DOI 10.1007/s00165-003-0005-6
BCS © 2003
Formal Aspects of Computing (2003) 15: 103–117

Formal Aspects of Computing

Compositional Action System Refinement

R. J. R. Back and J. von Wright
Åbo Akademi University and Turku Centre for Computer Science, Turku, Finland

Abstract. We show how a parallel composition of action systems can be refined by refining the components separately, and checking non-interference against invariants and guarantee conditions, which are abstract and stable. The guarantee condition can be thought of as a very abstract specification of how a system affects the global state, and it allows us to show that an action system refinement is valid in a given environment, even if we do not know any of the details of that environment. The paper extends the traditional notion of action systems slightly, and it makes use of a generalisation of the attribute model for program variables.

Keywords: Action systems; Refinement; Compositionality; Rely/guarantee

1. Introduction

The action system framework [BaK88, BaS91] is a popular state-based formalism for distributed systems. A system is described as a parallel composition of a number of component action systems, which repeatedly execute atomic actions on a global and a local state. Refinement of action systems reduces through a predicate transformer interpretation of the actions to data refinement and is efficiently handled using the Refinement Calculus [Bac80, BaW94].

Compositionality is important in all approaches to distributed systems: we want to be able to draw conclusions about a system as a whole by proving properties of the parts separately, with only minimal knowledge about the other parts of the system. In refinement, an important compositionality requirement is that it should be possible to refine components of a system in parallel, without knowledge about how other components are refined or implemented. Numerous methods for compositional verifications have been developed, but for state-based formalisms they can generally be seen as variations on the rely–guarantee (or assumption–commitment) style of reasoning that was originally part of the VDM method [CoJ95, Jon86, XRH97].

Although refinement methods for action systems were developed quite some time ago, the question of compositionality has not been investigated thoroughly. In this paper, we show how the rely–guarantee style of reasoning can be adapted to action systems. We reformulate action system refinement in a way that improves compositionality, by making it closer to the rely–guarantee paradigm. This is done by introducing invariants and guarantee conditions as part of the description of an action system. System descriptions in the action system approach are generally given as explicit compositions of action systems, so the intended environment is modelled in terms of action systems rather than rely conditions. Furthermore, we consistently use the predicate transformer framework of the refinement calculus, which allows us to express rules more abstractly than in a relational setting.

Correspondence and offprint requests to: J. von Wright, Åbo Akademi University, FIN-20520 Turku, Finland. Email: jwright@abo.fi

Traditionally, action systems have been written using a guarded command syntax:

$|[\,\mathsf{var}\ a \mid p_0; \mathsf{do}\ A_1 [] \ \ldots [] A_n\ \mathsf{od}]| : u$

where u are the global and a the local variables, p_0 is the initialisation and A_i are actions of the form $g_i \to S_i$ with guard g_i and body S_i. Intuitively, enabled actions are repeatedly executed (atomically) until no action is enabled and termination occurs. We simplify and generalise this slightly, by considering the actions as merged into a single action A, and by introducing a separate termination predicate p, so that an action system is a septuple (p_0, A, p, u, a, I, P) (where I is an invariant and P a guarantee relation). Intuitively, the system may choose between continuing and terminating in a situation where p holds and the action A is enabled, and it deadlocks if it reaches a point where A is disabled but the termination condition p does not hold. We also handle program variables more strictly than usual. Thus, program variables are normal variables within a higher-order logic, using an extension of the *attribute* model described in [BaW98].

2. States and Program Statements

We begin with an overiew of the refinement calculus background for this paper, and an extension of the program variable model described in [BaW98]. The program variable model makes it possible to reason rigorously about the relationships between the state spaces of different action systems.

2.1. Program Variables

We use a modified version of the *attribute model* for program variables, described in [BaW98]. The state space is assumed to be a polymorphic type Σ in higher-order logic. We do not assume anything specific about this type, but the assumptions that we make about program variables below will implicitly require that Σ is 'big enough'.

A *program variable x of type α* is a triple $x = (f, g, h)$ with types

$$f : \alpha \to \Sigma \to \Sigma \qquad g : \Sigma \to \alpha \qquad h : \Sigma \to \Sigma$$

For simplicity we write add, val and del for the three projections, so $f = \mathsf{add}.x$, $g = \mathsf{val}.x$, and $h = \mathsf{del}.x$. Intuitively, $\mathsf{add}.x.a.\sigma$ adds variable x to state σ, giving it the value a; $\mathsf{val}.x.\sigma$ reads the value of variable x in state σ; and $\mathsf{del}.x.\sigma$ deletes the variable x from the state σ.

For any program variable x, the following three conditions formalise the intended meanings of add, val and del:

(a) $\mathsf{del}.x.(\mathsf{add}.x.a.\sigma) = \sigma$
(b) $\mathsf{val}.x.(\mathsf{add}.x.a.\sigma) = a$
(c) $\mathsf{add}.x.(\mathsf{val}.x.\sigma).(\mathsf{del}.x.\sigma) = \sigma$

Note that an expression of the form $\mathsf{add}.x.a'.(\mathsf{add}.x.a.\sigma)$ is perfectly legal; intuitively the 'new version' of the variable x covers the value of the old one, until it is deleted from the state.

Furthermore, if x and y are two distinct program variables, then the following four conditions are assumed to hold:

(d) $\mathsf{del}.y.(\mathsf{add}.x.a.\sigma) = \mathsf{add}.x.a.(\mathsf{del}.y.\sigma)$
(e) $\mathsf{add}.y.b.(\mathsf{add}.x.a.\sigma) = \mathsf{add}.x.a.(\mathsf{add}.y.b.\sigma)$
(f) $\mathsf{del}.y.(\mathsf{del}.x.\sigma) = \mathsf{del}.x.(\mathsf{del}.y.\sigma)$
(g) $\mathsf{val}.y.(\mathsf{add}.x.a.\sigma) = \mathsf{val}.y.\sigma$

Conditions (d)–(g) indicate a kind of independence: operations on distinct variables do not interfere with each other.

As in [BaW98], we write $\mathsf{var}\ x_1, \ldots, x_n$ to indicate that the program variables are assumed to satisfy (pairwise) the assumptions above. There, the notion of program variables was based on pairs $(\mathsf{set}.x, \mathsf{val}.x)$, but we can define $\mathsf{set}.x$:

$$\mathsf{set}.x.a.\sigma \ \triangleq\ \mathsf{add}.x.a.(\mathsf{del}.x.\sigma) \qquad \text{(set variable)}$$

and all the properties of set postulated in [BaW98] can be derived from the properties of add and del above.

It is easily seen that the assumptions (a)–(g) above are consistent, by considering a model where each variable corresponds to an infinite stack of values, and where add pushes a new value onto the stack, val reads the top and del pops the top.

2.2. Assignments

As in [BaW98], an *expression* e of type α is a function $\Sigma \to \alpha$ written using higher-order logic syntax with the state argument left implicit and with possible occurrences of $\mathsf{val}.\, x$, for program variables x (for simplicity, we write x rather than $\mathsf{val}.\, x$ in expressions). For example, the value of expression $a + 2x$ in state σ is computed as $a + 2 \cdot \mathsf{val}.\, x.\, \sigma$ (assuming that x is a program variable and a is a constant or an ordinary variable).

An *assignment* $x := e$ is then an abbreviation for a state function, defined as follows:

$$(x := e).\, \sigma \;\hat{=}\; \mathsf{set}.\, x.\, (e.\, \sigma).\, \sigma \qquad \text{(assignment)}$$

where e is an expression of the same type as x.

A *relational assignment* has the form $(x := x' \mid t)$ and is defined as

$$(x := x' \mid t).\, \sigma.\, \sigma' \;\hat{=}\; (\exists a \bullet \sigma' = \mathsf{set}.\, x.\, a.\, \sigma \wedge t.\, a.\, \sigma)$$

Note that if x is a program variable of type α, then t has type $t : \alpha \to \Sigma \to \mathsf{Bool}$, and $x' : \alpha$ is a bound variable in the relational assignment.

In our analysis of action system refinement, we will need *state-replacing assignments* which add and delete variables in an orderly fashion:

$$(+x - y \mid t).\, \sigma.\, \sigma' \;\hat{=}\; (\exists a \bullet \sigma' = \mathsf{add}.\, x.\, a.\, (\mathsf{del}.\, y.\, \sigma) \wedge t.\, a.\, (\mathsf{val}.\, y.\, \sigma).\, (\mathsf{del}.\, y.\, \sigma))$$

Here, if x has type α and y type β, then t has type : $\alpha \to \beta \to \Sigma \to \mathsf{Bool}$. Intuitively, the expression t relates the new value of x, the old value of y, and the rest of the state.

All three kinds of assignments are easily generalised to the case when x and y stand for sets of program variables rather than single variables (*multiple assignments*). Since sets can be empty, we also get useful special cases of the state-replacing assignment that add or remove variables:

$$(+x \mid t).\, \sigma.\, \sigma' \;\equiv\; (\exists a \bullet \sigma' = \mathsf{add}.\, x.\, a.\, \sigma \wedge t.\, a.\, \sigma)$$
$$(-y \mid t).\, \sigma.\, \sigma' \;\equiv\; \sigma' = \mathsf{del}.\, y.\, \sigma \wedge t.\, (\mathsf{val}.\, y.\, \sigma).\, (\mathsf{del}.\, y.\, \sigma)$$

In these cases, the type of t is $\alpha \to \Sigma \to \mathsf{Bool}$ and $\beta \to \Sigma \to \mathsf{Bool}$, respectively.

For the inverse of relation R we write R^-:

$$R^-.\, \gamma.\, \sigma \;\hat{=}\; R.\, \sigma.\, \gamma \qquad \text{(inverse relation)}$$

In particular, we get

$$(+x - y \mid t)^- = (+y - x \mid t^-)$$

since t^- is t with the first two arguments swapped, and thus $t^-.\, y.\, x.\, \sigma = t.\, x.\, y.\, \sigma$.

We also sometimes need to coerce a predicate into a relation:

$$|p|.\, \sigma.\, \gamma \;\hat{=}\; p.\, \sigma \wedge \sigma = \gamma \qquad \text{(test relation)}$$

2.3. Program Statements

A *statement* S is a monotonic predicate transformer, i.e., a monotonic function $S : (\Sigma \to \mathsf{Bool}) \to (\Sigma \to \mathsf{Bool})$. In practice, statements are written using assignments, composed using standard constructs (sequential composition, conditionals, loops etc.). We will not go into the syntax of statements here, but we note that a state function f (e.g., an ordinary assignment) is lifted to a statement $\langle f \rangle$, while a state relation R (e.g., a relational or a state-replacing assignment) is lifted to a demonic statement $[R]$ or an angelic statement $\{R\}$ by the following definitions [BaW98]:

$$\langle f \rangle . q . \sigma \;\hat{=}\; q . (f . \sigma) \qquad \text{(functional update)}$$

$$[R] . q . \sigma \;\hat{=}\; (\forall \sigma' \bullet R . \sigma . \sigma' \Rightarrow q . \sigma') \qquad \text{(demonic update)}$$

$$\{R\} . q . \sigma \;\hat{=}\; (\exists \sigma' \bullet R . \sigma . \sigma' \wedge q . \sigma') \qquad \text{(angelic update)}$$

If $p : \Sigma \to \mathsf{Bool}$ is a state predicate, then the *guard* $[p]$ and the *assertion* $\{p\}$ are defined as follows:

$$[p] . q \;\hat{=}\; \neg p \cup q \qquad \text{(guard)}$$

$$\{p\} . q \;\hat{=}\; p \cap q \qquad \text{(assertion)}$$

where we use set notation for operations on predicates. Guards and assertions are in practice written as Boolean expressions. Finally we define two composition operators for statements

$$(S_1 \sqcap S_2) . q \;\hat{=}\; S_1 . q \cap S_2 . q \qquad \text{(demonic choice)}$$

$$(S_1 ; S_2) . q \;\hat{=}\; S_1 . (S_2 . q) \qquad \text{(sequential composition)}$$

A statement (predicate transformer) is *conjunctive* if it distributes over arbitrary non-empty meets (intersections) of predicates. Conjunctive statements model demonic non-determinism. Dually, a statement is *disjunctive* if it distributes over arbitrary non-empty joins (unions) of predicates. Disjunctive statements model angelic non-determinism [BaW98]. In this paper, non-conjunctive statements appear only as abstraction statements in data refinement.

In a syntactic framework, metalogical conditions of the form *x does not occur free in t* are important. We can describe similar *independence conditions* within the logic: a state function f is independent of program variable x if the condition $(\forall a \bullet f . \sigma = f . (\mathsf{add} . x . a . \sigma))$ holds. This gives us a notion of independence for both expressions, predicates and relations (relations are curried functions). Furthermore, a conjunctive statement S is independent of x if it can be written in the form $\{p\}; [Q]$ where both p and Q are independent of x.

2.4. Data Refinement

A relation of the form $R = (+a - a' \mid r)$ is called an *abstraction relation*. The corresponding angelic assignment $\{R\}$ is called an abstraction statement and is used in the definition of data refinement (we restrict our attention in this paper to forward data refinement). We say that *S is data refined by S′ with respect to abstraction relation R* if

$$S \sqsubseteq_R S' \;\hat{=}\; \{R\}; S \sqsubseteq S'; \{R\} \qquad \text{(data refinement)}$$

As is well known, there are alternative equivalent characterisations of forward data refinement [GaM91, BaW00]:

$$S \sqsubseteq_R S' \;\equiv\; \{R\}; S; [R^-] \sqsubseteq S' \qquad (1)$$

$$S \sqsubseteq_R S' \;\equiv\; S \sqsubseteq [R^-]; S'; \{R\} \qquad (2)$$

3. Action Systems

An *action system* is a seven-tuple $\mathcal{A} = (p_0, A, p, u, a, I, P)$, where u and a are disjoint sets of program variables (global and local); p_0, p and I are state predicates (the *initialisation*, the *exit predicate* and the *invariant*); A is a predicate transformer (the *action*); and P is a state relation over the global variables (the *guarantee relation*). We assume that p_0, A, p, I and P are independent of all program variables outside $u \cup a$.

Intuitively, execution of the action system $\mathcal{A}$ starts with the local variables being initialised so that p_0 holds (and I). After that, execution can terminate whenever p holds, and the action A can be executed whenever it is enabled, preserving I and changing the global variables in a way that is consistent with P. Formally, the semantics of an action system is a set of *traces*, as described below.

Our definition of an action system differs from the traditional one [BaS91] in two important respects. First, we include an invariant and a guarantee relation in the definition; and second, we identify three major components (initialisation, action, finalisation) and do not consider separate subactions.

3.1. Internal Consistency

The invariant and the guarantee relation of an action system can be considered as *annotations*. They give an abstract description of the system that that can be used as information when reasoning about the interaction between the system and its environment. If the invariant and the guarantee relation are well chosen, then they give all the essential information about the system, and there is no need to know the details of the initialisation and the action.

In order to be considered meaningful, an action system is assumed to satisfy the following *internal consistency* criteria:

- The action A is conjunctive (this means that we can without loss of generality assume that it is written in the form $\{q\};[Q]$ for some predicate q and some relation Q).
- p_0, I, A and p are independent of all program variables outside $u \cup a$ (this means that $\mathcal{A}$ cannot magically change the values of any other variables).
- I is in fact an invariant, i.e., the following two conditions are satisfied:

$$p_0 \subseteq I \tag{3}$$
$$I \subseteq A.I \tag{4}$$

- P is in fact a guarantee relation, which means the following. First, P must constrain only the global variables, i.e., P must be of the form

$$P = (a, u := a', u' \mid \overline{P}.u.u')$$

for some relation $\overline{P}$ over the global variables. Second, in any execution of the action A, the change in global variables must be consistent with P:

$$\{I\};[P] \sqsubseteq A \tag{5}$$

3.2. Semantics of Action Systems

We now briefly describe the formal semantics of action systems, as a generalisation of the semantics for a slightly simpler notion of action system described in [BaW94]. Execution of an action system gives rise to a sequence of states, which we call a *behaviour*. Behaviours can be infinite or finite, and finite behaviours can be aborted, (normally) terminated, or miraculous. For uniformity, we extend aborted (or terminated, or miraculous) behaviours with an infinite repetition of the special symbol $\perp$ (or $+$, or $\top$, respectively). The symbols $\perp$, $+$ and $\top$ are called *improper states*.

Assume that $\mathcal{A} = (p_0, A, p, u, a, I, P)$ is an internally consistent action system and that the action A is decomposed as $\{tA\};[nA]$. Then $\sigma = (\sigma_0, \sigma_1, \dots)$ is a possible behaviour of $\mathcal{A}$ if and only if the following conditions hold:

- $p_0.\sigma_0$

and for $i = 0, 1, \dots,$

- if σ_i is improper then $\sigma_{i+1} = \sigma_i$, and
- if σ_i is proper then
  - either $\neg tA.\sigma_i$ and $\sigma_{i+1} = \perp$ (aborting behaviour),
  - or $tA.\sigma_i \wedge p.\sigma_i$ and $\sigma_{i+1} = +$ (terminating behaviour),
  - or $tA.\sigma_i \wedge \neg p.\sigma_i \wedge nA.\sigma_i = \emptyset$ and $\sigma_{i+1} = \top$ (miraculous behaviour),
  - or $tA.\sigma_i \wedge nA.\sigma_i.\sigma_{i+1}$.

A miraculous behaviour can be understood as a deadlock; no action is enabled but neither is termination.

Because states contain hidden (local) components, behaviours as such cannot be observed. Instead, we assume that what can be observed is the *trace* of a behaviour, where the trace of a behaviour $(\sigma_0, \sigma_1, \dots)$ is the result of

- first extracting the global variables, to get $(u.\sigma_0, u.\sigma_1, \dots)$ (where we postulate $u.\gamma = \gamma$ for improper states γ),
- then removing all finite stuttering from the result,
- and finally replacing (possible) infinite stuttering with an infinite sequence of $\bot$.

A trace is supposed to be observable, and the justification of the definition is that only the global variables are externally visible, and only state changes can be observed. Note in particular that infinite stuttering (a kind of internal divergence) is equated with abortion.

The semantics of $\mathcal{A}$ is now the set of all traces of behaviours of $\mathcal{A}$:

$$[\![\mathcal{A}]\!] \triangleq \mathrm{tr}(\mathcal{A})$$

3.3. Example

We consider the following variation of a classical example. System $\mathcal{A}$ produces keys (natural numbers) in a set a and transfers them into a set x (a global variable, with maximum size n). Thus, $\mathcal{A} = (p_0, A, p, x, a, I, P)$ where

$$\begin{aligned} p_0 &= (a = \emptyset \wedge x = \emptyset) \\ A &= [a := a' \mid \exists z \notin a \cup x \cdot a' = a \cup \{z\}] \\ &\quad \sqcap [a \neq \emptyset \wedge \#x < n]; [a, x := a', x' \mid \exists z \in a \cdot a' = a - \{z\} \wedge x' = x \cup \{z\}] \\ p &= \mathsf{false} \\ I &= x \cap a = \emptyset \wedge \#x \leqslant n \\ P &= (x := x' \mid x \subseteq x') \end{aligned}$$

Here, p_0 says that initially both sets are empty. Note that the action A consists of two subactions. The first one adds a new key into a and it is always enabled (unless a contains all possible keys – in fact, the invariant could be strengthened to say that a is always finite). The second action transfers a key from a to x and it is enabled when a is non-empty and x is not full. Note that the abstract description that the guarantee condition gives is that $\mathcal{A}$ never removes elements from the global state variable x.

It is easy to check that $\mathcal{A}$ is internally consistent.

The system $\mathcal{A}$ does not make much sense on its own, but it is still possible to compute its semantics. For example, if $n = 2$ then the trace $(\emptyset, \{1\}, \{1, 2\}, \top, \top, \dots)$ is in $[\![\mathcal{A}]\!]$. It arises from many different behaviours (the set a may contain other keys that it did not move to x).

4. Composing Action Systems

4.1. Parallel Composition

The parallel composition of two action systems $\mathcal{A} = (p_0, A, p, u, a, I, P)$ and $\mathcal{B} = (q_0, B, q, u, b, J, Q)$ is defined as the new action system

$$\mathcal{A} \| \mathcal{B} \triangleq (p_0 \cap q_0, A \sqcap B, p \cap q, u, a \cup b, I \cap J, P \cup Q) \qquad \text{(parallel composition)}$$

Intuitively, $\mathcal{A} \parallel \mathcal{B}$ is executed by first initialising the local variables and then interleaving execution of actions of A and $\mathcal{B}$ (it is easily checked that $I \cap J$ is in fact an invariant). Termination is possible in a state where both $\mathcal{A}$ and $\mathcal{B}$ can terminate.

The following theorem shows when composition preserves internal consistency.

Theorem 4.1 Assume that $\mathcal{A} = (p_0, A, p, u, a, I, P)$ and $\mathcal{B} = (q_0, B, q, u, b, J, Q)$ are internally consistent. Furthermore assume that

(a) the local variables are disjoint: $a \cap b = \emptyset$, and
(b) each system respects the invariant of the other system: $I \cap J \subseteq [Q].\, I \cap [P].\, J$.

Then $\mathcal{A} \parallel \mathcal{B}$ is internally consistent.

Proof. Conjunctivity and independence of variables are easily verified. It then remains to prove the three conditions (3)–(5) for $\mathcal{A} \parallel \mathcal{B}$:

$$\begin{aligned} p_0 \cap q_0 &\subseteq I \cap J \\ I \cap J &\subseteq (A \sqcap B).(I \cap J) \\ \{I \cap J\};[P \cup Q] &\sqsubseteq A \sqcap B \end{aligned}$$

The first condition follows immediately from $p_0 \subseteq I$ and $q_0 \subseteq J$. For the second one, we have

$$\begin{aligned} & I \cap J \\ \subseteq\ & \{\text{assumption (b)}\} \\ & I \cap [P].J \cap J \cap [Q].I \\ \subseteq\ & \{I \text{ and } J \text{ are invariants of } \mathcal{A} \text{ and } \mathcal{B}, (4)\} \\ & A.I \cap I \cap [P].J \cap B.J \cap J \cap [Q].I \\ \subseteq\ & \{P \text{ and } Q \text{ are guarantee relations of } \mathcal{A} \text{ and } \mathcal{B}, (5)\} \\ & A.I \cap A.J \cap B.J \cap B.I \\ =\ & \{\text{conjunctivity and definition of choice}\} \\ & (A \sqcap B).(I \cap J) \end{aligned}$$

Finally the third condition is proved:

$$\begin{aligned} & \{I \cap J\};[P \cup Q] \\ =\ & \{\text{homomorphism}\} \\ & \{I \cap J\};([P] \sqcap [Q]) \\ \sqsubseteq\ & \{\text{distributivity, monotonicity}\} \\ & \{I\};[P] \sqcap \{J\};[Q] \\ \sqsubseteq\ & \{P \text{ and } Q \text{ are guarantee relations of } \mathcal{A} \text{ and } \mathcal{B}, (5)\} \\ & A \sqcap B \end{aligned}$$

□

From now on, we only permit parallel compositions of action systems $\mathcal{A}$ and $\mathcal{B}$ if they satisfy the *compatibility criteria* (a) and (b) of Theorem 4.1. Condition (a) can always be made to hold by renaming local variables, but (b) is a true restriction which has to be checked (unless I does not depend on the global state at all, but such invariants are probably not common). Note, however, that (b) does not refer to the main parts of the action systems, but only to the invariants and the guarantee relations. Essentially, (b) is an abstract formulation of the interference–freedom condition required for parallel programs by Owicki and Gries [OwG76].

4.2. Example

Let $\mathcal{A}$ be the action system given as an example in Section 3.3 and let $\mathcal{B}$ be a system which removes keys that are no longer in use. For simplicity, we ignore the question of how $\mathcal{B}$ makes use of the keys; B has no interesting actions and no local variables. Thus, we consider the parallel composition $\mathcal{A} \parallel \mathcal{B}$ where $\mathcal{B} = (q_0, B, q, x, \epsilon, J, Q)$ (here ϵ is an empty collection of variables) where

$$\begin{aligned} q_0 &= \mathsf{true} \\ B &= [x \neq \emptyset];[x := x' \mid \exists z \in x \bullet x' = x - \{z\}] \\ q &= \mathsf{false} \\ J &= \mathsf{true} \\ Q &= (x := x' \mid x' \subseteq x) \end{aligned}$$

Proving internal consistency of $\mathcal{B}$ is trivial, and the compatibility condition for the parallel composition can be expressed as

$$x \cap a = \emptyset \wedge \#x \leqslant n \Rightarrow (\forall x' \bullet x' \subseteq x \Rightarrow x' \cap a = \emptyset \wedge \#x' \leqslant n)$$

which is easily verified.

The invariant of $\mathcal{A}\|\mathcal{B}$ is $x \cap a = \emptyset \wedge \#x \leqslant n$ (inherited from $\mathcal{A}$) while the guarantee condition is the vacuous true (so there is no additional restriction on the ways in which mix of actions of $\mathcal{A}$ and $\mathcal{B}$ can change x).

5. Action System Refinement

Refining an action system $\mathcal{A}$ means replacing $\mathcal{A}$ by a system that cannot be distinguished from $\mathcal{A}$ by the environment. In order to make this notion formal we must first describe an approximation order on states and traces. The standard proof method for refinement (simulation via data refinement) is then described.

5.1. Approximation and Refinement

On the extended state space $\Sigma \cup \{\bot, +, \top\}$ we define the approximation ordering $\sigma \preceq \sigma'$ by

$$\sigma \preceq \sigma' \;\hat{=}\; \sigma = \bot \vee \sigma = \sigma' \vee \sigma' = \top \qquad \text{(approximation on states)}$$

This relation is then extended pointwise to traces:

$$(\sigma_0, \sigma_1, \dots) \preceq (\tau_0, \tau_1, \dots) \;\hat{=}\; (\forall i \bullet \sigma_i \preceq \tau_i) \qquad \text{(approximation on traces)}$$

Now assume that internally consistent action systems $\mathcal{A} = (p_0, A, p, u, a, I, P)$ and $\mathcal{A}' = (p_0', A', p', u, a', I', P')$ are given. The refinement relation is then defined as follows:

$$\mathcal{A} \sqsubseteq \mathcal{A}' \;\hat{=}\; (\forall \sigma' \in \mathrm{tr}(\mathcal{A}') \Rightarrow (\exists \sigma \in \mathrm{tr}(\mathcal{A}) \bullet \sigma \preceq \sigma'))$$

i.e., $\mathcal{A}'$ *is a refinement of* $\mathcal{A}$ if and only if every trace of $\mathcal{A}'$ is approximated by a trace of $\mathcal{A}$.

The refinement relation is easily shown to be a preorder (i.e., it is reflexive and transitive).

5.2. Data Refinement and Simulation

It is not practical to prove refinement between action systems using the definition. Instead, refinement is proved using the standard notion of (forward) data refinement. If $\mathcal{A} = (p_0, A, p, u, a, I, P)$ and $\mathcal{A}' = (p_0', A', p', u, a', I', P')$ are internally consistent action systems and $R = (+a - a' \mid r)$ is an abstraction relation, then we say that $\mathcal{A}$ *is data refined by* $\mathcal{A}'$ *under abstraction* R (written $\mathcal{A} \sqsubseteq_R \mathcal{A}'$) if the following conditions hold:

$$p_0' \;\subseteq\; \{R\}.\, p_0 \tag{6}$$

$$\{R\}; \{I\}; A \;\sqsubseteq\; \{I'\}; A'; \{R\} \tag{7}$$

$$\{R\}.\, (\neg p) \;\subseteq\; \neg p' \tag{8}$$

Using a standard simulation argument it can be shown that data refinement is a sufficient condition for refinement (the proof follows the lines of the soundness proof given in [BaW94]):

Theorem 5.1 If $\mathcal{A}$ and $\mathcal{A}'$ are internally consistent action systems and R is an abstraction relation with $\mathcal{A} \sqsubseteq_R \mathcal{A}'$, then $\mathcal{A} \sqsubseteq \mathcal{A}'$.

The three data refinement conditions can be interpreted as follows: for any initial state of the concrete (refining) system there must exist a corresponding initial state of the abstract (refined) system (6), for any action step of the concrete system there must exist a corresponding step of the abstract system (7), and for any state of the concrete system where termination can occur termination must be allowed in all corresponding states of the abstract system (8).

It is worth noting that conditions (6) and (8) above can also be stated on the level of refinement:

$$[+a \mid p_0] \;\sqsubseteq\; [+a' \mid p_0']; \{R\}$$

$$\{R\}; [-a \mid p] \;\sqsubseteq\; [-a' \mid p']$$

which shows the symmetry of the conditions for initialisation and finalisation.

5.3. Special Case of Refinement

The definition of refinement allows the invariant and the guarantee condition of the refining system to be fairly independent of those of the original system. We shall now see what is gained by imposing further restrictions.

Lemma 5.1 Assume that internally consistent action system $\mathcal{A} = (p_0, A, p, u, a, I, P)$ is given and that p_0', A', p' and $R = (+a - a' \mid r)$ are given, such that conditions (6)–(8) hold, and

$$I' = \{R\}.\, I \tag{9}$$

Then $\mathcal{A}' = (p_0', A', q', u, a', I', P)$ is an internally consistent action system and $\mathcal{A} \sqsubseteq_R \mathcal{A}'$.

Proof. We have

$$\begin{array}{ll} & p_0' \\ \subseteq & \{\text{refinement assumption (6)}\} \\ & \{R\}.\, p_0 \\ \subseteq & \{\text{assumption } p_0 \subseteq I \text{ (3), monotonicity}\} \\ & \{R\}.\, I \end{array}$$

and

$$\begin{array}{ll} & \{R\}.\, I \\ = & \{\text{general fact } \{p\}.\, p = p\} \\ & \{R\}.\, (\{I\}.\, I) \\ \subseteq & \{\text{assumption } I \subseteq A.\, I \text{ (4), monotonicity}\} \\ & \{R\}.\, (\{I\}.\, (A.\, I)) \\ \subseteq & \{\text{refinement assumption (7)}\} \\ & (\{I'\}; A').\, (\{R\}.\, I) \\ \subseteq & \{\text{general rule } \{p\} \sqsubseteq \mathsf{skip}\} \\ & A'.\, (\{R\}.\, I) \end{array}$$

so $\{R\}.\, I$ is in fact an invariant.

Now set $I' = \{R\}.\, I$ and we need to show that $\{I'\}; [P] \sqsubseteq A'$. We first recall [BaW98] that this refinement can be rewritten as

$$(\forall \gamma \bullet I'.\, \gamma \Rightarrow A'.\, (P.\, \gamma).\, \gamma) \tag{10}$$

We have

$$\begin{array}{ll} & A'.\, (P.\, \gamma).\, \gamma \\ \Leftarrow & \{\text{data refinement assumption (7), and (1)}\} \\ & (\{R\}; \{I\}; A; [R^-]).\, (P.\, \gamma).\, \gamma \\ \Leftarrow & \{P \text{ is guarantee relation of } \mathcal{A} \text{ (5)}\} \\ & (\{R\}; \{I\}; [P]; [R^-]).\, (P.\, \gamma).\, \gamma \\ \equiv & \{\text{definitions}\} \\ & (\exists \sigma \bullet R.\, \gamma.\, \sigma \wedge I.\, \sigma \wedge (\forall \sigma' \bullet P.\, \sigma.\, \sigma' \Rightarrow (\forall \gamma' \bullet R.\, \gamma'.\, \sigma' \Rightarrow P.\, \gamma.\, \gamma'))) \\ \equiv & \{\text{assumptions about } R \text{ and } P \text{ and program variables (see comment below)}\} \\ & (\exists \sigma \bullet R.\, \gamma.\, \sigma \wedge I.\, \sigma) \\ \equiv & \{\text{definition of angelic update}\} \\ & \{R\}.\, I.\, \gamma \\ \equiv & \{\text{refinement assumption (9)}\} \\ & I'.\, \gamma \end{array}$$

Fig. 1. Internal consistency as refinement.

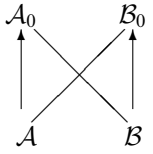

Fig. 2. Parallel composition.

Here the fourth step makes use of the independence assumptions: from $R.\gamma.\sigma$ follows $u.\gamma = u.\sigma$ and since $P = (a, u := a', u' \mid \overline{P}.u.u')$ it follows that $R.\gamma.\sigma$ and $P.\sigma.\sigma'$ and $R.\gamma'.\sigma'$ together imply $P.\gamma.\gamma'$ (the same kind of reasoning, which ultimately relies on the program variable properties postulated in Section 2.1, will be left implicit in some later proofs). □

Lemma 5.1 shows that if we accept the *calculated invariant* $\{R\}.I$ as the new invariant in a refinement, then internal consistency will be guaranteed. Furthermore, it is easily seen that for the refinement condition (7) it is in this case sufficient to prove

$$\{R\}; \{I\}; A \sqsubseteq A'; \{R\}$$

(this follows from the general rule $S; \{p\} = \{S.p\}; S; \{p\}$).

5.4. Illustrating Refinements

Note that the internal consistency conditions (3)–(5) express a refinement $\mathcal{A}_0 \sqsubseteq_{\mathsf{Id}} \mathcal{A}$ where $\mathcal{A}_0 = (I, [P], p, u, a, I, P)$ and Id is the identity relation. We illustrate this in Fig. 1, with an undecorated arrow for data refinement with respect to Id.

Similarly, we illustrate a parallel composition with action systems $\mathcal{A}$ and $\mathcal{B}$ beside each other in Fig. 2, where the lines illustrate the respect of invariants ($\mathcal{A}$ respects $\mathcal{B}_0$ and $\mathcal{B}$ respects $\mathcal{A}_0$).

6. Compositionality

The rely/guarantee approach to distributed systems generalises the idea of pre- and postconditions; a system component is specified by giving a rely relation (which describes how the component assumes that its environment behaves) and a guarantee relation (which describes how the environment can assume that the component behaves). Rely/guarantee specifications have mainly been used in the context of VDM [Jon86] and various rules have been developed for composing rely/guarantee specifications of components into specifications of systems [Sto91, XRH97].

Now consider the situation with internally consistent action systems $\mathcal{A} = (p_0, A, p, u, a, I, P)$ and $\mathcal{B} = (q_0, B, q, u, b, J, Q)$ working parallel. In general, it is not possible to conclude a refinement of the form $\mathcal{A}|\mathcal{B} \sqsubseteq \mathcal{A}'\|\mathcal{B}$ from a refinement of the form $\mathcal{A} \sqsubseteq \mathcal{A}'$. This is easily seen by considering possible traces: the traces of $\mathcal{A}\|\mathcal{B}$ need not have any relation to the traces of $\mathcal{A}$ or $\mathcal{B}$.

However, since refinement is proved using data refinement, it is sufficient if data refinement is compositional. In other words, we want to formulate conditions under which we can deduce $\mathcal{A}\|\mathcal{B} \sqsubseteq_R \mathcal{A}'\|\mathcal{B}$ from $\mathcal{A} \sqsubseteq_R \mathcal{A}'$. For this, we need a *non-interference* condition which guarantees that the relation R is maintained regardless of the interleaving of actions from $\mathcal{B}$. However, we also want this non-interference condition to be described in terms of the invariant and the guarantee condition of $\mathcal{B}$ (since these are more abstract and also more stable if $\mathcal{B}$ is refined) rather than in terms of the 'inner parts' (initialisation, action, finalisation) of $\mathcal{B}$.

6.1. Basic Compositionality Theorem

Thus, we want to verify non-interference without referring to q_0, B or q. We shall now show that this is indeed possible; it is sufficient to verify non-interference of the guarantee relation Q instead:

Theorem 6.1 Assume that $\mathcal{A}$, $\mathcal{A}'$ and $\mathcal{B}$ are internally consistent action systems, as above. Furthermore, assume that $\mathcal{A}$ and $\mathcal{B}$ respect the invariants of each other, and that the refinement $\mathcal{A} \sqsubseteq_R \mathcal{A}'$ holds. Then $\mathcal{A}'$ and $\mathcal{B}$ respect the invariants of each other and the refinement

$$\mathcal{A} \parallel \mathcal{B} \sqsubseteq_R \mathcal{A}' \parallel \mathcal{B}$$

holds, provided that

$$R; |J|; Q \subseteq R$$

Proof. First we note that the condition $R; |J|; Q \subseteq R$ can be stated as follows:

$$(\forall \gamma\ \sigma\ \sigma' \bullet R.\gamma.\sigma \wedge J.\sigma \wedge Q.\sigma.\sigma' \ \Rightarrow\ R.\gamma.\sigma')$$

and from the independence assumptions it follows that this is equivalent to

$$(\forall \gamma\ \sigma\ \sigma' \bullet R.\gamma.\sigma \wedge J.\gamma \wedge Q.\gamma.\gamma' \ \Rightarrow\ R.\gamma'.\sigma)$$

or $R^-; |J|; Q \subseteq R^-$.

Now we verify the three data refinement conditions for the parallel systems. The initialisation and finalisation conditions

$$\begin{aligned} p_0' \cap q_0 &\subseteq \{R\}.(p_0 \cap q_0) \\ p' \cap q &\subseteq \{R\}.(p \cap q) \end{aligned}$$

are straightforward, under the assumptions that q_0 and q are independent of the variables a and a'. For the action condition, if we knew $\{R\}; \{J\}; B \sqsubseteq B; \{R\}$, then we would have

$$\begin{aligned} & \{R\}; \{I \cap J\}; (A \sqcap B) \\ \sqsubseteq\ & \{\text{distributivity, monotonicity}\} \\ & \{R\}; \{I\}; A \sqcap \{R\}; \{J\}; B \\ \sqsubseteq\ & \{\text{assumptions}\} \\ & A'; \{R\} \sqcap B; \{R\} \\ =\ & \{\text{distributivity}\} \\ & (A' \sqcap B); \{R\} \end{aligned}$$

So it remains to prove that $\{R\}; \{J\}; B \sqsubseteq B; \{R\}$ in fact holds.

Since B is conjunctive, we can assume $B = \{s\}; [S]$ and we have

$$\begin{aligned} & \{R\}; \{J\}; B \sqsubseteq B; \{R\} \\ \equiv\ & \{\text{assumption, reformulate data refinement}\} \\ & \{J\}; \{s\}; [S] \sqsubseteq [R^-]; \{s\}; [S]; \{R\} \\ \equiv\ & \{\text{restate refinement}\} \\ & (\forall \sigma \bullet J.\sigma \wedge s.\sigma \ \Rightarrow\ ([R^-]; \{s\}; [S]; \{R\}).(S.\sigma).\sigma) \\ \equiv\ & \{\text{definitions, quantifier rules}\} \\ & (\forall \sigma\ \gamma \bullet J.\sigma \wedge s.\sigma \wedge R.\gamma.\sigma \\ & \Rightarrow\ s.\gamma \wedge (\forall \gamma' \bullet S.\gamma.\gamma' \Rightarrow (\exists \sigma' \bullet R.\gamma'.\sigma' \wedge S.\sigma.\sigma')) \\ \equiv\ & \{s \text{ independent of } a \text{ and } a', \text{ quantifier rules}\} \\ & (\forall \sigma\ \gamma\ \gamma' \bullet J.\sigma \wedge s.\sigma \wedge R.\gamma.\sigma \wedge S.\gamma.\gamma' \Rightarrow (\exists \sigma' \bullet R.\gamma'.\sigma' \wedge S.\sigma.\sigma')) \end{aligned}$$

Now assume $J.\sigma \wedge s.\sigma \wedge R.\gamma.\sigma \wedge S.\gamma.\gamma'$ and choose $\sigma' := \mathsf{add}.a.(\mathsf{val}.a.\sigma).(\mathsf{del}.a'.\gamma')$. Then (using the independence assumptions)

- from $J.\sigma$ and $R.\gamma.\sigma$ we get $J.\gamma$,
- because $\{J\};[Q] \sqsubseteq B$, from $J.\gamma$ and $S.\gamma.\gamma'$ we get $Q.\gamma.\gamma'$,
- because $R^-;|J|;Q \subseteq R^-$, from $R.\gamma.\sigma$ and $J.\gamma$ and $Q.\gamma.\gamma'$ we get $R.\gamma'.\sigma$,
- from $R.\gamma'.\sigma$ and the definition of σ' we get $R.\gamma'.\sigma'$, and
- from $S.\gamma.\gamma'$ and $R.\gamma.\sigma$ and $R.\gamma'.\sigma'$ we get $S.\sigma.\sigma'$.

A similar argument is then used to show that $\mathcal{A}'$ and $\mathcal{B}$ respect the invariants of each other, i.e., that $I' \cap J \subseteq [Q].\, I' \cap [P].\, I$. □

Thus, a total system refinement $\mathcal{A} \parallel \mathcal{B} \sqsubseteq_R \mathcal{A}' \parallel \mathcal{B}$ can be deduced from a refinement $\mathcal{A} \sqsubseteq_R \mathcal{A}'$ by verifying conditions that refer only to the original invariants and the guarantee conditions:

$$\frac{I \cap J \subseteq [Q].\, I \cap [P].\, J \qquad \mathcal{A} \sqsubseteq_R \mathcal{A}' \qquad R;|J|;Q \subseteq R}{\mathcal{A}\|\mathcal{B} \sqsubseteq_R \mathcal{A}'\|\mathcal{B}}$$

Note that the first and third hypotheses in this rule each states a kind of non-interference property. On the one hand, the systems are not allowed to interfere with each other (the invariants I and J), and on the other hand the system that is not refined is not allowed to interfere with the refinement (the abstraction relation R).

6.2. The General Case

Thus we have achieved a major step towards the compositionality that we wanted: we are not required to 'look inside' $\mathcal{B}$ to verify a refinement of $\mathcal{A}$ in the parallel context $\mathcal{A}\|\mathcal{B}$. Furthermore, as the following summarising theorem shows, we can permit $\mathcal{B}$ to be refined separately, independently of the refinement of $\mathcal{A}$.

Theorem 6.2 Assume that $\mathcal{A} = (p_0, A, p, u, a, I, P)$ and $\mathcal{B} = (q_0, B, q, u, b, J, Q)$ respect the invariants of each other and that $\mathcal{A}' = (p_0', A', p', u', a', I', P)$ and $\mathcal{B} = (q_0', B', q', u', b', J', Q)$ where $I' = \{R\}.\, I$ and $J' = \{S\}.\, J$. Then the following rule is valid:

$$\frac{I \cap J \subseteq [Q].\, I \cap [P].\, J \quad \mathcal{A} \sqsubseteq_R \mathcal{A}' \quad R;|J|;Q \subseteq R \quad \mathcal{B} \sqsubseteq_S \mathcal{B}' \quad S;|I|;P \subseteq S}{\mathcal{A}\|\mathcal{B} \sqsubseteq_{R;S} \mathcal{A}'\|\mathcal{B}'}$$

Proof. We first note that from $S;|I|;P \subseteq S$ and the independence assumptions we easily get $S;|I'|;P \subseteq S$, where $I' = \{R\}.\, I$. Applying Theorem 6.1 twice in succession, we then get

$$\mathcal{A}\|\mathcal{B} \sqsubseteq_R \mathcal{A}'\|\mathcal{B} \text{ and } \mathcal{A}'\|\mathcal{B} \sqsubseteq_S \mathcal{A}'\|\mathcal{B}'$$

so it is sufficient to show that $\sqsubseteq_R$ and $\sqsubseteq_S$ compose to $\sqsubseteq_{S;R}$. For this, we verify the three refinement conditions (6)–(8). First we assume $p_0' \subseteq \{R\}.\, p_0$ and $p_0'' \subseteq \{S\}.\, p_0'$ and get

$$p_0'' \subseteq \{S\}.\, p_0' \subseteq \{S\}.(\{R\}.\, p_0) = \{S;R\}.\, p_0$$

and similarly for the finalisation condition. For the body condition assume $\{R\};\{I\};A \sqsubseteq \{I'\};A';\{R\}$ and $\{S\};\{I'\};A' \sqsubseteq \{I''\};A'';\{S\}$ and get (where $I' = \{R\}.\, I$)

$$\begin{aligned}
&\{S;R\};\{I\};A \\
&= \{\text{homomorphism}\} \\
&\quad \{S\};\{R\};\{I\};A \\
&\sqsubseteq \{\text{assumption } \{R\};\{I\};A \sqsubseteq \{I'\};A';\{R\}\} \\
&\quad \{S\};\{I'\};A';\{R\} \\
&\sqsubseteq \{\text{assumption } \{S\};\{I'\};A' \sqsubseteq \{I''\};A'';\{S\}\} \\
&\quad \{I''\};A'';\{S\};\{R\} \\
&= \{\text{homomorphism}\} \\
&\quad \{I''\};A'';\{S;R\}
\end{aligned}$$

and the proof is finished. □

Figure 3 illustrates the refinement of $\mathcal{A}\|\mathcal{B}$ into $\mathcal{A}'\|\mathcal{B}'$; consistency and non-interference only have to be checked with respect to the abstract specification (invariant and guarantee relation) of the other party.

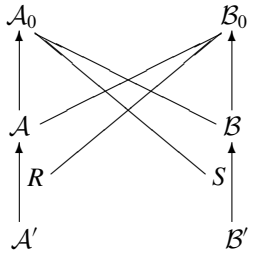

Fig. 3. Compositional refinement.

It is worth noting that the abstraction relation $S; R$ of the joint action system refinement $\mathcal{A}\|\mathcal{B} \sqsubseteq_{S;R} \mathcal{A}'\|\mathcal{B}'$ is not really a relational composition but rather a conjunction. In fact, when S and R are abstraction relations over different local variables, one can show that if $R = (+a - a' \mid r)$ and $S = (+b - b' \mid s)$ then

$$\begin{aligned}(S; R).\sigma.\gamma \;=\; & (\exists a_0\, b_0 \bullet \gamma = \mathsf{add}.(a, b).(\mathsf{del}.(a', b').\sigma) \\ & \wedge r.a_0.(a'.\sigma).(\mathsf{del}.(a', b').\sigma) \wedge s.b_0.(b'.\sigma).(\mathsf{del}.(a', b').\sigma))\end{aligned}$$

This shows that $S; R$ replaces a', b' by a, b so that r and s both hold, and furthermore that $S; R = R; S$, so the composition is symmetric. Thus, it would be reasonable to use a more symmetric notation (such as $S\|R$) for $S; R$ in this case.

The conditions $I' = \{R\}.I$ and $J' = \{S\}.J$ in Theorem 6.2 essentially say that we must use the *calculated invariant* when refining an action system; if we want to use another (e.g., stronger) invariant then we have to verify respect separately.

Note that the non-interference condition $R; |J|; Q \subseteq R$ in Theorem 6.2 holds trivially if R is local, i.e., it depends only on the local variables a and a' (and similarly for $S; |I|; P \subseteq S$).

6.3. Example

To illustrate compositionality, we return to the example from Section 4.2. Thus, $\mathcal{A} = (p_0, A, p, x, a, I, P)$ where

$$\begin{aligned}p_0 &= (a = \emptyset \wedge x = \emptyset) \\ A &= [a := a' \mid \exists z \notin a \cup x \bullet a' = a \cup \{z\}] \\ &\quad \sqcap [a \neq \emptyset \wedge \#x < n]; [a, x := a', x' \mid \exists z \in a \bullet a' = a - \{z\} \wedge x' = x \cup \{z\}] \\ p &= \mathsf{false} \\ I &= x \cap a = \emptyset \wedge \#x \leqslant n \\ P &= (x := x' \mid x \subseteq x')\end{aligned}$$

and $\mathcal{B} = (q_0, B, q, x, \epsilon, J, Q)$ where

$$\begin{aligned}q_0 &= \mathsf{true} \\ B &= [x \neq \emptyset]; [x := x' \mid \exists z \in x \bullet x' = x - \{z\}] \\ q &= \mathsf{false} \\ J &= \mathsf{true} \\ Q &= (x := x' \mid x' \subseteq x)\end{aligned}$$

Now we can refine $\mathcal{A}$ so that it always chooses, as the next key, a natural number that is larger than the largest so far. Thus $\mathcal{A}' = (p_0', A', p', u, (a, c), I', P)$ where c is always the next key to be generated:

$$\begin{aligned}p_0' &= (c = 0 \wedge a = \emptyset \wedge x = \emptyset) \\ A' &= [c, a := c', a' \mid c' > c \wedge a' = a \cup \{c\}] \\ &\quad \sqcap [a \neq \emptyset \wedge \#x < n]; [a, x := a', x' \mid \exists z \in a \bullet a' = a - \{z\} \wedge x' = x \cup \{z\}] \\ p' &= \mathsf{false}\end{aligned}$$

The guarantee relation P is the same as for $\mathcal{A}$ while the invariant is

$$I' = \{R\}.\, I \;=\; (x \cap a = \emptyset \wedge \#x \leqslant n \wedge (\forall z \in a \cap x \bullet z < c))$$

where R is the abstraction relation

$$R = (-c \mid \forall z \in a \cup x \bullet z < c)$$

The refinement $\mathcal{A} \sqsubseteq_R \mathcal{A}'$ is easily shown to hold and the non-interference condition $R; |J|; Q \subseteq R$ becomes

$$(\forall z \in a \cup x \bullet z < c) \wedge x' \subseteq x \Rightarrow (\forall z \in a \cup x' \bullet z < c)$$

which is easily proved. Note, however, that if $\mathcal{B}$ were allowed to add elements to x, then non-interference would not hold.

This is really an example of a *superposition refinement*; we added a new local variable without removing old variables. The argument for such a refinement can be efficiency – with the added variable c, new keys do not have to be checked against all old keys for uniqueness. Superposition refinement does not require any separate theory, since we can think of a as being replaced with two variables: c and a new copy of a.

7. Conclusion

The main technical contribution of this paper is to show how a parallel composition of action systems can be refined by refining the components separately, and checking non-interference against invariants and guarantee conditions, which are abstract and stable. The guarantee condition can be thought of as a very abstract specification of how a system affects the global state, and it allows us to show that an action system refinement is valid in a given environment, even if we do not know any of the details of that environment.

The paper also includes a generalisation of the attribute model for program variables to include adding and deleting program variables. This extension lets us reason about data refinement without treating program variables as a separate syntactic class outside the underlying logic. This is a definite improvement to the tuple model used before [BaW94] where 'plumbing' of tuples was needed to combine the state spaces (this was particularly visible when mechanising action system theory in the HOL theorem prover [LaW97]).

Comparing our approach with the traditional rely/guarantee reasoning of VDM, the main difference is that we assume an explicit environment with its guarantee condition given, rather than having a rely condition for the system that is refined. Also, VDM advocates a two-stage refinement methodology, where (data) refinement steps are first carried out on rely/guarantee specifications, and then refinement to code, while the action system approach allows interleaving of data refinement and algorithmic refinement.

It should be noted that Theorem 6.1 can be reinterpreted in a rely-guarantee framework by taking Q to be a rely relation for $\mathcal{A}$ and omitting the invariant of $\mathcal{B}$ (i.e., setting J to true), and requiring that the rely condition respects the invariant: $I \subseteq [Q].\, I$. We then get the following rule:

$$\frac{I \subseteq [Q].\, I \qquad \mathcal{A} \sqsubseteq_R \mathcal{A}' \qquad R; Q \subseteq R}{\mathcal{A} \| \mathcal{B} \sqsubseteq_R \mathcal{A}' \| \mathcal{B}}$$

i.e., action system refinement is compositional for any environment $\mathcal{B}$ that guarantees the rely relation of $\mathcal{A}$. The condition $R; Q \subseteq R$ is then a consistency check that the data refinement does not violate the rely relation of $\mathcal{A}$.

We have here only considered forward data refinement, i.e., refinement where the abstraction statement ($\{R\}$ in conditions (6)–(8)) is angelic. Action systems also admit backward data refinement (refinement with a demonic abstraction statement) but the kind of compositionality studied here does not hold for the backward case.

References

[Bac80] Back, R.: *Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of *Mathematical Centre Tracts*. Mathematical Centre, Amsterdam, 1980.

[BaK88] Back, R. and Kurki-Suonio, R.: Distributed co-operation with action systems. *ACM Transactions on Programming Languages and Systems*, 10:513–554, October 1988.

[BaS91] Back, R. and Sere, K.: Stepwise refinement of action systems. *Structured Programming*, 12:17–30, 1991.

[BaW94] Back, R. and von Wright, J.: Trace refinement of action systems. In B. Jonsson and J. Parrow, editors, *Proc. CONCUR-94,* volume 836 *of* Lecture Notes in Computer Science, Uppsala, Sweden, 1994. Springer-Verlag.

[BaW98] Back, R. and von Wright, J.: *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.

[BaW00] Back, R. and von Wright, J.: Encoding, decoding, and data refinement. *Formal Aspects of Computing*, 12(5):313–349, 2000.
[CoJ95] Collette, P. and Jones, C.: Enhancing the tractability of rely-guarantee specifications in the development of interfering operations. Technical report, University of Manchester, 1995.
[GaM91] Gardiner, P. and Morgan, C.: Data refinement of predicate transformers. *Theoretical Computer Science*, 87(1):143–162, 1991.
[Jon86] Jones, C.: *Systematic Software Development Using VDM*. Prentice-Hall International, 1986.
[LaW97] Långbacka, T. and von Wright, J.: Refining reactive systems in HOL using action systems. In *Proc. TPHOLs'97 – Theorem Proving in Higher Order Logics: 10th International Conference,* volume 1275 *of* Lecture Notes in Computer Science, pages 183–197, Murray Hill, NJ, August 1997. Springer-Verlag.
[OwG76] Owicki, S. and Gries, D.: Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, 1976.
[Sto91] Stølen, K.: An attempt to reason about shared-state concurrency in the style of VDM. In *VDM'91: Formal Software Development Methods*, volume 551 of *Lecture Notes in Computer Science*, pages 324–342. Springer-Verlag, 1991.
[XRH97] Xu, Q., de Roever, W.-P. and He, J.: The rely-guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9(2):149–174, 1997.

Received October 2002
Accepted April 2003