness respectively. A computation of $\mathscr{A}$ is fair if, (1) when any action A_i in $WF(\mathscr{A})$ is continuously enabled from some point onwards, there is a state transition from it executed at a certain stage afterwards, (2) this is so when any action in $SF(\mathscr{A})$ is infinitely often enabled. Without losing the expressive power, we follow the convention assuming that $WF(\mathscr{A}) \cap SF(\mathscr{A}) = \emptyset$.

We do not require all the actions to be treated fairly. To distinguish the three types of actions, we put a label wf before an action if it is associated with weak fairness, and a label sf if it is associated with strong fairness; undecorated actions denote that they have no fairness requirements (the execution of such an action is only guaranteed under the so-called *minimal progress* assumption, i.e., the action will definitely be executed if it is the only enabled action). For an action A, l_A denotes the fairness label of A; by abusing notation a little, we use the same symbol A to denote the program part of the action (i.e., the part without the fairness label). Note that we can group all the unfair actions together and make them into one big unfair action, but there are no direct ways to combine fair actions in general. In our setting, fairness is only associated with top level choice; if action A_i contains further nondeterministic branches, their selection is not subject to any fairness.

For a fair action system $\mathscr{A}$, $tr(\mathscr{A})$ is the set of traces induced from its fair computations. The semantic refinement order remains unchanged.

3. Termination and total correctness of weakly fair iteration statements

As mentioned earlier, termination will be used as a proof condition in showing refinement. Termination is part of total correctness, while the latter in addition requires that the program is abortion free and its terminating states satisfy a postcondition. Let $\mathscr{A}$ be the fair iteration statement

$$\mathsf{do}\ A_1\ []\ A_2\ []\ \ldots\ []\ A_n\ \mathsf{od}$$

We denote that $\mathscr{A}$ is (fairly) terminating with respect to precondition p by

$$\{p\}\ \mathscr{A}$$

3.1. Proof rules

Our formulation is based mainly on the work by Francez [13] as well as Manna and Pnueli [20]. In this section, we only consider weak fairness.

To prove that $\mathscr{A}$ terminates under precondition p, we find a well-founded set W and a family of predicates $I = \{I_w | w \in W\}$. First, we show that if p holds, then there exists a w such that I_w holds. We next prove that execution of any action under I_w leads to I_v with $v \leq w$. Thirdly, we verify that if the iteration is executed sufficiently often then the new index in I_v will be strictly smaller. Since the index set is well-founded, there cannot exist any infinite computations.

Therefore, an important concept is that some actions must decrease the indexes if executed. Such actions are usually said to be *helpful*. However, in general an action is only helpful in some states. Let predicate r characterise the set of states, and define

$$\mathsf{helpful}(I_w, r, A) \overset{\text{def}}{=} \{I_w \wedge r \wedge tA\}\ A\ \{\exists v | v < w.I_v\}$$

Aborting and disabled actions are helpful by the definition; intuitively, it is clear that neither of them can contribute to infinite computations of the iteration statement. Under appropriate conditions, fairness assumption ensures that certain helpful action will eventually be executed, but as an action may only be helpful in some states, we need to be sure that the execution of other actions will either keep the original action helpful or decrease the index. This leads us to define another important concept

$$\mathsf{helpful\_or\_stay}(I_w, r, A) \overset{\text{def}}{=} \{I_w \wedge r \wedge tA\}\ A\ \{(\exists v | v < w.I_v) \vee (I_w \wedge r)\}$$

It is easy to see that $\mathsf{helpful}(I_w, r, A)$ implies $\mathsf{helpful\_or\_stay}(I_w, r, A)$.

Theorem 1. (Proof rule for termination of weakly fair iteration statements). *$\{p\}\ \mathscr{A}$ holds, if there exist a well-founded set W and a family of predicates $I = \{I_w | w \in W\}$ such that the following holds:*

(i) $p \leq (\exists w.I_w)$
(ii) assume $WF(\mathscr{A}) = \{A_{J_1}, \ldots, A_{J_k}\}$, there exist $r_1, \ldots, r_k$ such that

(1) $\mathsf{helpful\_or\_stay}(I_w, r_i, A_j)$*, for any $i = 1, \ldots, k$ and any $j \neq J_i$*

(2) $\mathsf{helpful}(I_w, r_i, A_{J_i})$ *and $I_w \wedge r_i \leq gA_{J_i}$, for any $i = 1, \ldots, k$*

(3) $\mathsf{helpful}(I_w, \neg(r_1 \vee \ldots \vee r_k), A_i)$*, for any $i = 1, \ldots, n$*

We argue the soundness of this rule by showing that a computation will not be 'stuck' at a level w. Any state in a computation satisfies I_w for certain w and in addition one of the following $k + 1$ predicates

$$r_1, \qquad \ldots \qquad r_k, \qquad \neg(r_1 \vee \ldots \vee r_k)$$

If a predicate r_i holds in the state, then it follows from *(ii)_(1)* that execution of any actions other than A_{J_i} will either decrease the index or leave $I_w \wedge r_i$ satisfied. Therefore, if the index has not already been decreased, from *(ii)_(2)* we know that A_{J_i} is helpful and continuously enabled, and subsequently, it will be executed eventually by the fairness assumption, causing the index to be lowered. If none of $r_1, \ldots, r_k$ hold, then *(ii)_(3)* indicates that all actions are helpful and the index is decreased immediately by the next transition.

Note that we do not care at which level w the loop terminates. Some other termination rules in literature require termination to occur when the index is exactly the minimal element 0.

The iteration statement derived in a refinement step, and which we have to prove terminating, may also contain exit commands. Execution of an exit command causes the iteration to stop immediately. The effect of such commands can be coded by assignments to a special variable which once being set to a special value disables all the actions, but the presentation is slightly neater when the command is retained. For our purpose, the exit commands can occur at two places: either the exit command is at the end of an entire action, or an action is composed of two alternatives and the exit command is at the end of one alternative. Modifying the rules to accommodate the exit commands is easy, as only the definitions of helpful and helpful_or_stay need to be slightly extended. Since execution of A;exit leads immediately to termination of the statement, we define

$$\mathsf{helpful}(I_w, r, A;\mathsf{exit}) \overset{\text{def}}{=} \mathsf{helpful\_or\_stay}(I_w, r, A;\mathsf{exit}) \overset{\text{def}}{=} \mathsf{T}$$
$$\mathsf{helpful}(I_w, r, A;\mathsf{exit}\ []\ A') \overset{\text{def}}{=} \mathsf{helpful}(I_w, r, A')$$
$$\mathsf{helpful\_or\_stay}(I_w, r, A;\mathsf{exit}\ []\ A') \overset{\text{def}}{=} \mathsf{helpful\_or\_stay}(I_w, r, A')$$

Theorem 1 is valid for iteration statements with exit commands when the definitions of helpful and helpful_or_stay are defined as above.

Theorem 2. (Proof rule for total correctness of weakly fair iteration statements). $\{p\}\ \mathscr{A}\ \{q\}$*, if there exist a well-founded set* W *and a family of predicates* $I = \{I_w | w \in W\}$ *such that conditions (i) and (ii) in Theorem 1 and the following hold*

(iii) $(\exists w.I_w) \wedge gA_i \leq tA_i, i = 1, \ldots, n$
(iv) *(1)* $(\exists w.I_w) \wedge \neg(gA_1 \vee \ldots \vee gA_n) \leq q$

(2) $\{\exists w.I_w\}\ A_i'\ \{q\}$*, for all* $i = 1, \ldots, n$ *such that* A_i *is of the form* A_i';exit *or* A_i';exit $[]A_i''$

Condition *(iii)* ensures that abortion will not occur. The first and second conditions in *(iv)* guarantee respectively that the postcondition is satisfied in the normal terminating states and the terminating states resulted from the execution of exit commands.

3.2. Examples

Example 1 Consider proving

```
{true}
do
  wf : ($z \geq 150 \rightarrow z := z + 1$;exit [] $z < 100 \rightarrow z := z + 2$)   % action $A_1$
  [] $z := z + 2$                                                                       % action $A_2$
od
```

For brevity, we assume that z ranges over integers in this and the following a few examples. Informally, we can argue about termination like this. The loop can only terminate when the exit command is executed. Started with any initial value, the two actions can only increase the value of z and eventually make it greater than or equal to 150. The first action is then continuously enabled, and therefore due to fairness assumption after a number of executions of the second action, the first action will be taken. Since the second alternative in the first action is disabled, only the first alternative can be taken, leading to the termination of the whole program.

We now formalise this argument. Choose natural numbers as the well founded domain and define

$$I_w \overset{\text{def}}{=} w = max(150 - z, 0)$$

The proof obligations *(i)* and *(ii)* are established as follows.

$$\begin{array}{lll} (i) & (\exists w.I_w) & \{\text{definition of } I_w\} \\ & = (\exists w.w = \mathit{max}(150 - z, 0)) & \{\text{let } w = \mathit{max}(150 - z, 0)\} \\ & = \mathsf{true} & \end{array}$$

(ii) Choose $r_1 \overset{\text{def}}{=} (z \geq 150)$

(1) follows from

$$\begin{array}{ll} & \{I_w\}\ z := z + 2\ \{\exists v | v \leq w.I_v\} \\ = & \{\text{definition of } I_w\} \\ & \{w = \mathit{max}(150 - z, 0)\}\ z := z + 2\ \{\exists v | v \leq w.v = \mathit{max}(150 - z, 0)\} \\ \geq & \{\text{let } v = \mathit{max}(w - 2, 0)\} \\ & (w = \mathit{max}(150 - z, 0)) \leq (\mathit{max}(w - 2, 0) = \mathit{max}(150 - z - 2, 0)) \\ \geq & \{\text{case analysis: } z \leq 150 \text{ and } z > 150\} \\ & \mathsf{T} \end{array}$$

and $\{z \geq 150\}\ z := z + 2\ \{z \geq 150\}$

(2)

$$\begin{array}{ll} & \mathsf{helpful}(I_w, r_1,\ (z \geq 150 \rightarrow z := z + 1; \mathsf{exit}\ []\ z < 100 \rightarrow z := z + 2)) \\ = & \{\text{definitions of } \mathsf{helpful} \text{ and } r_1\} \\ & \mathsf{helpful}(I_w, z \geq 150,\ (z < 100 \rightarrow z := z + 2)) \\ = & \{\text{definition of } \mathsf{helpful}\} \\ & \{I_w \wedge z \geq 150 \wedge z < 100\}\ z := z + 2\ \{\exists v | v < w.I_v\} \\ = & \{\mathsf{false}\}\ z := z + 2\ \{\exists v | v < w.I_v\} \\ = & \mathsf{T} \end{array}$$

$$\begin{array}{ll} & (I_w \wedge r_1) \leq g(z \geq 150 \rightarrow z := z + 1; \mathsf{exit}\ []\ z < 100 \rightarrow z := z + 2)) \\ = & \{\text{definitions of guard and } r_1\} \\ & (I_w \wedge z \geq 150) \leq (z \geq 150 \vee z < 100) \\ = & \mathsf{T} \end{array}$$

(3) We first check A_2

$$\begin{array}{ll} & \mathsf{helpful}(I_w, \neg r_1,\ z := z + 2) \\ = & \{\text{definitions of } \mathsf{helpful} \text{ and } r_1\} \\ & \{I_w \wedge z < 150\}\ z := z + 2\ \{\exists v | v < w.I_v\} \\ = & \{\text{definition of } I_w\} \\ & \{w = \mathit{max}(150 - z, 0) \wedge z < 150\} \\ & \qquad z := z + 2 \\ & \{\exists v | v < w.v = \mathit{max}(150 - z, 0)\} \\ \geq & \{\text{let } v = \mathit{max}(w - 2, 0)\} \\ & (w = 150 - z) \leq (\mathit{max}(w - 2, 0) = \mathit{max}(150 - z - 2, 0)) \\ = & \mathsf{T} \end{array}$$

Next we verify A_1

$$\begin{array}{ll} & \mathsf{helpful}(I_w, \neg r_1,\ (z \geq 150 \rightarrow z := z + 1; \mathsf{exit}\ []\ z < 100 \rightarrow z := z + 2)) \\ = & \{\text{definitions of } \mathsf{helpful} \text{ and } r_1\} \\ & \mathsf{helpful}(I_w, z < 150,\ (z < 100 \rightarrow z := z + 2)) \\ = & \{\text{definition of } \mathsf{helpful}\} \\ & \{I_w \wedge z < 150 \wedge z < 100\}\ z := z + 2\ \{\exists v | v < w . I_v\} \\ = & \{\text{similar to the proof for } A_2\} \\ & \mathsf{T} \end{array}$$

Example 2 Consider the loop

```
do
  wf : ($z \geq 150 \rightarrow z := z + 1$;exit [] $z > 100 \rightarrow z := z + 2$)   % action $A_1$
 [] $z := z + 2$                                                          % action $A_2$
od
```

This program is the same as the previous one, except that one guard becomes $z > 100$, but now it does not terminate. Should we try to carry over the previous proof, we will notice that the last part is no longer valid. Indeed, when $z \geq 150$ holds, the alternative without the exit command can be executed infinitely instead.

3.3. Termination under continuous condition

So far, we have discussed proving termination under preconditions. However, the termination arisen in verifying refinement is weaker in that we can assume more than just preconditions. The following termination notion is useful when there are actions associated with weak fairness in the higher level system. Iteration statement $\mathscr{A}$ terminates under a continuous condition p, denoted as $[p]\,\mathscr{A}$, is defined by

$[p]\,\mathscr{A}$ iff there are no infinite computations from $\mathscr{A}$ such that p always holds

Obviously, if $\{p\}\mathscr{A}$ holds, then $[p]\mathscr{A}$ holds also. This means that we can use the same termination rule to prove $[p]\mathscr{A}$. But there are many cases where $[p]\mathscr{A}$ holds, while $\{p\}\mathscr{A}$ does not. Consider the program do $z := z - 2$ od for example. It does not terminate under the precondition $z > 100$, but $[z > 100]$ do $z := z - 2$ od holds, because an infinite execution of the loop will cause the value of z to be arbitrarily small, and hence $(z > 100)$ cannot hold continuously.

It is easy to reduce the proof of $[p]\mathscr{A}$ to that of ordinary fair termination: if $\mathscr{A}$ is the iteration statement do A_1 [] A_2 [] ... [] A_n od, then

$$[p]\,\mathscr{A} \quad \text{iff} \quad \{\mathsf{true}\}\ \mathsf{do}\ p \rightarrow A_1\ []\ p \rightarrow A_2\ []\ \ldots\ []\ p \rightarrow A_n\ \mathsf{od}$$

Example 3 Returning to the example program, to prove

$$[z > 100]\ \mathsf{do}\ z := z - 2\ \mathsf{od}$$

we only need to show

$$\{\mathsf{true}\}\ \mathsf{do}\ z > 100 \to z := z - 2\ \mathsf{od}$$

which is trivial.

4. Forward simulation of weakly fair action systems

We now come to our major concern: proving refinement. Again, we restrict ourselves to weak fairness first, and consider the case that the higher and lower level action systems are respectively of the form:

$$\mathscr{A} \overset{\text{def}}{=} [\![\ \mathsf{var}\ a \bullet p; \mathsf{do}\ A\ \mathsf{od}\ ]\!] : z$$
$$\mathscr{C} \overset{\text{def}}{=} [\![\ \mathsf{var}\ c \bullet q; \mathsf{do}\ C\ []\ H\ \mathsf{od}\ ]\!] : z$$

where

$$A \overset{\text{def}}{=} A_1\ []\ A_2\ []\ \dots\ []\ A_n$$
$$C \overset{\text{def}}{=} C_1\ []\ C_2\ []\ \dots\ []\ C_m$$
$$H \overset{\text{def}}{=} H_1\ []\ H_2\ []\ \dots\ []\ H_k$$

and fairness may be associated with A_i, C_i and H_i. As reviewed earlier, simulation methods for unfair action systems reduce the verification of the complete action systems to the verification of individual actions. We wish to extend the approach to verify fair action systems. For the unfair case, it is enough that the compound action A is simulated by the compound action C, which holds if and only if A is simulated by each C_i. However, this is not sufficient if there are fairness assumptions in $\mathscr{A}$. To formulate the fairness condition, we decompose C_i into $C_{i,1}\ []\ C_{i,2}\ []\ \dots\ []\ C_{i,n}$ such that

$$A_j \leq_R^f C_{i,j}$$

i.e., action A_j is (forward) simulated by $C_{i,j}$ with respect to data refinement relation R. This results in the following decomposition matrix,

$$\begin{array}{c|cccc} & A_1 & A_2 & \dots & A_n \\ \hline C_1 & C_{1,1} & C_{1,2} & \dots & C_{1,n} \\ C_2 & C_{2,1} & C_{2,2} & \dots & C_{2,n} \\ \dots & & & & \\ C_m & C_{m,1} & C_{m,2} & \dots & C_{m,n} \end{array}$$

in which every lower level action simulates the higher level action in the same column. The decomposition $C_i = C_{i,1}\ []\ C_{i,2}\ []\ \dots\ []\ C_{i,n}$ can be verified using traditional refinement calculus.

We use $C_{-,i}$ to denote an arbitrary lower level subaction in column i. Roughly speaking, if a higher level action A_i must be executed due to a fairness assumption, then we need to guarantee that a subaction $C_{-,i}$ will also be executed in the lower level system. However, fairness is not directly associated with such subactions, and therefore we must prove that all the subactions in other columns will not be infinitely and exclusively executed.

Forward simulation of weakly fair action systems. Data refinement relation $R(a, c, z)$ is said to be a forward simulation between $\mathscr{A}$ and $\mathscr{C}$, denoted by $\mathscr{A} \leq_R^f \mathscr{C}$, if

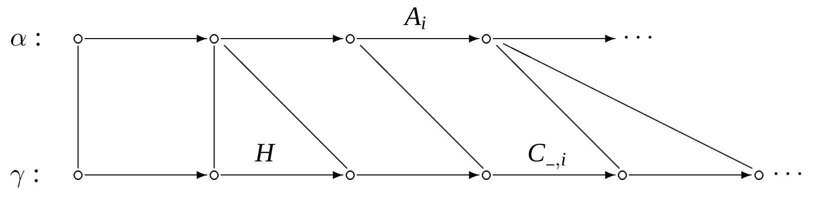
(i) Initialisation: $q(c, z) \leq (\exists a . R(a, c, z) \wedge p(a, z))$
(ii) Main actions: $A_i \leq_R^f C_{j,i}$, for any $i = 1, \ldots, n$ and any $j = 1, \ldots, m$
(iii) Stuttering actions $\mathsf{skip} \leq_R^f H_i$, for any $i = 1, \ldots, k$
(iv) Exit condition: $R \wedge \neg(gC \vee gH) \leq \neg tA \vee \neg gA$
(v) Internal convergence: $\{\exists a . R(a, c, z) \wedge tA(a, z)\}\, D$, where

$$D = \mathsf{do}\ l_{C_1} : C_1; \mathsf{exit}\ []\ l_{C_2} : C_2; \mathsf{exit}\ []\ \ldots\ []\ l_{C_n} : C_n; \mathsf{exit}$$
$$[]\ H_1\ []\ \ldots\ []\ H_k$$
$$\mathsf{od}$$

(vi) Fairness condition: $[\exists a . R(a, c, z) \wedge gA_i(a, z)]\ \mathscr{C}^i$, for any $A_i \in WF(\mathscr{A})$, where

$$\mathscr{C}^i = \mathsf{do}\ C_1^i\ []\ C_2^i\ []\ \ldots\ []\ C_n^i\ []\ H_1 \ldots\ []\ H_k\ \mathsf{od}, \text{ in which}$$
$$C_j^i = l_{C_j} : (C_{j,1}\ []\ \ldots\ []\ C_{j,i-1}\ []\ C_{j,i}; \mathsf{exit}\ []\ C_{j,i+1}\ []\ \ldots\ []\ C_{j,n})$$

The first four conditions are either the same or imply directly the corresponding conditions in forward simulation of unfair systems. The internal convergence condition is still used to ensure that there are not infinitely many consecutive extra stuttering transitions unless A is aborting; now it may happen that some main actions must be executed due to the fairness assumptions, and the new derived iteration statement D takes into account this situation. The last condition is used to guarantee the fairness requirement. It implies that when action A_i is continuously enabled the lower level computation is either finite or has a $C_{-,i}$ transition.



For any fair computation γ of $\mathscr{C}$, it is known from simulation of unfair systems that one can construct a computation α of $\mathscr{A}$ such that it approximates

γ. In particular, an A_i transition is chosen in α to match a $C_{\_,i}$ transition in γ. The resulting computation is fair, for otherwise there must exist a higher level action in $WF(\mathscr{A})$, say A_k, whose enableness condition is satisfied but is never taken from certain point onwards. The fairness condition implies that a $C_{\_,k}$ transition must be present in γ, hence a contradiction.

The fairness condition is void when there are no fairness assumptions in the higher level system. If there are no fairness assumptions in the lower level system, internal convergence condition is also equivalent to the one in the unfair case. Therefore, simulation of unfair action systems is a special case of simulation of fair action systems.

Example 4 Let two action systems be

```
$\mathscr{A} \overset{\text{def}}{=} [\![$ do
     wf :  $z := z + 1$                                    % action $A_1$
    [] $z := z + 2$                                       % action $A_2$
   od $]\!]$ :  $z$
$\mathscr{C} \overset{\text{def}}{=} [\![$ do
     wf :  ($z \geq 150 \rightarrow z := z + 1$ [] $z < 100 \rightarrow z := z + 2$) % $C_{1,1}$ [] $C_{1,2}$
    [] $z := z + 2$                                       % $C_{2,2}$
   od $]\!]$ :  $z$
```

The comments in the program denote the names of actions. Subactions are omitted if they are magic (in this example, $C_{2,1}$ = magic). This describes a decomposition, and for the example, we have the following corresponding matrix

$z := z + 1$	$z := z + 2$
$z \geq 150 \rightarrow z := z + 1$	$z < 100 \rightarrow z := z + 2$
magic	$z := z + 2$

In this and the next few examples, conditions (i),(iii),(iv) and (v) are trivial to prove, and therefore we only concentrate on conditions (ii) and (vi). By choosing $R \overset{\text{def}}{=}$ true as the data refinement relation, the simulation of the corresponding actions is easy to show. The fairness condition is implied by the following termination formula:

```
{true} do
     wf :  ($z \geq 150 \rightarrow z := z + 1$;exit [] $z < 100 \rightarrow z := z + 2$)
    [] $z := z + 2$
   od
```

which we have already established in Example 1.

Example 5 Consider next the following action systems

```
$\mathscr{A} \overset{\text{def}}{=} [\![$ do
        wf : $z := z + 1$                                          % action $A_1$
        [] $z := z + 2$                                             % action $A_2$
       od $]\!]$ : $z$
$\mathscr{C} \overset{\text{def}}{=} [\![$ do
        wf : $(z \geq 150 \to z := z + 1$ [] $z > 100 \to z := z + 2)$ % $C_{1,1}$ [] $C_{1,2}$
        [] $z := z + 2$                                             % $C_{2,2}$
       od $]\!]$ : $z$
```

The programs are the same as the previous ones, except that one guard in $\mathscr{C}$ becomes $z > 100$, but now refinement does not hold. Indeed, when $z \geq 150$ holds, the second alternative in the first action of $\mathscr{C}$ can be executed infinitely instead, and as a result, z is only increased by 2. Should we try to carry over the previous proof, we will then have to prove termination of the following loop:

```
do
  wf : $(z \geq 150 \to z := z + 1$;exit [] $z > 100 \to z := z + 2)$
  [] $z := z + 2$
od
```

As we have already discussed in Example 2, this is not the case.

Example 6 Now consider the refinement of the following two systems:

```
$\mathscr{A} \overset{\text{def}}{=} [\![$ do
        wf : $z > 100 \to z := z + 1$   % action $A_1$
        [] $z := z - 2$                  % action $A_2$
       od $]\!]$ : $z$
$\mathscr{C} \overset{\text{def}}{=} [\![$ do
        $z := z - 2$                     % action $C_{1,2}$
       od $]\!]$ : $z$
```

Let data refinement relation be $R \overset{\text{def}}{=}$ true. The fairness condition is

$[z > 100]$ do $z := z - 2$ od

and it is proved in Example 3.

5. Including strong fairness

In the previous sections, we have considered weak fairness. Strong fairness is also a useful notion. In the general case, an action system may have a mixture of weak and strong fairness. In accordance with the order of presentation in the previous sections, we first extend the normal termination and total correctness rules, then investigate an additional termination notion which is useful in proving refinement, and finally we extend the refinement rule.

5.1. Termination and total correctness of fair iteration statements

The following is the extended rule for proving termination.

Theorem 3. (Proof rule for termination of fair iteration statements). $\{p\}\ \mathscr{A}$, *if there exist a well-founded set W and a family of predicates $I = \{I_w | w \in W\}$ such that the following holds:*

(i) $p \leq (\exists w.I_w)$
(ii) assume $WF(\mathscr{A}) = \{A_{J_1}, \ldots, A_{J_k}\}$ *and* $SF(\mathscr{A}) = \{A_{S_1}, \ldots, A_{S_l}\}$, *there exist* $r_1, \ldots, r_k, r_{k+1}, \ldots, r_{k+l}$ *such that*

(1) helpful_or_stay(I_w, r_i, A_j), *for any* $i = 1, \ldots, k$ *and any* $j \neq J_i$
helpful_or_stay(I_w, r_{k+i}, A_j), *for any* $i = 1, \ldots, l$ *and any* $j \neq S_i$

(2) helpful(I_w, r_i, A_{J_i}) *and* $I_w \wedge r_i \leq gA_{J_i}$, *for any* $i = 1, \ldots, k$

(3) helpful(I_w, r_{k+i}, A_{S_i}) *and*
$\{$true$\}$ do $[]\, j | j \neq S_i .\ l_{A_j} : (\neg gA_{S_i} \wedge I_w \wedge r_{k+i} \rightarrow A_j)$ od,
for any $i = 1, \ldots, l$
(4) helpful$(I_w, \neg(r_1 \vee \ldots \vee r_k \vee r_{k+1} \ldots \vee r_{k+l}), A_i)$, *for any* $i = 1, \ldots, n$

We argue its soundness as follows. The case that an intermediate state satisfies r_i, where $1 \leq i \leq k$, or $\neg(r_1 \vee \ldots \vee r_k \vee r_{k+1} \ldots \vee r_{k+l})$, is the same as in Theorem 1. If the state satisfies r_{k+i}, where $1 \leq i \leq l$, then it follows from conditions *(ii)_(1)* and *(ii)_(3)* that execution of any action other than A_{S_i} will either decrease the index or cause A_{S_i} enabled eventually. This implies that the iteration will not be 'stuck' at one level. For otherwise, A_{S_i} will be enabled after a finite number of steps, and this will happen infinitely often. Then the associated strong fairness ensures that A_{S_i} will be executed.

Note although condition *(ii)_(3)* is termination of yet another iteration statement, the new program has one action less and therefore the proof process will come to an end.

Theorem 4. (Proof rule for total correctness of fair iteration statements). $\{p\}\ \mathscr{A}\ \{q\}$, *if there exist a well-founded set W and a family of predicates $I = \{I_w | w \in W\}$ such that conditions (i), (ii) in Theorem 3 and the following hold*

(iii) $(\exists w.I_w) \wedge gA_i \leq tA_i$, *for any* $i = 1, \ldots, n$
(iv) (1) $(\exists w.I_w) \wedge \neg(gA_1 \vee \ldots \vee gA_n) \leq q$
(2) $\{\exists w.I_w\}\ A_i'\ \{q\}$, *if A_i is of the form* A_i';exit *or* A_i';exit $[]A_i''$

The proof rules for weakly fair iteration statements are special cases where $l = 0$.

Example 7 Consider a program with strong fairness

```
do
  sf : ($z \geq 150 \wedge t \rightarrow z := z + 1$;exit [] $z < 100 \rightarrow z := z + 2$)  % action $A_1$
  [] $z, t := z + 2, \neg t$                                                             % action $A_2$
od
```

This program terminates. Although the alternative with the exit command is no longer continuously enabled, the fairness notion is now strongly fair, and as the result, the command will be executed eventually. For a more rigorous proof, we can use the same family of predicates $\{I_w\}$ as in Example 1

$$I_w \overset{\text{def}}{=} w = max(150 - z, 0)$$

and in fact a large part of the previous argument. Only when $z \geq 150$, we turn to the condition associated with strong fairness instead. Let $r_1 \overset{\text{def}}{=} (z \geq 150)$. We need to prove

$$\{\mathsf{true}\}\ \mathsf{do}\ \neg gA_1 \wedge I_w \wedge z \geq 150 \rightarrow z, t := z + 2, \neg t\ \mathsf{od}$$

where $gA_1 = (z \geq 150 \wedge t) \vee z < 100$. Note that having only one action, this program is simpler than the original one, and indeed its termination is trivial to prove.

5.2. Termination under infinitely often condition

The really useful termination notion in proving refinement when there are actions associated with strong fairness is as follows:

$\langle p\rangle\ \mathscr{A}$ iff there are no infinite computations from $\mathscr{A}$ such that p holds initially and infinitely often after

It is easy to see that $\{p\}\ \mathscr{A}$ implies $\langle p\rangle\ \mathscr{A}$, but the other direction does not hold. For example, $\langle z > 100\rangle$ do $z := z - 2$ od holds but $\{z > 100\}$ do $z := z - 2$ od does not. It can also be noted that $\langle p\rangle\ \mathscr{A}$ implies $[p]\ \mathscr{A}$, but the other way is not true. One such example is that $[z > 100 \wedge t]$ do $z := z - 2$ [] $(z := z + 1; t := \neg t)$ od holds but $\langle z > 100 \wedge t\rangle$ do $z := z - 2$ [] $(z := z + 1; t := \neg t)$ od does not.

Verifying $\langle p\rangle\ \mathscr{A}$ is rather difficult, and in fact, the property expressed by $\langle p\rangle\ \mathscr{A}$ falls into the most general class in temporal logic, the so called *reactivity*. But looking at the general proof rule for reactivity in [20], we notice that it is still unnecessary to assume real knowledge of temporal logic, because most premises are given in Hoare logic style and all the temporal operators occur in the same formula [1]. Therefore, if we rewrite it by a new correctness formula we end up with a proof rule for it without mentioning any temporal operators. More precisely, define

$\{p\}\langle r\rangle\ \mathscr{A}\ \langle q\rangle$ iff under the precondition p, a computation of $\mathscr{A}$ is either finite, or if r holds infinitely often then q holds eventually

To take into account of the 'goal' q, we extend the definitions of helpful and helpful_or_stay. More precisely, for an ordinary action A, let

$$\mathsf{helpful}(I_w, r, q, A) \overset{\text{def}}{=} \{I_w \wedge r \wedge tA\}\ A\ \{(\exists v | v < w.I_v) \vee q\}$$

$$\mathsf{helpful\_or\_stay}(I_w, r, q, A)$$
$$\overset{\text{def}}{=} \{I_w \wedge r \wedge tA\}\ A\ \{(\exists v | v < w.I_v) \vee q \vee (I_w \wedge r)\}$$

[1] The actual form of the temporal formula is $p \wedge \Box \Diamond r \Rightarrow \Diamond q$

The incremental definitions of helpful and helpful_or_stay for actions with exit commands remain unchanged (except for the parameters). Note that the new definitions are conservative extensions of the old ones, as the latter can be considered as special cases of the former with q = false. The following is an alternative formulation of the reactivity rule in [20].

Theorem 5. (Proof rule for reactivity of fair iteration statements). *$\{p\}\langle r\rangle\ \mathscr{A}\ \langle q\rangle$, if there exist a well-founded set W and a family of predicates $I = \{I_w | w \in W\}$ such that the following holds:*

(i) $p \Rightarrow ((\exists w.I_w) \vee q)$

(ii) assume $WF(\mathscr{A}) = \{A_{J_1}, \ldots, A_{J_k}\}$ and $SF(\mathscr{A}) = \{A_{S_1}, \ldots, A_{S_l}\}$, there exist $r_1, \ldots, r_k, r_{k+1}, \ldots, r_{k+l}$ such that

(1) helpful_or_stay(I_w, r_i, q, A_j), *for any* $i = 1, \ldots, k$ *and any* $j \neq J_i$
helpful_or_stay(I_w, r_{k+i}, q, A_j), *for any* $i = 1, \ldots, l$ *and any* $j \neq S_i$

(2) helpful(I_w, r_i, q, A_{J_i}) *and* $I_w \wedge r_i \leq gA_{J_i}$, *for any* $i = 1, \ldots, k$

(3) helpful$(I_w, r_{k+i}, q, A_{S_i})$ *and*
$\{I_w \wedge r_{k+i}\}\langle r\rangle$ do $[]\, j | j \neq S_i\, .\, A_j$ od $\langle q \vee gA_{S_i} \vee \neg I_w\rangle$
for any $i = 1, \ldots, l$

(4) $\{I_w \wedge \neg(r_1 \vee \ldots \vee r_k \vee r_{k+1} \ldots \vee r_{k+l}) \wedge tA_i\}\ A_i\ \{(\exists v | v \leq w.I_v) \vee q\}$
helpful$(I_w, \neg(r_1 \vee \ldots \vee r_k \vee r_{k+1} \ldots \vee r_{k+l}) \wedge r, q, A_i)$
for any $i = 1, \ldots, n$

The second condition of *(ii)_(3)* is similar to the one in the termination rule: eventually, the execution of actions other than A_{S_i} either leads to the goal q, or A_{S_i} enabled, or a change in the index. The change in *(ii)_(4)* reflects the extra assumption that predicate r holds infinitely often, and the second condition can be interpreted as saying that every occurrence of r brings us closer to the goal q. Termination formula

$$\langle p\rangle\ \mathscr{A}$$

can be expressed as

$$\{p\}\langle p\rangle\ \mathscr{A}\ \langle\neg(gA_1 \vee \ldots \vee gA_n)\rangle$$

5.3. Weak invariants

The above general rule is often difficult to use. Fortunately, in many practical cases, one can design special rules which are easier to use albeit incomplete. Below we introduce one special rule. For a program $\mathscr{A}$, a *weak invariant* is a predicate which holds eventually in any infinite computations of $\mathscr{A}$ and remains so afterwards. Our interest in weak invariants is due to the following property: for an iteration statement $\mathscr{A}$, if q is a weak invariant of it, then

$$\langle p\rangle\ \mathscr{A} \quad \text{if} \quad \{p \wedge q\}\ \mathscr{A}$$

We prove this by showing that if $\langle p\rangle\,\mathscr{A}$ does not hold, then $\{p \wedge q\}\,\mathscr{A}$ does not hold either. If $\langle p\rangle\,\mathscr{A}$ does not hold, then there exists an infinite computation such that p holds infinitely often. Since q is a weak invariant, it holds after a finite number of steps, and therefore after at most some other finite steps, both p and q hold. The suffix of the original computation from that position onwards is also an infinite computation of $\mathscr{A}$, and consequently, $\{p \wedge q\}\,\mathscr{A}$ does not hold either.

Actually, termination is a special case where the weak invariant is false, and the rule for fair termination can be easily extended to prove weak invariants.

Theorem 6 (Proof rule for weak invariants). *q is a weak invariant of iteration statement $\mathscr{A}$ under precondition p, if there exist a well-founded set W and a family of predicates $I = \{I_w | w \in W\}$ such that the following holds:*

(i) $p \Rightarrow ((\exists w.I_w) \vee q)$
(ii) *same as in Theorem 3 except* $\mathsf{helpful}(I_w, r_i, A_j)$, $\mathsf{helpful\_or\_stay}(I_w, r_i, A_j)$ *etc. are replaced by* $\mathsf{helpful}(I_w, r_i, q, A_j)$ *and* $\mathsf{helpful\_or\_stay}(I_w, r_i, q, A_j)$ *etc.*
(iii) $\{(\exists w.I_w) \wedge q\}\ A_i\ \{q\}$*, for any* $i = 1, \ldots, n$

The last condition guarantees that, in a computation of $\mathscr{A}$, once q holds, it remains so afterwards. Sometimes this condition does not hold even when q is indeed a weak invariant, because q may become false again in the first few occurrences. In this case, one can first find a predicate q_1 which is set to false until q is stabilised, and prove instead that $q_1 \wedge q$ is a weak invariant.

Example 8 Consider proving

$$\langle z > 100\rangle\ \mathsf{do}\ z := z - 2\ \mathsf{od}$$

We first show that $z < 100$ is a weak invariant. This is straightforward, and we shall not present the details here. The second step is to show

$$\{z > 100 \wedge z < 100\}\ \mathsf{do}\ z := z - 2\ \mathsf{od}$$

which is trivial, since the precondition is equivalent to false.

5.4. Refinement of fair action systems (general case)

It is easy to extend the simulation rules to general fair action systems: all the proof conditions of Section 4 remain the same except the fairness condition has to be modified to take into account of strong fairness. Suppose the action systems and decomposition matrix are the same as in the previous section.

Forward simulation of fair action systems. Data refinement relation $R(a, c, z)$ is said to be a forward simulation between $\mathscr{A}$ and $\mathscr{C}$, denoted by $\mathscr{A} \leq_R^f \mathscr{C}$, if

(i) Initialisation: $q(c, z) \leq (\exists a.R(a, c, z) \wedge p(a, z))$

(ii) Main actions: $A_i \leq^f_R C_{j,i}$, for any $i = 1, \ldots, n$, $j = 1, \ldots, m$
(iii) Stuttering actions skip $\leq^f_R H_i$, for any $i = 1, \ldots, k$
(iv) Exit condition: $R \wedge \neg(gC \vee gH) \leq \neg tA \vee \neg gA$
(v) Internal convergence: $\{\exists a.R(a,c,z) \wedge tA(a,z)\}\ D$, where

```
    D = do l_{C_1} : C_1;exit [] l_{C_2} : C_2;exit [] ... [] l_{C_n} : C_n;exit
                 [] H_1 [] ... [] H_k
        od
```

(vi) Fairness condition: $[\exists a.R(a,c,z) \wedge gA_i(a,z)]\ \mathscr{C}^i$ holds if $A_i \in WF(\mathscr{A})$, and $\langle \exists a.R(a,c,z) \wedge gA_i(a,z)\rangle\ \mathscr{C}^i$ holds if $A_i \in SF(\mathscr{A})$, for $i = 1, \ldots, n$, where

$\mathscr{C}^i$ = do C^i_1 [] C^i_2 [] ... [] C^i_n [] H_1 ... [] H_k od, in which
$C^i_j = l_{C_j}$: ($C_{j,1}$ [] ... [] $C_{j,i-1}$ [] $C_{j,i}$;exit [] $C_{j,i+1}$ [] ... [] $C_{j,n}$)

Example 9 Let two action systems be

```
 𝒜 ≝ [[ do
         sf : $z := z + 1$                                     % action $A_1$
        [] $z := z + 2$                                        % action $A_2$
       od ]] : $z$
 𝒞 ≝ [[ var $t \bullet t = T$;
       do
        sf : ($z \geq 150 \wedge t \rightarrow z := z + 1$ [] $z < 100 \rightarrow z := z + 2$) % $C_{1,1}$ [] $C_{1,2}$
        [] $z, t := z + 2, \neg t$                             % $C_{2,2}$
       od ]] : $z$
```

The simulation of the corresponding actions is obvious, and the data refinement relation can be chosen as $R \overset{\text{def}}{=}$ true. The fairness condition is implied by the following termination formula:

```
   {true} do
            sf : ($z \geq 150 \wedge t \rightarrow z := z + 1$;exit [] $z < 100 \rightarrow z := z + 2$)
           [] $z, t := z + 2, \neg t$
          od
```

which is proved in Example 7.

Example 10 Now consider the refinement of the following two systems:

```
   𝒜 ≝ [[ do
           sf : $z > 100 \rightarrow z := z + 1$   % action $A_1$
          [] $z := z - 2$                   % action $A_2$
          od ]] : $z$
   𝒞 ≝ [[ do
           $z := z - 2$                     % action $C_{1,2}$
          od ]] : $z$
```

Let R be true. The fairness condition is

$$\langle z > 100 \rangle \text{ do } z := z - 2 \text{ od}$$

which is proved in Example 8.

5.5. A special case

One useful case in practice is that each higher level action corresponds exactly to one lower level action. The remaining lower level actions are stuttering ones. Since this situation occurs often, it is useful to derive a special rule. Let two action systems be

$$\mathscr{A} \overset{\text{def}}{=} [\![\text{ var } a \bullet p; \text{do } A \text{ od }]\!] : z$$
$$\mathscr{C} \overset{\text{def}}{=} [\![\text{ var } c \bullet q; \text{do } C \ [\,] \ H \text{ od }]\!] : z$$

where

$$A \overset{\text{def}}{=} A_1 \ [\,] \ A_2 \ [\,] \ \ldots \ [\,] \ A_n$$
$$C \overset{\text{def}}{=} C_1 \ [\,] \ C_2 \ [\,] \ \ldots \ [\,] \ C_n$$
$$H \overset{\text{def}}{=} H_1 \ [\,] \ \ldots \ [\,] \ H_k$$

A special rule for proving forward simulation of fair action systems. $\mathscr{A} \leq_R^f \mathscr{C}$ if

(i) Initialisation: $q(c,z) \leq (\exists a . R(a,c,z) \wedge p(a,z))$
(ii) Main actions: $A_i \leq_R^f C_i$, for any $i = 1, \ldots, n$
(iii) Stuttering actions: skip $\leq_R^f H_i$, for any $i = 1, \ldots, k$
(iv) Exit condition: $R \wedge \neg(gC \vee gH) \leq \neg tA \vee \neg gA$
(v) Internal convergence: $\{\exists a . R(a,c,z) \wedge tA(a,z)\} D$, where

$$D = \text{do } l_{C_1} : C_1; \text{exit } [\,] \ l_{C_2} : C_2; \text{exit } [\,] \ \ldots \ [\,] \ l_{C_n} : C_n; \text{exit } [\,] \ H \text{ od}$$

(vi) Fairness condition: $[\exists a . R(a,c,z) \wedge gA_i(a,z)] \ \mathscr{C}^i$ holds if $A_i \in WF(\mathscr{A})$, and $\langle \exists a . R(a,c,z) \wedge gA_i(a,z) \rangle \ \mathscr{C}^i$ holds if $A_i \in SF(\mathscr{A})$, for $i = 1, \ldots, n$, where

$$\mathscr{C}^i = \text{do } C_1 \ [\,] \ \ldots \ [\,] \ C_{i-1} \ [\,] \ l_{C_i} : C_i; \text{exit } [\,] \ C_{i+1} \ [\,] \ \ldots \ [\,] \ C_n \ [\,] \ H \text{ od}$$

6. A case study

As a more advanced example, we consider the refinement of a mutual exclusion algorithm, studied earlier in a temporal logic framework by Kesten, Manna and Pnueli [17]. Let CS_i be a flag denoting whether or not process i is in its critical section by values 1 or 0 respectively. Mutual exclusion requires that the two processes are not in their critical sections at the same time. Let pc_i^A be the program counter of process i, initially set to 1. An abstract mutual exclusion algorithm can be described as follows:

$$\text{var } pc_1^A, pc_2^A \bullet \ (pc_1^A = pc_2^A = 1) \wedge (CS_1 = CS_2 = 0);$$

```
do
  $(A_1^1)$ wf : $pc_1^A = 1$
                    $\rightarrow CS_1 := 0; pc_1^A := 2$
 [] $(A_1^2)$ sf : $pc_1^A = 2 \wedge CS_2 \neq 1$
                    $\rightarrow CS_1 := 1; pc_1^A := 1$
od : $CS_1, CS_2$
```

```
do
  $(A_2^1)$ wf : $pc_2^A = 1$
                    $\rightarrow CS_2 := 0; pc_2^A := 2$
 [] $(A_2^2)$ sf : $pc_2^A = 2 \wedge CS_1 \neq 1$
                    $\rightarrow CS_2 := 1; pc_2^A := 1$
od : $CS_1, CS_2$
```

Since one process can only enter its critical section when the other process is not in a critical section, mutual exclusion property is obviously satisfied. Strong fairness ensures that one process will enter the critical section if the other process leaves its critical section infinitely often. Note that weak fairness cannot guarantee this, since it does not rule out the possibility that one process waits while the other process enters and exits the critical section all the time. There are two reasons that this abstract algorithm should be refined: first, CS_1 and CS_2 are logical variables used to express specifications and hence are not like program variables (the latter can be tested, for example); second, although strong fairness ensures that each process will eventually enter its critical section, one process may have to wait for an arbitrarily long time if the other process is very fast and enters the critical section again before the first process finishes the boolean test. The next action system models the well-known Peterson's algorithm which overcomes these problems

$$\text{var } pc_1^C, pc_2^C, y_1, y_2, s \bullet (pc_1^C = pc_2^C = 1) \wedge (CS_1 = CS_2 = 0) \wedge s = 1;$$

```
do
  $(C_1^1)$ wf : $pc_1^C = 1 \rightarrow$
                  $(CS_1, y_1) := (0, \mathsf{F})$;
                  $pc_1^C := 2$
 [] $(H_1)$ wf : $pc_1^C = 2 \rightarrow$
                  $(y_1, s) := (\mathsf{T}, 1)$;
                  $pc_1^C := 3$
 [] $(C_1^2)$   $pc_1^C = 3 \wedge (\neg y_2 \vee s = 2)$
                  $\rightarrow CS_1 := 1; pc_1^C := 1$
od : $CS_1, CS_2$
```

```
do
  $(C_2^1)$ wf : $pc_2^C = 1 \rightarrow$
                  $(CS_2, y_2) := (0, \mathsf{F})$;
                  $pc_2^C := 2$
 [] $(H_2)$ wf : $pc_2^C = 2 \rightarrow$
                  $(y_2, s) := (\mathsf{T}, 2)$;
                  $pc_2^C := 3$
 [] $(C_2^2)$   $pc_2^C = 3 \wedge (\neg y_1 \vee s = 1)$
                  $\rightarrow CS_2 := 1; pc_2^C := 1$
od : $CS_1, CS_2$
```

For easy reference, let us denote the higher level processes by $\mathscr{A}_i$ and the lower level processes by $\mathscr{C}_i$. The actions in $\mathscr{A}_i$ are labelled by A_i^1 and A_i^2; the actions in $\mathscr{C}_i$ are labelled by C_i^1, H_i and C_i^2. Apart from the new program counters, the lower level system has three additional variables y_1, y_2 and s with which the access of the critical sections are controlled. Process $\mathscr{C}_i$ indicates the wish of entering its critical section by setting y_i to value T. Variable s is used to record which process is the later one to request access; when both processes are contesting access, the one requested earlier gets the right. Therefore, the waiting time for each process is bounded, because its first two actions only depend on the program counter for enableness and after the second action which sets variables y_i and s, the other process can at most access the critical section once.

Note that now the first two actions in $\mathscr{C}_i$ are associated with weak fairness and the third action has no fairness requirement at all; this is because some fairness requirements have been achieved through explicit programming.

Let

$$\begin{aligned}
&a \overset{\text{def}}{=} pc_1^A, pc_2^A\\
&c \overset{\text{def}}{=} pc_1^C, pc_2^C, y_1, y_2, s\\
&z \overset{\text{def}}{=} CS_1, CS_2
\end{aligned}$$

The refinement relation basically relates the program counters under an invariant

$$R(a,c,z) \overset{\text{def}}{=} R_1(a,c,z) \wedge R_2(a,c,z) \wedge inv(c,z)$$

where

$$\begin{aligned}
R_i(a,c,z) \overset{\text{def}}{=} &(pc_i^A = 1 \wedge pc_i^C = 1) \vee (pc_i^A = 2 \wedge pc_i^C = 2)\\
&\vee(pc_i^A = 2 \wedge pc_i^C = 3 \wedge y_i)
\end{aligned}$$

and $inv(c,z)$ is

$$\begin{aligned}
&(CS_1 = 1 \Rightarrow y_1 \wedge (\neg y_2 \vee s = 2)) \wedge (pc_1^C = 2 \vee pc_1^C = 3 \Rightarrow CS_1 = 0)\\
&\wedge(CS_2 = 1 \Rightarrow y_2 \wedge (\neg y_1 \vee s = 1)) \wedge (pc_2^C = 2 \vee pc_2^C = 3 \Rightarrow CS_2 = 0)\\
&\wedge(pc_1^C = 1 \vee pc_1^C = 2 \vee pc_1^C = 3) \wedge (pc_2^C = 1 \vee pc_2^C = 2 \vee pc_2^C = 3)\\
&\wedge(s = 1 \vee s = 2)
\end{aligned}$$

Checking simulation of actions is straightforward. The less trivial part of it is to show

$$A_1^2 \preceq_R^f C_1^2 \qquad \text{(and the symmetrical case } A_2^2 \preceq_R^f C_2^2\text{)}$$

which as we indicated earlier, is divided into

$$pc_1^C = 3 \wedge (\neg y_2 \vee s = 2) \wedge R(a,c,z) \leq pc_1^A = 2 \wedge CS_2 \neq 1 \qquad (*)$$

and

$$\begin{aligned}
&pc_1^C = 3 \wedge (\neg y_2 \vee s = 2) \wedge R(a,c,z) \wedge (CS_1 := 1; pc_1^A := 1)q(a,z) \qquad (**)\\
&\leq (CS_1 := 1; pc_1^C := 1)(\lambda(c,z).(\exists a.R(a,c,z) \wedge q(a,z)))(c,z)
\end{aligned}$$

Lemma 1.

$$\begin{aligned}
&(pc_1^C = 3 \wedge (\neg y_2 \vee s = 2) \wedge R(a,c,z))\\
&= (pc_1^A = 2 \wedge pc_1^C = 3 \wedge y_1 \wedge (\neg y_2 \vee s = 2) \wedge R_2(a,c,z) \wedge inv(c,z))
\end{aligned}$$

Proof. Direct from the definition of R. □

Therefore, (*) holds because

$$\begin{aligned}
&\quad pc_1^C = 3 \wedge (\neg y_2 \vee s = 2) \wedge R(a,c,z)\\
&= \{\text{Lemma 1}\}\\
&\quad pc_1^A = 2 \wedge pc_1^C = 3 \wedge y_1 \wedge (\neg y_2 \vee s = 2) \wedge R_2(a,c,z) \wedge inv(c,z)\\
&\leq \{y_1 \wedge (\neg y_2 \vee s = 2) \leq \neg(y_2 \wedge (\neg y_1 \vee s = 1)),\\
&\qquad inv(c,z) \leq (CS_2 = 1 \Rightarrow y_2 \wedge (\neg y_1 \vee s = 1))\}\\
&\quad pc_1^A = 2 \wedge \neg(y_2 \wedge (\neg y_1 \vee s = 1)) \wedge (CS_2 = 1 \Rightarrow y_2 \wedge (\neg y_1 \vee s = 1))\\
&\leq pc_1^A = 2 \wedge CS_2 \neq 1
\end{aligned}$$

Lemma 2.

$$(CS_1 := 1; pc_1^A := 1)q(a,z) = q(1, pc_2^A, 1, CS_2)$$

Proof. Direct from the predicate transformer for assignments. □

Lemma 3.

$$\begin{aligned}&(CS_1 := 1; pc_1^C := 1)(\lambda(c,z).(\exists a.R(a,c,z) \wedge q(c,z)))\\ \geq&(\exists a.y_1 \wedge (\neg y_2 \vee s = 2) \wedge R_2(a,c,z) \wedge inv(c,z) \wedge q(1, pc_2^A, 1, CS_2))\end{aligned}$$

Proof. : Direct from the predicate transformer for assignments. □

Thus, (**) follows from Lemmas 1, 2 and 3.

As for fairness conditions, the ones concerning A_1^1 and A_2^1 are easy, since when they are enabled the corresponding lower level actions are also enabled and they are associated with the same kind of fairness notion.

Let $\mathscr{C}_1^2$ be the following loop

```
do
  $(C_1^1)$ wf : $pc_1^C = 1 \rightarrow (CS_1, y_1) := (0, \mathsf{F}); pc_1^C := 2$
 [] $(H_1)$ wf : $pc_1^C = 2 \rightarrow (y_1, s) := (\mathsf{T}, 1); pc_1^C := 3$
 [] $(C_1^{2'})$      $pc_1^C = 3 \wedge (\neg y_2 \vee s = 2) \rightarrow CS_1 := 1; pc_1^C := 1;$ exit
od
```

The fairness condition for A_1^2 is implied by

$$\{(pc_1^C = 2 \vee (pc_1^C = 3 \wedge y_1)) \wedge inv(c,z)\}\ \mathscr{C}_1^2 \parallel \mathscr{C}_2$$

Informally this is easy to understand. Because of the fairness assumption of H_1, the control will eventually reach $C_1^{2'}$ with y_1 being set to T. Then process $\mathscr{C}_2$ can at most execute each of its actions once before reaching its third action after setting s to 2, and gets blocked afterwards. Then $C_1^{2'}$ becomes the only enabled action, and will therefore be executed, causing the loop to terminate.

A formal proof is quite lengthy. Since only weak fairness is present in the iteration statement, we use the proof rule in Theorem 1. Choose the well-founded domain as the lexicographic product over natural numbers, and define

$$\begin{aligned}
I_w &\overset{\text{def}}{=} (I_w^1 \vee I_w^2) \wedge ipc\\
I_w^1 &\overset{\text{def}}{=} pc_1^C = 2 \wedge w = (2,0)\\
I_w^2 &\overset{\text{def}}{=} pc_1^C = 3 \wedge y_1\\
&\qquad \wedge((pc_2^C = 3 \wedge s = 1 \wedge w = (1,3))\\
&\qquad\quad \vee(pc_2^C = 1 \wedge w = (1,2))\\
&\qquad\quad \vee(pc_2^C = 2 \wedge w = (1,1))\\
&\qquad\quad \vee(pc_2^C = 3 \wedge s = 2 \wedge w = (1,0))\\
&\qquad)\\
ipc &\overset{\text{def}}{=} ipc_1 \wedge ipc_2\\
ipc_i &\overset{\text{def}}{=} pc_i^C = 1 \vee pc_i^C = 2 \vee pc_i^C = 3
\end{aligned}$$

The proof obligations in the termination rule are established as follows.

(i) $(\exists w.I_w)$
$=$ $\{$definition of I_w and predicate calculus$\}$
$(pc_1^C = 2 \vee (pc_1^C = 3 \wedge y_1$
$\wedge((pc_2^C = 3 \wedge (s = 1 \vee s = 2)) \vee pc_2^C = 1 \vee pc_2^C = 2)) \wedge ipc$
$\geq$ $\{$predicate calculus$\}$
$(pc_1^C = 2 \vee (pc_1^C = 3 \wedge y_1$
$\wedge(pc_2^C = 3 \vee pc_2^C = 1 \vee pc_2^C = 2) \wedge (s = 1 \vee s = 2))) \wedge ipc$
$\geq$ $\{inv \leq (pc_2^C = 3 \vee pc_2^C = 1 \vee pc_2^C = 2) \wedge (s = 1 \vee s = 2)\ \}$
$(pc_1^C = 2 \vee (pc_1^C = 3 \wedge y_1)) \wedge inv$

(ii) Let $r_1 \overset{\text{def}}{=} r_3 \overset{\text{def}}{=} r_4 \overset{\text{def}}{=} \mathsf{false}, r_2 \overset{\text{def}}{=} pc_1^C = 2$, corresponding respectively to actions C_1^1 , C_2^1, H_2 and H_1 associated with weak fairness.

(1) $\mathsf{helpful\_or\_stay}(I_w, r_i, A_j)$ holds trivially when $r_i = \mathsf{false}$, and the case for r_2 follows from the following facts

(a) $\mathsf{helpful}(I_w, \mathsf{true}, C_1^1)$
$\geq$ $\{$definition of $\mathsf{helpful}\}$
$\{I_w\}\ C_1^1\ \{\exists v | v < w.I_v\}$
$=$ $\{$definition of $C_1^1\}$
$\{I_w \wedge pc_1^C = 1\}\ (CS_1, y_1) := (0, \mathsf{F}); pc_1^C := 2\ \{\exists v | v < w.I_v\}$
$=$ $\{$definition of $I_w\}$
$\{\mathsf{false}\}\ (CS_1, y_1) := (0, \mathsf{F}); pc_1^C := 2\ \{\exists v | v < w.I_v\}$
$=$ T

(b) $\mathsf{helpful}(I_w, \mathsf{true}, C_1^{2'})$
$=$ $\{$definitions of $\mathsf{helpful\_or\_stay}$ and $C_1^{2'}\}$
T

(c) $\{I_w \wedge r_2\}\ C_2^1\ \{I_w \wedge r_2\}$
$\geq$ $\{$definitions of I_w, r_2, C_2^1 and correctness formula$\}$
$(pc_1^C = 2 \wedge w = (2,0) \wedge ipc)$
$\leq ((CS_2, y_2) := (0, \mathsf{F}); pc_2^C := 2)(pc_1^C = 2 \wedge w = (2,0) \wedge ipc)$
$\geq$ $\{$predicate transformer for assignments$\}$
$pc_1^C = 2 \wedge w = (2,0) \wedge ipc \leq (pc_1^C = 2 \wedge w = (2,0) \wedge ipc_1)$
$\geq$ $\{$predicate calculus$\}$
T

(d) $\{I_w \wedge r_2\}\ H_2\ \{I_w \wedge r_2\}$
$\geq$ $\{$definitions of I_w, r_2, H_2 and correctness formula$\}$
$(pc_1^C = 2 \wedge w = (2,0) \wedge ipc)$
$\leq ((y_2, s) := (\mathsf{T}, 2); pc_2^C := 3)(pc_1^C = 2 \wedge w = (2,0) \wedge ipc)$
$\geq$ $\{$predicate transformer for assignments$\}$
$pc_1^C = 2 \wedge w = (2,0) \wedge ipc \leq (pc_1^C = 2 \wedge w = (2,0) \wedge ipc_1)$
$\geq$ $\{$predicate calculus$\}$
T

(e) $\{I_w \wedge r_2\}\ C_2^2\ \{I_w \wedge r_2\}$
$\geq$ {definitions of I_w, r_2, C_2^2 and correctness formula}
$(pc_1^C = 2 \wedge w = (2,0) \wedge ipc)$
$\leq (CS_2 := 1; pc_2^C := 1)(pc_1^C = 2 \wedge w = (2,0) \wedge ipc)$
$\geq$ {predicate transformer for assignments}
$pc_1^C = 2 \wedge w = (2,0) \wedge ipc \leq (pc_1^C = 2 \wedge w = (2,0) \wedge ipc_1)$
$\geq$ {predicate calculus}
T

(*2*) This follows from

$\mathsf{helpful}(I_w, \mathsf{true}, H_1)$
$\geq$ {definition of $\mathsf{helpful}$}
$\{I_w\}\ H_1\ \{\exists v | v < w.I_v\}$
$=$ {definition of H_1}
$\{I_w \wedge pc_1^C = 2\}\ (y_1, s) := (\mathsf{T}, 1); pc_1^C := 3\ \{\exists v | v < w.I_v\}$
$=$ {definitions of I_w and correctness formula}
$pc_1^C = 2 \wedge w = (2,0) \wedge ipc \leq ((y_1, s) := (\mathsf{T}, 1); pc_1^C := 3)(\exists v | v < w.I_v)$
$\geq$ {definition of I_v and predicate transformer for assignments}
$pc_1^C = 2 \wedge w = (2,0) \wedge ipc$
$\leq (\exists v | v < w.((pc_2^C = 3 \wedge v = (1,3)) \vee (pc_2^C = 1 \wedge v = (1,2))$
$\vee (pc_2^C = 2 \wedge v = (1,1))) \wedge ipc_1)$
$\geq$ {$ipc \leq pc_2^C = 1 \vee pc_2^C = 2 \vee pc_2^C = 3$}
T

$I_w \wedge r_2$
$=$ {definition of r_2}
$I_w \wedge pc_1^C = 2$
$\leq$ {definition of H_1}
gH_1

(*3*) This follows from (a), (b), (2) and

(f) $\{I_w \wedge \neg(r_1 \vee r_2 \vee r_3 \vee r_4)\}\ C_2^1\ \{\exists v | v < w.I_v\}$
$\geq$ {definitions of I_w, r_i, C_2^1 and the correctness formula}
$(pc_1^C = 3 \wedge y_1 \wedge pc_2^C = 1 \wedge w = (1,2) \wedge ipc)$
$\leq ((CS_2, y_2) := (0, \mathsf{F}); pc_2^C := 2)(\exists v | v < w.I_v)$
$=$ {definition of I_v and predicate transformer for assignments}
$(pc_1^C = 3 \wedge y_1 \wedge pc_2^C = 1 \wedge w = (1,2) \wedge ipc)$
$\leq (\exists v | v < w.((pc_1^C = 2 \wedge w = (2,0))$
$\vee (pc_1^C = 3 \wedge y_1 \wedge v = (1,1))) \wedge ipc_1)$
$\geq$ {predicate calculus}
T

(g) $\{I_w \wedge \neg(r_1 \vee r_2 \vee r_3 \vee r_4)\}\ H_2\ \{\exists v | v < w.I_v\}$

$\geq$ {definitions of I_w, r_i, H_2 and correctness formula}

$pc_1^C = 3 \wedge y_1 \wedge pc_2^C = 2 \wedge w = (1,1) \wedge ipc$

$\leq ((y_2, s) := (\mathsf{T}, 2); pc_2^C := 3)(\exists v | v < w.I_v)$

$=$ {definition of I_v and predicate transformer for assignments}

$pc_1^C = 3 \wedge y_1 \wedge pc_2^C = 2 \wedge w = (1,1) \wedge ipc$

$\leq (\exists v | v < w.((pc_1^C = 2 \wedge w = (2,0))$

$\quad \vee (pc_1^C = 3 \wedge y_1 \wedge v = (1,0))) \wedge ipc_1)$

$\geq$ {predicate calculus}

T

(h) $\{I_w \wedge \neg(r_1 \vee r_2 \vee r_3 \vee r_4)\}\ C_2^2\ \{\exists v | v < w.I_v\}$

$\geq$ {definitions of I_w, r_i, C_2^2 and correctness formula}

$pc_1^C = 3 \wedge y_1 \wedge pc_2^C = 3 \wedge s = 1 \wedge w = (1,3) \wedge ipc$

$\leq (CS_2 := 1; pc_2^C := 1)(\exists v | v < w.I_v)$

$=$ {definition of I_v and predicate transformer for assignments}

$pc_1^C = 3 \wedge y_1 \wedge pc_2^C = 3 \wedge s = 1 \wedge w = (1,3) \wedge ipc$

$\leq (\exists v | v < w.((pc_1^C = 2 \wedge w = (2,0))$

$\quad \vee (pc_1^C = 3 \wedge y_1 \wedge v = (1,2))) \wedge ipc_1)$

$\geq$ {predicate calculus}

T

7. Backward simulation

It is known that forward simulation alone is not complete, i.e., there exist cases that semantically refinement holds between two systems but the fact cannot be proved by constructing a forward simulation. This happens when a nondeterministic choice is made earlier in the higher level system than in the lower level system. One simple example is as follows:

Example 11 Consider the refinement of the following two unfair systems:

```
𝒜 ≝ [[ var $a$ • $a = 0$;
      do
        $a = 0 \rightarrow (z := z + 1; a := 1$ [] $z := z + 1; a = 2)$
       [] $a = 1 \rightarrow z := z + 1$ [] $a = 2 \rightarrow z := z + 2$
      od ]] : $z$
𝒞 ≝ [[ var $c$ • $c = 0$;
      do
        $c = 0 \rightarrow z := z + 1; c := -1$
       [] $c = -1 \rightarrow (c := 1$ [] $c := 2)$
       [] $c = 1 \rightarrow z := z + 1$ [] $c = 2 \rightarrow z := z + 2$
      od ]] : $z$
```

Action system $\mathscr{A}$ immediately makes a choice whether to increase z by 1 or 2 afterwards, whereas action system $\mathscr{C}$ postpones that until the next step. To deal with situations like this, backward simulation has been proposed. A related method involves introducing the so called prophecy variables [1]. Backward simulation of action systems is based on backward simulation of actions. In refinement calculus, action A is *backward simulated* by action C under R, is defined as follows:

$$A \leq_R^b C \overset{\text{def}}{=} (\forall a . R \Rightarrow Aq) \leq C(\forall a . R \Rightarrow q)$$

This definition is difficult to verify directly (unlike the definition for forward simulation), and consequently, is not suitable as the proof obligation. In practice, we can use a condition based on the following property. Suppose the next-state relations of actions A and C are denoted respectively by $n\,A$ and $n\,C$, then $A \leq_R^b C$ if the following two conditions hold

(i) $R(a', c', z') \wedge nC(c, z)(c', z') \leq (\exists a . R(a, c, z) \wedge (\neg tA(a, z) \vee nA(a, z)(a', z')))$

(ii) $\neg tC(c, z) \leq (\exists a . R(a, c, z) \wedge \neg tA(a, z))$

Again, we study refinement of action systems of the following forms:

$$\mathscr{A} \overset{\text{def}}{=} [\![\ \mathsf{var}\ a \bullet p; \mathsf{do}\ A\ \mathsf{od}\ ]\!] : z$$
$$\mathscr{C} \overset{\text{def}}{=} [\![\ \mathsf{var}\ c \bullet q; \mathsf{do}\ C\ [\,]\ H\ \mathsf{od}\ ]\!] : z$$

Backward simulation of (unfair) action systems. Data refinement relation $R(a, c, z)$ is said to be a backward simulation between $\mathscr{A}$ and $\mathscr{C}$, denoted by $\mathscr{A} \leq_R^b \mathscr{C}$, if R is a total relation and the following holds

(i) Initialisation: $R \wedge q \leq p$
(ii) Main actions: $A \leq_R^b C$
(iii) Stuttering actions: $\mathsf{skip} \ \leq_R^b H$
(iv) Exit condition: $\neg(gC \vee gH)(c, z) \leq (\exists a . R(a, c, z) \wedge (\neg tA \vee \neg gA)(a, z))$
(v) Internal convergence: $\neg t\,(\mathsf{do}\ H\ \mathsf{od})(c, z) \leq (\exists a . R(a, c, z) \wedge \neg tA(a, z))$

Backward simulation gets the name from the way that the corresponding higher level computations are constructed. For any computation γ of $\mathscr{C}$, a computation α of $\mathscr{A}$ is built backwards. If γ is finite, this construction is straightforward. If γ is infinite, one resorts to a continuity argument. For this to work, the action systems must satisfy some conditions. One sufficient condition is to require that the higher level action system $\mathscr{A}$ to be *internally continuous* (or in Abadi & Lamport's terminology, it has no *infinite invisible nondeterminism*), i.e., both the set $\{a | p(a, z)\}$ and the set $\{a' | nA(a, z)(a', z')\}$ are finite for all z and a, z, z' respectively.

For the above example, we can choose data refinement $R(a, c, z)$ as

$$(a = 0 \wedge c = 0) \vee (a = 1 \wedge (c = -1 \vee c = 1)) \vee (a = 2 \wedge (c = -1 \vee c = 2))$$

Checking the conditions of backward simulation amounts to routine calculation and is hence omitted.

We now extend backward simulation to verify fair action systems. Consider the higher and lower level action systems respectively of the form:

$$\mathscr{A} \overset{\text{def}}{=} [\![\text{ var } a \bullet p; \text{do } A \text{ od }]\!] : z$$
$$\mathscr{C} \overset{\text{def}}{=} [\![\text{ var } c \bullet q; \text{do } C \; [] \; H \text{ od }]\!] : z$$

where

$$A \overset{\text{def}}{=} A_1 \; [] \; A_2 \; [] \; \dots \; [] \; A_n$$
$$C \overset{\text{def}}{=} C_1 \; [] \; C_2 \; [] \; \dots \; [] \; C_m$$
$$H \overset{\text{def}}{=} H_1 \; [] \; H_2 \; [] \; \dots \; [] \; H_k$$

Decomposition of lower level actions is now with respect to backward simulation: as indicated by the matrix,

$$\begin{array}{c|cccc} & A_1 & A_2 & \dots & A_n \\ \hline C_1 & C_{1,1} & C_{1,2} & \dots & C_{1,n} \\ C_2 & C_{2,1} & C_{2,2} & \dots & C_{2,n} \\ \dots & & & & \\ C_m & C_{m,1} & C_{m,2} & \dots & C_{m,n} \end{array}$$

C_i is decomposed into $C_{i,1} \; [] \; C_{i,2} \; [] \; \dots \; [] \; C_{i,n}$ such that $A_j \leq^b_R C_{i,j}$.

Backward simulation of fair action systems. Data refinement relation $R(a, c, z)$ is said to be a backward simulation between $\mathscr{A}$ and $\mathscr{C}$, denoted by $\mathscr{A} \leq^b_R \mathscr{C}$, if R is a total relation and the following holds

(i) Initialisation: $R \wedge Q \leq P$
(ii) Main actions: $A_i \leq^b_R C_{j,i}$, for any $i = 1, \dots, n$ and any $j = 1, \dots, m$
(iii) Stuttering actions $\text{skip} \leq^b_R H_i$, for each $i = 1, \dots, k$
(iv) Exit condition: $\neg(gC \vee gH)(c, z) \leq (\exists a . R(a, c, z) \wedge (\neg tA \vee \neg gA)(a, z))$
(v) Internal convergence: $\neg tD(c, z) \leq (\exists a . R(a, c, z) \wedge \neg tA(a, z))$, where

$$\begin{aligned} D = \text{do } & l_{C_1} : C_1;\text{exit} \; [] \; l_{C_2} : C_2;\text{exit} \; [] \; \dots \; [] \; l_{C_n} : C_n;\text{exit} \\ & [] \; H_1 \; [] \; \dots \; [] \; H_k \\ \text{od} & \end{aligned}$$

(vi) Fairness condition: $[\exists a . R(a, c, z) \wedge gA_i(a, z)] \; \mathscr{C}^i$ holds if $A_i \in WF(\mathscr{A})$, and $\langle \exists a . R(a, c, z) \wedge gA_i(a, z) \rangle \; \mathscr{C}^i$ holds if $A_i \in SF(\mathscr{A})$, for $i = 1, \dots, n$, where

$$\mathscr{C}^i = \text{do } C^i_1 \; [] \; C^i_2 \; [] \; \dots \; [] \; C^i_n \; [] \; H_1 \dots \; [] \; H_k \text{ od}, \text{ in which}$$
$$C^i_j = l_{C_j} : (C_{j,1} \; [] \; \dots \; [] \; C_{j,i-1} \; [] \; C_{j,i};\text{exit} \; [] \; C_{j,i+1} \; [] \; \dots \; [] \; C_{j,n})$$

The soundness of the backward simulation can be argued as follows. For any fair computation γ of $\mathscr{C}$, it is known from the results about backward simulation of unfair systems that one can construct a computation α of $\mathscr{A}$ such that it approximates γ with A_i transitions matching $C_{\_,i}$ transitions. When α is finite, the construction is finite, and α is fair (all finite computations are fair). We next consider the case that α is infinite.

When α is infinite, there is no way to build it directly. Instead, for each prefix of γ, one constructs a computation prefix of $\mathscr{A}$ which approximates the former. All the resulting higher level computation prefixes form a tree whose nodes are of the form (s, L) where s is a higher level state and L is the set of indexes of those lower level states that are coupled with s by the data refinement relation, and two nodes are connected if there is a transition between them. Since the construction is infinite, the tree is also infinite. Because of the internal continuity assumption, the computation tree is finitely branching, and it follows from the well known König lemma that there exists an infinite computation, which we take as α. It has the following property: any prefix of α must also be a computation prefix constructed corresponding to a prefix of γ. Similar argument as in the case of forward simulation shows that α is a fair computation of $\mathscr{A}$ and it is easy to see that α approximates γ.

Example 12 Consider refinement between the following two systems:

```
$\mathscr{A} \overset{\text{def}}{=} [\![$ var $a \bullet a = 0;$
       do
         wf : $a = 0 \to (z := z + 1; a := 1 \; [] \; z := z + 1; a = 2)$    %action $A_1$
        [] $a = 0 \to z := z - 1 \; [] \; a = 1 \to z := z + 1 \; [] \; a = 2 \to z := z + 2$
                                                                    % action $A_2$
       od $]\!]$ : $z$
```

```
$\mathscr{C} \overset{\text{def}}{=} [\![$ var $c \bullet c = 0;$
       do
         wf : $c = 0 \to z := z + 1; c := -1$                              % action $C_{1,1}$
        [] $c = -1 \to (c := 1 \; [] \; c := 2)$                             % action $H$
        [] $c = 0 \to z := z - 1 \; [] \; c = 1 \to z := z + 1 \; [] \; c = 2 \to z := z + 2$
                                                                    % action $C_{2,2}$
       od $]\!]$ : $z$
```

These action systems are the same as in the last example, except we have introduced fairness to one action and added a new unfair action in each system (so that the fairness assumption indeed makes a difference). To prove backward simulation, we can still use the same data refinement relation

$$(a = 0 \wedge c = 0) \vee (a = 1 \wedge (c = -1 \vee c = 1)) \vee (a = 2 \wedge (c = -1 \vee c = 2))$$

The fairness condition is implied by the following termination formula

```
$\{c = 0\}$
do
  wf : $c = 0 \to z := z + 1; c := -1;$ skip                        % action $C_{1,1}$
 [] $c = -1 \to (c := 1 \; [] \; c := 2)$                            % action $H$
 [] $c = 0 \to z := z - 1 \; [] \; c = 1 \to z := z + 1 \; [] \; c = 2 \to z := z+2$
                                                                  %action $C_{2,2}$
od
```

which is easy to prove.

8. Discussion

According to our knowledge, Abadi and Lamport [1] were the first to use simulations, called refinement mappings by them, to verify refinement of fair systems. Refinement of similar systems have been studied using temporal logics in, e.g., [17] where the systems are expressed using fair transition systems, and [16] where systems are based on joint actions. Our work differs from these in that our formalism stays entirely within the tradition of the refinement calculus. In particular, the proof condition concerning fairness is expressed by termination of derived iteration statements instead of temporal logic formulas. Other related work includes [23], where Singh studied refinement of UNITY programs, which have a more restricted fairness notion.

However, our method as well as all the other ones that we are aware of for fair systems suffers the shortcoming that the number of involved proof obligations is huge for even relatively small size applications. Therefore, machine support is essential. Mechanical tools for refinement calculus have been investigated; e.g., the Refinement Calculator system [9] has been developed on the HOL theorem prover [14]. Up till now, most of such tools still lack the automation that is desired in practical software verification. However, recent advance in theorem proving techniques, such as those implemented in PVS [11], has indicated it is possible to build tools with a substantial degree of automation.

We have not considered completeness of the method in depth. There are two issues involved: first, completeness of termination rules, and second, completeness of the simulation rules. The earlier version [8] of this paper had given a set of slightly simpler termination rules, but it turned out that they were incomplete. The current rules follow closely those of Manna and Pnueli [20], and in view of their completeness results, we expect that ours are also complete. The rules in [8] are sound and easier to use when applicable, but due to the limit of space, we do not include them here. Forward simulation and backward simulation were shown to be jointly complete for a large class of systems, see e.g., [15]; on the other hand, simulation techniques are known to be incomplete in a number of situations when liveness (of which fairness is the most useful special case) is present. One example which cannot be proved by simulation is as follows:

$$\mathscr{A} \overset{\text{def}}{=} [\![\ \mathsf{var}\ i \bullet (z = 0 \land i \geq 0); \mathsf{do}\ i > 0 \rightarrow z := z + 1; i := i - 1\ \mathsf{od}\ ]\!] : z$$
$$\mathscr{C} \overset{\text{def}}{=} [\![\ \mathsf{var}\ b \bullet (z = 0 \land b = \mathsf{T}); \mathsf{do}\ b \rightarrow z := z + 1\ [\,]\ \mathsf{wf}:\ b \rightarrow b := \mathsf{F}\ \mathsf{od}\ ]\!] : z$$

In the higher level system $\mathscr{A}$, the initial value of i can be any natural number. The traces of $\mathscr{A}$ are any finite sequences of states in which z is increased by 1 from the initial value 0. The traces of the lower level system $\mathscr{C}$ are the same set, as the fairness assumption ensures that the guard will be disabled eventually. This example is typically not possible to verify by forward simulation (since the higher level system decides the number of iterations when the action system is initialised, hence it makes the nondeterministic choice earlier than the lower level system), but we cannot apply backward simulation either, because the higher level system is not internally continuous. We are aware that Jonsson [15] proposed a weaker condition than internal continuity and the above example can indeed be verified, but his new condition, as observed in [19], lacked the key feature of simulation techniques, namely, it did not reduce global verification about infinite computations to that of individual actions.

Acknowledgement. We thank Michael Butler, Kai Engelhardt, Kaisa Sere, Yih-Kuen Tsay and Jockum von Wright for a number of discussions and comments. The work reported here was carried out within the IRENE-project supported by the Academy of Finland.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. Theoret. Comput. Sci. **82**, 253–284 (1991)
2. Back, R.: Correctness preserving program refinements: Proof theory and applications. Mathematical Center Tracts No. **131**, Mathematical Centre, Amsterdam, 1980.
3. Back, R.: A calculus of refinements for program derivations. Acta Inf. **25**, 593–624 (1988)
4. Back, R.: Refinement calculus, part II: Parallel and reactive programs. In: De Bakker, J.W., De Roever, W.-P., Rozenberg, G. (eds.) REX workshop on refinement of distributed systems. Proceedings, Nijmegen, the Netherlands 1989. (Lect. Notes Comput. Sci., vol. 430) Berlin Heidelberg New York: Springer 1990
5. Back, R., Kurki-Suonio, R.: Decentralization of process nets with centralized control. In: 2nd ACM SIGACT-SIGOPS symp. on principles of distributed computing. Proceedings. (pp. 131–142) ACM 1983
6. Back, R., Sere, K.: Stepwise refinement of parallel algorithms. Sci. of Comput. Prog. **13**, 133–180 (1990)
7. Back, R., von Wright, J.: Refinement calculus, part I: Sequential programs. In: De Bakker, J.W., De Roever, W.-P., Rozenberg, G. (eds.) REX workshop on refinement of distributed systems. Proceedings, Nijmegen, the Netherlands 1989. (Lect. Notes Comput. Sci., vol. 430) Berlin Heidelberg New York: Springer 1990
8. Back, R., Xu, Q.-W.: Fairness in action systems. Technical report No. **159**, Åbo Akademi, Finland (1995).
9. Butler, M.J., Långbacka, T.: Program derivation using the refinement calculator. In: Von Wright, J., Grundy, J., Harrison, J. (eds.) The 1996 international conference on theorem proving in higher order logics. Proceedings, Turku, Finland 1996. (Lect. Notes Comput. Sci., vol. 1125) Berlin Heidelberg New York: Springer 1996
10. Chandy, K., Misra, J.: Parallel program design: A foundation. Reading, MA: Addison–Wesley 1988
11. Owre, S., Shankar, N., Rushby, J.M.: User guide for the PVS specification and verification system. Comput. Sci. Lab., SRI International, Menlo Park, CA, USA 1993
12. Dijkstra, E.: A Discipline of programming. Englewood Cliffs, NJ: Prentice–Hall 1976
13. Francez, N.: Fairness. Berlin Heidelberg New York: Springer 1986
14. Gordon, M.J.C., Melham, T.F. (eds): Introduction to HOL: A theorem proving environment for higher order logic. New York: Cambridge University Press 1993
15. Jonsson, B.: Simulations between specifications of distributed systems. In: Baeten, J.C.M., Groote, J.F. (eds.) 2nd International Conference on Concurrency Theory (CONCUR'91). Proceedings, Amsterdam, the Netherlands, 1991. (Lect. Notes Comput. Sci., vol. 527) Berlin Heidelberg New York: Springer 1991
16. Jonsson, B.: Compositional specification and verification of distributed systems. ACM Trans. Program. Lang. Syst. **16**(2), 259–303 (1994)
17. Kesten, Y., Manna, Z., Pnueli, A.: Temporal verification of simulation and refinement. In: De Bakker, J.W., De Roever, W.-P., Rozenberg, G. (eds.) REX school a decade of concurrency: Reflections and perspectives. Proceedings, Noordwijkerhout, the Netherlands, 1993. (Lect. Notes Comput. Sci., vol. 803) Berlin Heidelberg New York: Springer 1994

18. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. **16**(3), 872–923 (1994)
19. Lynch, N., Vaandrager, F.: Forward and backward simulations for timing-based systems. In: De Bakker, J.W., De Roever, W.-P., Rozenberg, G. (eds.) REX workshop on real-time: Theory in practice. Proceedings, Mook, the Netherlands, 1991. (Lect. Notes Comput. Sci., vol. 600) Berlin Heidelberg New York: Springer 1992
20. Manna, Z., Pnueli, A.: Completing the temporal picture. Theoret. Comput. Sci. **83**(1), 97–130 (1991)
21. Morgan, C.: Programming from specifications. Englewood Cliffs, NJ: Prentice–Hall 1990
22. Morris, J.: A theoretical basis for stepwise refinement and the programming calculus. Sci. of Comput. Prog. **9**, 287–306 (1987)
23. Singh, A.: Program refinement in fair transition systems. Acta Inf. **30**, 503–535 (1993)

This article was processed by the author using the LaTeX style file *pljour1m* from Springer-Verlag.