Component-Oriented Development of Action Systems

Rimvydas Rukšėnas

Turku Centre for Computer Science
TUCS Technical Report No 544
July 2003
ISBN 952-12-1203-9
ISSN 1239-1891

Abstract

We present an approach to compositional refinement of action systems and their interfaces. Our approach is intended to provide a support for the component-oriented development of action systems. We introduce a notion of *context-sensitive* simulation that is related to the rely-guarantee methods for program verification. We consider conditions under which context-sensitive simulation preserves component matching and present the corresponding theorems. A small example is used to illustrate our approach.

Keywords: interface refinement, compositionality, concurrent systems, component-based development

TUCS Laboratory
Distributed Systems Design

1 Introduction

The development of concurrent systems is especially prone to errors, since such systems usually consist of a number of simultaneously running processes that interact with each other via messages or shared variables. Component technology increases the reliability of software systems and, therefore, can ease the correct development of concurrent systems.

Component-based software engineering relies upon the existence of large libraries of software components that store the abstract specifications of components in addition to their implementations. The idea is that the designer picks a component from the library so that the provided specification of its behaviour matches (refines in our terminology) the specification to be implemented. The ease of component matching is crucial for the success of component technology. This suggests the necessity of providing components with the abstract descriptions of their intended interaction with the environment. On the other hand, abstract interfaces need not be implementable. Thus, in general, components themselves may interact with their environments via more concrete mechanisms which are derived in the refinement process. Assuming that matching between such components has been checked on the abstract level of specifications, the crucial question is whether the parallel composition of their refinements is valid. In general, the answer is negative, since the refinements of component interfaces may interfere. Thus, the main aim of this paper is to suggest an approach that guarantees the compositionality of such refinements.

We use the action system framework [2] which is a state-based formalism for reasoning about concurrent systems similar to UNITY [12] and TLA [21]. Action systems are developed in top-down manner. The initial specification (an abstract action system) can be decomposed into a number of action systems. The decomposition step introduces interface variables and a certain communication mechanism. Separate refinements of component systems using existing approaches [6] is only possible under the assumption that the system interfaces remain unchanged. Here, by interface we mean both the variables used for communication with the environment and the communication protocol itself. Until now the only way to refine interfaces within the action system framework was to compose the involved parties into one system and then refine the latter. Such an approach is infeasible for several reasons. Firstly, it violates the essence of the top-down development strategy and, thereby, increases the complexity of reasoning. Secondly, it is simply impossible when action systems are developed as reusable components, since, in this case, there is no specific party to compose an action system with.

We extend the action system framework by providing means for developing action systems in a component-oriented way. Namely, we consider an action system together with the specification of the most general environment, called a *context*, that is required by the system to work properly. When refining action systems, we refine their contexts as well. This is captured by the notion of *context-sensitive* simulation between action systems. The simulation relation guarantees that refinements are preserved in the parallel composition of action systems provided each refined system matches the context of the other. Next, we consider conditions under which component matching on the abstract level is preserved by context-sensitive simulation.

Compositional methods have been extensively studied within various frameworks for the verification of concurrent systems. The excellent overview of the work in this area is given by [27]. However, much less attention has been devoted to compositional refinement techniques for developing such systems. Our approach to refinement is related to the rely-guarantee (assumption-commitment) methods for program verification. The main novelty is a possibility to refine the interfaces of action systems in a compositional manner. We consider interface refinement in a rather general setting. It may involve both (data) refinement of shared variables and refinement of atomicity of interaction. The latter means that an atomic communication action may be replaced by several actions in the refined system. Furthermore, some communication actions from the refined system may be delegated to its environment.

As a mathematical basis we use the refinement calculus [5] which is a formalisation of the stepwise refinement approach to the derivation of sequential programs introduced by Dijkstra [13] and Wirth [33]. The refinement calculus was extended with the action system formalism [3, 4] to accommodate parallel and reactive programs . We follow their approach, thus, refinement of action systems rests upon the notion of data refinement of predicate transformers.

The remainder of the paper is organised as follows. We start from an introduction to the refinement calculus and action systems. In Section 3, the notions of an action system context and context-sensitive simulation are formalised. We also describe an example that is used to illustrate various concepts throughout the paper. In Section 4, we show how context-sensitive simulation of action systems entails refinement between the corresponding parallel compositions. Our approach is generalised for action systems with stuttering in Sections 5 and 6. Finally, we give a brief overview of the related work and some concluding remarks in Section 7.

2 Preliminaries

In this section, a brief introduction is given on how program statements are modelled as predicate transformers. The material follows [5] and is based on a higher-order logic. We write the application of function f to argument a as $f.a$.

2.1 Predicate transformers

The *state space* of program statements is modelled as a higher-order logic type Σ. Then *predicates* are state functions of type $\Sigma \to \mathrm{Bool}$ and state *relations* are binary functions of type $\Gamma \to \Sigma \to \mathrm{Bool}$. Conjunction $\cap$, disjunction $\cup$ and negation $\neg$ of predicates and relations are defined by pointwise extension of the corresponding boolean operations. The predicates $false$ and $true$ stand for universally false and true predicates. Ordering on predicates is defined using the boolean implication:

$$p \subseteq q \quad \hat{=} \quad (\forall \sigma \bullet p.\sigma \Rightarrow q.\sigma)$$

We use the standard relational composition, written as $P\,;Q$. Also, two relations can be composed using a *quotient* operator defined below:

$$(P \setminus Q).\sigma.\sigma' \quad \hat{=} \quad (\forall \gamma \bullet P.\sigma.\gamma \Rightarrow Q.\gamma.\sigma')$$

The *inverse* of relation P is written as P^{-1}.

Predicate transformers are functions of type $(\Gamma \to \mathrm{Bool}) \to (\Sigma \to \mathrm{Bool})$. We assume that program statements correspond to *monotonic* predicate transformers. Program statements can be composed using sequential composition and two (demonic and angelic, respectively) choice operators:

$$\begin{array}{rcl} (S;T).q & \hat{=} & S.(T.q) \\ (S \sqcap T).q & \hat{=} & S.q \cap T.q \\ (S \sqcup T).q & \hat{=} & S.q \cup T.q \end{array}$$

Thus, the demonic choice $S \sqcap T$ establishes a postcondition q if both S and T do so. The angelic choice $S \sqcup T$ establishes q if one of S or T does.

Basic program statements are *functional update* and two *relational updates*:

$$\begin{array}{rcl} <f>.q.\sigma & \hat{=} & q.(f.\sigma) \\ [R].q.\sigma & \hat{=} & (\forall \sigma' \bullet R.\sigma.\sigma' \Rightarrow q.\sigma') \\ \{R\}.q.\sigma & \hat{=} & (\exists \sigma' \bullet R.\sigma.\sigma' \wedge q.\sigma') \end{array}$$

Both relational updates change the program state nondeterministically according to the state relation R. If there is no σ' such that $R.\sigma.\sigma'$ holds, $[R]$ establishes any postcondition, whereas $\{R\}$ establishes no postcondition. Two special cases of updates have to do with the universally false relation $False$. Thus, $\{False\}$, denoted *abort*, always aborts, while $[False]$, denoted *magic* always behaves miraculously, i.e. it establishes any postcondition.

We have two lifting operations that form a relation from a predicate and a state function, respectively:

$$\begin{array}{rcl} |q|.\sigma.\sigma' & \hat{=} & q.\sigma \wedge (\sigma' = \sigma) \\ |f|.\sigma.\sigma' & \hat{=} & \sigma' = f.\sigma \end{array}$$

Two additional special cases of updates are *assertions* and *guards*. The assertion $\{q\}$ stands for $\{|q|\}$, whereas the guard $[q]$ denotes $[|q|]$. Finally, *skip* stands for $\{true\}$ (or $[true]$) and always leaves the state unchanged.

The *refinement* ordering on predicate transformers is the pointwise extension of predicate ordering:

$$S \sqsubseteq T \quad \hat{=} (\forall q \bullet S.q \subseteq T.q)$$

A predicate transformer is *conjunctive* if it distributes over arbitrary nonempty intersections of predicates. Predicate transformer S is (always) *terminating* if $S.true = true$.

For modelling action systems, iteration operator on program statements is important. Monotonic predicate transformers form a complete boolean lattice with $\sqcap$ as a meet, $\sqcup$ as a join, *abort* as a bottom and *magic* as a top. Therefore, any monotonic function on predicate transformers has unique least and greatest fixpoints. These can be used to define two iteration operators — *strong* iteration and *weak* iteration, respectively:

$$\begin{array}{lcl} S^{\omega} & \hat{=} & (\mu X \bullet S; X \sqcap skip) \\ S^{*} & \hat{=} & (\nu X \bullet S; X \sqcap skip) \end{array}$$

Intuitively, both iteration operators allow arbitrary number of repetitions of S. The choice to stop iteration is a prerogative of the demon. The difference between weak and strong iteration is that S^* is always finite, whereas S^{ω} can be infinite (if the demon makes such choice). In this case, strong iteration is equivalent to *abort*.

2.2 Program variables

We use the axiomatic model of program variables from [6, 5]. In this model, a program variable x is a triple of projection functions $\mathsf{add}.x$, $\mathsf{del}.x$ and $\mathsf{val}.x$. Informally, $\mathsf{add}.x.a.\sigma$ adds variable x with the value a to state σ, $\mathsf{del}.x.\sigma$ deletes x from σ, and $\mathsf{val}.x.\sigma$ gives the value of x in σ. Formally, the three functions are defined by a collection of axioms [6] which also ensure that the operations associated with distinct variables are independent. The model implicitly assumes that all possible variables are part of the program state. Thus, the function $\mathsf{add}.x.a$ can be thought of as pushing a to the stack of values associated with the variable x instead of adding the variable itself. In this paper, we usually deal with collections of program variables, e.g. local variables, global variables, etc. Therefore, the projection functions are associated with such collections instead of a single program variable.

Since various properties of action systems and their refinements are based on the fact that the local variables of one system are hidden from other systems, a notion of *independence* is essential. Thus, we say that state relation R is independent of variables x, iff $|\mathsf{add}.x.a|; R = R; |\mathsf{add}.x.a|$ for any a. Then predicate p is independent of x, iff the relation $|p|$ is such. While a predicate transformer S is independent of x, iff $<\mathsf{add}.x.a>; S = S; <\mathsf{add}.x.a>$ for any a.

We will deal with program statements that work with different program variables. Thus, transitions from a state with variables x to a state with variables y are described using a *state-replacing* relation $(+y - x \mid r)$ defined as follows:

$$\begin{array}{l} (+y - x \mid r).\sigma.\sigma' \ \hat{=} \\ \qquad \exists a \bullet \sigma' = \mathsf{add}.y.a.(\mathsf{del}.x.\sigma) \quad \wedge \quad r.a.(\mathsf{val}.x.\sigma).(\mathsf{del}.x.\sigma) \end{array}$$

Here, $r.y.x$ is a boolean expression which refer to the value of a program variable $\mathsf{val}.z$ by its name z and omits references to the program state σ. Intuitively, the relation adds y with the value a to state σ while removing x so that $r.a.(\mathsf{val}.x.\sigma)$ holds on the remaining state $\mathsf{del}.x.\sigma$. For example, $(+b - m \mid b \equiv m < n).\sigma.\sigma'$ replaces variable b in σ by a new variable m in σ' so that $\mathsf{val}.b.\sigma \equiv (\mathsf{val}.m.\sigma' < \mathsf{val}.n.\sigma)$ holds and the value of n remains unchanged in σ'.

We will use the following special cases of the state-replacing relation:

$$\begin{array}{rcl} (-y) & = & |\mathsf{del}.y| \\ (+x).\sigma.\sigma' & \equiv & (\exists a \bullet \sigma' = \mathsf{add}.x.a.\sigma) \\ (+x - y) & = & (+x);(-y) \end{array}$$

Now, more independence properties can be stated for state relations and predicate transformers. They are derived from the above independence definitions and program variable axioms. Let R be a relation independent of x. Then the following holds:

$$(-x)\,; R = R\,;(-x), \qquad (+x)\,; R = R\,;(+x) \tag{1}$$

Furthermore, both relational updates $[R]$ and $\{R\}$ are also independent of x. If S is a monotonic predicate transformer and R is a relation both independent of x, we have:

$$\begin{array}{rcll} \{-x\}\,; S & = & S\,; \{-x\} & (2) \\ \{+x\}\,; S & \sqsubseteq & S\,; \{+x\} & (3) \\ \{+x\}\,; \{R\} & = & \{R\}\,; \{+x\} & (4) \end{array}$$

Finally, let e be a state expression. Then a predicate transformer S is said to *preserve* e, iff $\{e = a\}\,; S = S\,; \{e = a\}$ for any a, where a is a logical variable (specification constant). When the preserved expression is program variables x, we have that S *reads-only* x.

2.3 Data refinement

Data refinement is a generalisation of the refinement relation between predicate transformers. It involves predicate transformers that work with different program variables. Formally, let S and S' be predicate transformers working with variables a and c, respectively. Assume that D is a monotonic predicate transformer which moves from a *concrete* state with variables c to an *abstract* state with variables a. We say that D is an *abstraction* statement. Then, S is *data-refined* by S' via D, iff

$$D\,; S \sqsubseteq S'\,; D.$$

In this paper, we deal with abstraction commands $\{R\}$ which corresponds to forward data refinement. In particular, we will use relations of the form $(+a - c \mid r)$.

For monotonic predicate transformers, forward data refinement is equivalent to $\{R\}\,; S\,; [R^{-1}] \sqsubseteq S'$. This shows that $\{R\}\,; S\,; [R^{-1}]$ is the least data refinement of

S. We will use the least data refinement of $[P]$ that is also a universally conjunctive (terminating and conjunctive) predicate transformer. Such a refinement, denoted $[P] \downarrow \{R\}$, can be calculated from P as follows:

$$[P] \downarrow \{R\} \quad = \quad [R \setminus P \, ; R^{-1}] \tag{5}$$

2.4 Action systems

Action systems is a shared-state model for concurrent programs [2]. Any nondeterminism in the model is resolved demonically, i.e. all choices are made by a computer system. Informally, an action system is the iteration of atomic actions. These can be executed in parallel under constraints that preserve their atomicity. In the interleaving model, parallel execution of such iteration is equivalent to nondeterministic sequential execution. Therefore, any collection of actions $A_1, \ldots, A_n$ can be viewed as the demonic choice between them: $A_1 \sqcap \ldots \sqcap A_n$. Since the latter is one composite action, we will consider action systems with a single action.

Thus, an action system is the following construct:

$$\mathcal{A} = \mathsf{var}\ a \mid p_a \bullet\ A^{\omega}\, ; [h_A]$$

where the *initialisation* p_a and the *exit condition* h_A are both state predicates and the *body* A is a conjunctive predicate transformer, called the *action*. We assume that the system works with *global* variables v, while a are its *local* variables. The execution of $\mathcal{A}$ starts in a state from p_a which corresponds to the initialisation of its local variables. Then, the action A is repeatedly executed. However, if the exit condition h_A holds, the demon may terminate the iteration instead of executing A. The system is blocked, if it reaches a state where A is disabled but h_A does not hold. A blocked system resumes execution, if its environment (another action system) enables A.

Parallel composition Let $\mathcal{A} = \mathsf{var}\ a \mid p_a \bullet\ A^{\omega}; [h_A]$ and $\mathcal{B} = \mathsf{var}\ b \mid p_b \bullet\ B^{\omega}; [h_B]$ be action systems with the disjoint local variables a and b. Then, the parallel composition of $\mathcal{A}$ and $\mathcal{B}$, written $\mathcal{A} \| \mathcal{B}$, is the following action system:

$$\mathsf{var}\ a, b \mid p_a \cap p_b \bullet\ (A \sqcap B)^{\omega}\, ; [h_A \cap h_B]$$

Thus, the parallel composition nondeterministically chooses between the actions A and B in each iteration which corresponds to their interleaving. The composed system may terminate only if the exit conditions of both systems hold. Note that there are no restrictions concerning the global variables of both systems which means that they can either overlap or be disjoint.

Hiding When some global variables of two action systems are used exclusively for the interaction between them, we may want to hide them in the parallel composition of the action systems. Let $\mathcal{A}$ and $\mathcal{B}$ be action systems as above. Assume that their global variables are ab, v. Then, variables ab are hidden in the following action system by making them local and initialising so that some predicate p_{ab} holds:

$$\mathsf{var}\ ab \mid p_{ab} \bullet \mathcal{A} \| \mathcal{B}$$

Refinement The action system semantics is a set of traces [4]. Informally, a trace is a sequence of values of the global variables produced by the execution of a system. The trace semantics is a basis for defining a refinement relation between action systems - that of trace refinement. However, since trace refinement is traditionally proved using simulation methods [4], we do not consider it here. Instead, action system refinement is directly defined as a forward simulation as shown below.

Let $\mathcal{A} = \mathsf{var}\ a \mid p_a \bullet\ A^\omega\ ; [h_A]$ and $\mathcal{C} = \mathsf{var}\ c \mid p_c \bullet\ C^\omega\ ; [h_C]$ be action systems with local variables a and c, respectively. Then, $(+a - c \mid r)$ is called a *simulation* relation. Now, (forward) action system simulation is defined as follows:

Definition 1 *Assume that $\mathcal{A}$ and $\mathcal{C}$ are action systems as above. Let R be a simulation relation. Then $\mathcal{A}$ is* simulated *by $\mathcal{C}$ via R, written $\mathcal{A} \leq_R \mathcal{C}$, iff*

(a) $p_c \subseteq \{R\}.p_a$,

(b) $\{R\}\,;A \sqsubseteq C\,;\{R\}$,

(c) $\{R\}\,;[h_A] \sqsubseteq [h_C]\,;\{R\}$.

Thus, simulation is basically expressed in terms of data refinement. In Section 5 we consider a more general concept – that of stuttering-insensitive simulation.

3 Context-sensitive simulation of action systems

A notion of context is first introduced for action systems to specify their potential environments. Then, context-sensitive simulation is defined.

3.1 Environment of action systems

When an action system is developed in a component-oriented manner, its precise environment is unknown. On the other hand, it is infeasible to expect and require for a system under development to work in arbitrary environment. One way to deal with such situation is to introduce rely conditions. The idea is that a rely condition specifies interferences from potential environments that have been taken into account by the developers of a system. We introduce a notion of a context to describe the potential environments of an action system.

Definition 2 *Assume that $\mathcal{A}$ is an action system with local variables a and global variables ia, v. Let I be a monotonic predicate transformer, and h_I be a predicate both independent of a. Then we say that*

$$\mathcal{I} = \lambda \mathcal{X} \cdot \mathsf{var}\ ia \mid p_{ia} \bullet\ (I^\omega; [h_I]) \| \mathcal{X}$$

is a parallel context *of* $\mathcal{A}$.

Thus, we distinguish two kinds of global variables accessible to an action system. Variables ia are called *interface* variables, while the remaining global variables v are called *observable* variables. The latter are characterised by the fact that they remain the same, during system development. In other words, an abstract specification and all its refinements maintain the same observable variables. On the other hand, interface variables are assumed to be used for describing how an action system interacts with other components and are hidden when action systems are composed. Their values are not considered as observable from outside. Furthermore, interface variables can be replaced by new ones to refine the interaction between action systems.

We refer to I as the *rely* action and h_I as the *external* exit condition. Intuitively, a rely action gives an abstract description of what transitions on the global state are tolerated by the system. Since I has access to the interface variables of $\mathcal{A}$, it can also be refined together with the latter. Note, that $\mathcal{I}.\mathcal{A}$ is an action system, which execution interleaves actions from $\mathcal{A}$ with the *rely* action I taking in this way into account possible interferences from the potential environment of $\mathcal{A}$. Thus, $\mathcal{I}$ specifies components that do not disrupt operation of $\mathcal{A}$ when composed with it. An external exit condition specifies those global states when the system expects its environment (other components) to be enabled. External exit conditions are used for stuttering-insensitive simulations and are treated in more detail in Section 6. Until then we consider contexts whose exit condition is always true ($h_I = true$) and omit the latter writing such context as $(\lambda\mathcal{X} \cdot \mathsf{var}\ ia \mid p_{ia} \bullet\ I^{\omega} \| \mathcal{X})$. In the next section we shall be more specific about how to match actual components and the context of an action system.

Example 3 (sender-receiver interaction) *To illustrate the introduced concepts, a small example of interaction between two generic asynchronous systems is used throughout the paper. One system, a* sender, *produces data and puts it on a communication channel, while the other system, a* receiver, *reads it from there. First, we give a very abstract specification of the channel and then refine it to a more concrete one. The abstract channel is modelled as a sequence, chan, of some values, and a boolean variable, b, which is used to synchronize communication. We have the following action system specification of an abstract sender:*

$$Sender \quad \hat{=} \quad Send^{\omega}\,;[h_{\mathrm{send}}]$$

where the action Send and the predicate h_{send} *are defined as:*

$$\begin{array}{lcl} Send & \hat{=} & [\neg b \wedge a]\,;[chan, b := chan', b' \mid b' \wedge |chan'| > 0] \\ h_{\mathrm{send}} & \hat{=} & \neg a \end{array}$$

Note that a is a global variable, chan and b are interface variables; the system has no local variables. When the previous interaction has been acknowledged by a receiver, *i.e. b is false, the action Send produces a new data (a nonempty sequence of values), puts it on chan and sets b to true. Finally, the exit predicate specifies that Sender operates as long as a remains true.*

Next, we specify the environment of Sender by giving an initialisation predicate for the interface variables and a rely action:

$$\begin{array}{rcl} pi & \hat{=} & \neg b \\ I_{\text{send}} & \hat{=} & [b, chan, a := b', chan', a' \bullet \ \neg b \Rightarrow (\neg b' \wedge chan' = chan)] \end{array}$$

The rely action states two things: (i) $\neg b$ remains stable after the acknowledgment, i.e. the environment is not allowed to set b to true, and (ii) the value of chan is preserved unless the environment has acknowledged communication. Note that the rely action is an unguarded statement.

3.2 Context-sensitive simulation

Now we define simulation between action systems so that their environments are taken into account. It is derived from Definition 1 by additionally considering refinement between rely actions:

Definition 4 *Let $\mathcal{A} = \mathsf{var}\ a \mid p_a \bullet\ A^{\omega} ; [h_A]$ and $\mathcal{C} = \mathsf{var}\ c \mid p_c \bullet\ C^{\omega} ; [h_C]$ be action systems. Assume that p_a and p_c are independent of the interface variables ia and ic, respectively. Let $\mathcal{I} = \lambda \mathcal{X} \cdot \mathsf{var}\ ia \mid p_{ia} \bullet\ I^{\omega} \| \mathcal{X}$ and $\mathcal{K} = \lambda \mathcal{X} \cdot \mathsf{var}\ ic \mid p_{ic} \bullet\ K^{\omega} \| \mathcal{X}$ be the contexts of $\mathcal{A}$ and $\mathcal{C}$. Let the simulation relation be $R = (+a, ia - c, ic \mid r)$. Then $\mathcal{C}$ is a* context-sensitive *simulation of $\mathcal{A}$ via R, written $\mathcal{I}[\mathcal{A}] \leq_R \mathcal{K}[\mathcal{C}]$, iff*

(a) $\{R\} ; I \sqsubseteq K ; \{R\}$,

(b) $\mathsf{var}\ ia \mid p_{ia} \bullet\ \mathcal{A}\ \leq_R\ \mathsf{var}\ ic \mid p_{ic} \bullet\ \mathcal{C}$.

Condition (b) shows that context-sensitive simulation permits the replacement of interface variables. This means that the interaction between an action system and its environment can be refined as well. Moreover, condition (a) stipulates data refinement between the abstract and concrete rely actions. Intuitively, it guarantees that, the potential environments of $\mathcal{C}$ does not interfere with its execution, provided that the potential environments of $\mathcal{A}$ did not.

Example 5 (sender refinement) *Now, we refine the abstract interaction between the sender system and its environment by a more concrete one. Let us assume that the concrete channel can transmit only single data element instead of the whole sequence. Suppose also that concrete interactions are synchronised using a two-phase handshake protocol. Thus, the concrete channel is modelled as a data variable, val, and three boolean variables — req, ack and rdy. We have the following action system for the refined sender:*

$$Sender' \quad \hat{=} \quad (Send_1' \sqcap Send_2' \sqcap Send_3')^{\omega} ; [h_{\text{send}}']$$

where the sender actions and exit predicate h'_{send} are defined as:

$$\begin{array}{rcl}
Send'_1 & \hat{=} & [(req \equiv ack) \wedge rdy \wedge a]\ ; \\
 & & [d_1, rdy := d'_1, rdy' \mid \neg rdy' \wedge |d'_1| > 0] \\
Send'_2 & \hat{=} & [(req \equiv ack) \wedge |d_1| > 0]\ ; \\
 & & [d_1, val, req := d'_1, val', req' \mid \\
 & & \quad (val' = hd.d_1) \wedge (d'_1 = tl.d_1) \wedge (req' \not\equiv req)] \\
Send'_3 & \hat{=} & [(req \equiv ack) \wedge \neg rdy \wedge (|d_1| = 0)]\ ; \\
 & & [req, rdy := req', rdy' \mid (req' \not\equiv req) \wedge rdy'] \\
h'_{\mathrm{send}} & \hat{=} & rdy \wedge \neg a
\end{array}$$

Here, we use the following operations on sequences: $|s|$ is the length of s; $hd.s$ and $tl.s$ give, respectively, the first element in s and the rest of it. Note that val, req and ack are interface variables at the concrete level. Furthermore, refinement also introduces a variable, d_1, that stores the data produced by the sender system and not transmitted through the channel as yet. At the end, d_1 is to be implemented as a local variable of the sender system, however, to illustrate our approach and for simplicity purposes, we treat d_1 as an interface variable at this stage.

The action $Send'_1$ produces new data d_1 and sets rdy to false to indicate that data transmission is not completed. Then the action $Send'_2$ transmits data items via the concrete channel one by one. It indicates a request by updating req so that $req \not\equiv ack$ holds. Similarly, a state such that $req \equiv ack$ signals that a receiver has read the data item in $chan$. Finally, when all the data has been transmitted, the action $Send'_3$ indicates this by setting rdy to true and sending the final request ($req' \not\equiv req$). Note that rdy in the exit condition h'_{send} ensures that $Sender'$ can exit only when data transmission is completed.

The described refinement of interaction between the sender and its environment can be encoded as a simulation relation $R_1 = IR \wedge R_{\mathrm{send}}$, where IR and R_{send} are defined as follows:

$$\begin{array}{rcl}
IR & \hat{=} & \neg b \equiv (rdy \wedge (req \equiv ack)) \\
 & \wedge & b \Rightarrow (chan = d_2 \frown (req \equiv ack \to d_1 \mid val :: d_1)) \\
R_{\mathrm{send}} & \hat{=} & rdy \Rightarrow |d_1| = 0
\end{array}$$

Here, the operator $\frown$ concatenates two sequences, while the operator $::$ is used to add an element both to the beginning and to the end of a sequence. We write $(p \to e_1 \mid e_2)$ to denote the conditional expression which evaluates to e_1, if p is true, and to e_2, otherwise.

The relation IR specifies essentially two things. First, the abstract acknowledgment $\neg b$ of the completed interaction is equivalent to a concrete state such that (i) the sender has transfered all the data items via the concrete channel, i.e. rdy is true, and (ii) a receiver has acknowledged the last interaction, i,e. $req \equiv ack$ must hold. Second, assuming that the abstract interaction has not been completed, i.e. b is true, *the abstract data stored in chan corresponds to the concatenation of two concrete sequences: data items that had already been read by a receiver (and stored in d_2), and data items to be transmitted as yet (d_1, if a receiver has acknowledged transmission, otherwise $val :: d_1$). Here, the variable d_2 is expected to be implemented as a local variable of a receiver system (note that all the actions and predicates of $\mathcal{S}ender'$ are independent of d_2). For simplicity, we treat it as an interface variable similarly as in the case of d_1. For reasons explained in Section 4, we assume that IR is a part of a simulation relation that proves the corresponding interface refinement of a receiver system as well. Finally, R_{send} is simply an invariant of the concrete sender.*

Using R_1, the simulation conditions for initialisation and exit predicates of Definition 4 are established:

$$pi' \subseteq \{R_1\}.pi$$
$$\{R_1\}\,;[h_{\text{send}}] \sqsubseteq [h'_{\text{send}}]\,;\{R_1\}$$

Here, $pi' \ \hat{=}\ \neg rdy \wedge (req \equiv ack) \wedge |d_1| = 0$ is assumed to be the initialisation predicate of the context of $\mathcal{S}ender'$. Also, data refinement via R_1 between the actions Send and $Send'_1$ holds:

$$\{R_1\}\,;Send \sqsubseteq Send'_1\,;\{R_1\}$$

To establish simulation between the action systems $\mathcal{S}ender$ and $\mathcal{S}ender'$, however, stuttering of the concrete action $Send'_2$ ought to be taken into account as shown in Section 6. Note that refinement of the abstract rely action I_{send} is considered in the next section.

4 Composing context-sensitive simulations

In this section, we show how context-sensitive simulations of component systems lead to the simulation between the corresponding parallel compositions. For simplicity, interface variables are assumed to be the same in both components. In practise, a component system may have several interfaces, each one associated with distinct variables. In Section 7, we briefly discuss how our approach can be generalised to handle compositions of such systems. Here, we consider the following setting.

Assume that action systems $\mathcal{A}$ and $\mathcal{B}$ share interface variables ab. Let $\mathcal{I}$ and $\mathcal{J}$ be their contexts with an initialisation predicate p_{ab}. Assume also that context-sensitive simulations

$$\mathcal{I}[\mathcal{A}] \leq_{R_1} \mathcal{K}[\mathcal{C}], \quad \mathcal{J}[\mathcal{B}] \leq_{R_2} \mathcal{L}[\mathcal{D}]$$

are established for some relations R_j, concrete systems $\mathcal{C}$, $\mathcal{D}$ and contexts $\mathcal{K}$, $\mathcal{L}$. Let cd and p_{cd} be concrete interface variables and their initialisation predicate, respectively. We show that, under certain compatibility assumptions, the following simulation holds between the corresponding parallel compositions:

$$\mathsf{var}\ ab \mid p_{ab} \bullet\ \mathcal{A} \| \mathcal{B}\ \leq_R\ \mathsf{var}\ cd \mid p_{cd} \bullet\ \mathcal{C} \| \mathcal{D}$$

Here, R is an appropriate combination of relations R_j.

4.1 Compatibility

In the parallel composition $\mathcal{A}\|\mathcal{B}$, the actual environment of $\mathcal{A}$ is the action system $\mathcal{B}$. On the other hand, context-sensitive simulation of $\mathcal{A}$ refers to some context $\mathcal{I}$. Clearly, when dealing with simulations of action systems and their parallel composition, one must guarantee that the actual environment of a component 'matches' its context (potential environment). Intuitively, this means that $\mathcal{B}$ must refine $\mathcal{I}$. Formally, we introduce a notion of compatibility between an action system and a context. For this, we first define the demonic extension of predicate transformer S with respect to variables x as follows:

$$\perp_x \times S \quad \hat{=} \quad \{-x\}\,;S\,;[+x] \tag{6}$$

Intuitively, $\perp_x \times S$ arbitrary updates variables x otherwise behaving as S. For monotonic S, $\perp_x \times S$ can be expressed in terms of the product of predicate transformers. Hence, we use the notation $\perp_x \times S$.

Definition 6 *Let $\mathcal{B}$, $\mathcal{J}$ be an action system and its context, respectively. Then $\mathcal{B}$ within the context $\mathcal{J}$ is* compatible *via q with a context $\mathcal{I}$, iff*

$$\mathcal{J}[\perp_b \times \mathcal{I}] \leq_q \mathcal{J}[\mathcal{B}]$$

where $\perp_b \times \mathcal{I}\ \hat{=}\ \mathsf{var}\ b \bullet\ (\perp_b \times I)^\omega$, and q is a predicate.

Intuitively, the definition says that the system $\mathcal{B}$ is compatible with the context $\mathcal{I}$, provided:

(a) the predicate q holds in initial states of $\mathsf{var}\ ab \mid p_{ab} \bullet \mathcal{B}$ and is also preserved by the rely action J, and

(b) the body of $\mathcal{B}$ refines I under invariant q.

Note that I is independent of the local variables of $\mathcal{B}$. Therefore, when thought of as an abstract constraint on $\mathcal{B}$, I admits arbitrary changes to the local variables of the latter. This is precisely the effect specified by $\perp_b \times I$, thus, we have the demonic extension $\perp_b \times I$ in the above definition.

Formally, these observations can expressed as the following lemma:

Lemma 7 *Let $\mathcal{B} = \mathsf{var}\ b \mid p_b \bullet\ B^\omega; [h_B]$ and $\mathcal{J}$ be an action system and its context, respectively. Assume that ab are interface variables initialised according to p_{ab}. Let $\mathcal{I}$ be another context. Let q be a predicate. Then $\mathcal{B}$ within the context $\mathcal{J}$ is compatible via q with $\mathcal{I}$, iff:*

(a) $p_{ab} \cap p_b \subseteq q$, $\{q\}\,;J \sqsubseteq J\,;\{q\}$,

(b) $\{q\}\,;\bot_b{\times}I \sqsubseteq B\,;\{q\}$.

PROOF. Using the definitions of compatibility and action system simulation. □

Example 8 (compatibility) *To illustrate the compatibility notion, we specify an abstract receiver* $\mathcal{R}eceiver = Recv^{\omega}\,;[h_{\mathrm{recv}}]$ *with a rely action* I_{recv}*:*

$$\begin{array}{rcl} Recv & \hat{=} & [b]\,;[b, chan := b', chan' \mid \neg b' \wedge consume.chan] \\ h_{\mathrm{recv}} & \hat{=} & \neg b \\ I_{\mathrm{recv}} & \hat{=} & [b, chan, a := b', chan', a' \mid b \Rightarrow (b' \;\;\wedge\;\; chan' = chan)] \end{array}$$

How the received data is processed is not important for our example, thus, the predicate consume.chan simply denotes the fact that it has been consumed in some way. Also, we assume that the same initialisation predicate pi is used in the context of $\mathcal{R}eceiver$*. Now, it is easy to see that the refinement* $I_{\mathrm{send}} \sqsubseteq Recv$ *holds for* I_{send} *defined in Example 3. Note that the invariant is simply the predicate true and, therefore, the remaining conditions from Lemma 7(a) hold trivially. This establishes compatibility between the context of* $\mathcal{S}ender$ *and the receiver system. Similarly, compatibility between the context of* $\mathcal{R}eceiver$ *and the sender can be shown.*

We will consider two cases depending on when compatibility of an action system with the corresponding context is checked. The first one postpones the compatibility check to the concrete system (implementation) level. In the second one, compatibility is established for abstract systems and is preserved by context-sensitive simulations of action systems provided certain constraints on concrete contexts are satisfied.

4.2 Composing simulations

Now, we move to composition theorems for action system simulations. First, we discuss how two simulation relations are composed so that their composition is a simulation relation for the parallel compositions of the corresponding action systems.

Let $\mathcal{A}$ and $\mathcal{B}$ be abstract action systems sharing interface variables ab. The interface variables model some interaction between the two systems. Refining this interaction normally requires replacement of the old interface variables by new ones. Let these be cd. Assume that the replacement rules are encoded as a simulation relation $Ri = (+ab - cd \mid ri)$. Clearly, the same replacement rules are to be adhered when refining both component systems. Therefore, we may assume that Ri is a part of both simulation relations, say R_1 and R_2. Additionally, refinements may also replace the local variables of an action system. We permit such a replacement to depend on new interface variables as well. Thus, $Rr_1 = (+a-c \mid rr_1)$, where rr_1 may refer to cd, is the simulation relation for the local variables of the first component. Then the combined simulation relation for $\mathcal{A}$ is defined as $R_1 \,\hat{=}\, Rr_1 \,;Ri$. Similarly, for the second component: $Rr_2 = (+b-d \mid rr_2)$ and $R_2 \,\hat{=}\, Rr_2 \,;Ri$. We show below that the following composition of R_1 and R_2 can be used to establish simulation between the parallel compositions of abstract and concrete action systems:

$$(Rr_1 \cap Rr_2)\,;Ri \;=\; Rr_2\,;R_1 \;=\; Rr_1\,;R_2 \tag{7}$$

Now we are ready to formulate composition theorems for simulations of action systems. We start from a more general one which postpones the compatibility check to the level of concrete systems.

Theorem 9 *Let $\mathcal{A}$, $\mathcal{B}$ be abstract action systems sharing interface variables ab, and let $\mathcal{I}$, $\mathcal{J}$ be their corresponding contexts such that I and J are terminating. Assume that the following context-sensitive simulations hold:*

$$\mathcal{I}[\mathcal{A}] \leq_{R_1} \mathcal{K}[\mathcal{C}]$$
$$\mathcal{J}[\mathcal{B}] \leq_{R_2} \mathcal{L}[\mathcal{D}]$$

Here $\mathcal{C}$, $\mathcal{D}$ are concrete action systems sharing interface variables cd, and $\mathcal{K}$, $\mathcal{L}$ are their contexts, while R_j are simulation relations as discussed above. Assume that the concrete systems are compatible with the corresponding contexts via some invariants q_j. Let $R = |q_1 \cap q_2|\,;(Rr_1 \cap Rr_2)\,;Ri$. Then

$$\mathsf{var}\ ab \mid p_{ab} \bullet\ \mathcal{A} \| \mathcal{B} \ \leq_R\ \mathsf{var}\ cd \mid p_{cd} \bullet\ \mathcal{C} \| \mathcal{D}$$

PROOF. First, we show how to establish data refinement between the actions A and C. From the theorem assumptions and Definitions 4, 6, we get:

$$\{R_1\}\,;A \sqsubseteq C\,;\{R_1\} \tag{8}$$
$$\{q_2\}\,;L \sqsubseteq L\,;\{q_2\} \tag{9}$$
$$\{R_2\}\,;J \sqsubseteq L\,;\{R_2\} \tag{10}$$
$$\{q_1\}\,;\{-c\}\,;L\,;[+c] \sqsubseteq C\,;\{q_1\} \tag{11}$$

Finally, note that b_0 is a specification constant that records the values of the local variables of $\mathcal{B}$ in the derivations below.

The first derivation essentially lifts data refinement via R_1 to the extended program states by permitting arbitrary state replacements $(+b-d)$ on the extended parts:

$$\begin{array}{ll} & \{R\}\,;\{b=b_0\}\,;A \\ \sqsubseteq & \{\text{ def. of } R,\ \{q\} \sqsubseteq \mathit{skip},\ (7),\ A \text{ independent of } b\ \} \\ & \{Rr_2\,;R_1\}\,;A\,;\{b=b_0\} \end{array}$$

$$
\begin{array}{ll}
\sqsubseteq & \{\ Rr_2 \subseteq (+b-d)\,;R_1,\ \text{monotonicity}\ \} \\
& \{(+b-d)\,;R_1\}\,;A\,;\{b=b_0\} \\
\sqsubseteq & \{\ \text{split, data refinement (8)}\ \} \\
& \{+b-d\}\,;C\,;\{;R_1\}\,;\{b=b_0\} \\
\sqsubseteq & \{\ C \text{ independent of } b \text{ and } d\text{, merge}\ \} \\
& C\,;\{(+b-d)\,;R_1\}\,;\{b=b_0\}
\end{array}
$$

The next derivation shows that C does not interfere with R_2:

$$
\begin{array}{ll}
& \{R\}\,;\{b=b_0\}\,;A \\
\sqsubseteq & \{\ \text{property } C \sqsubseteq \mathit{magic}\ \} \\
& \{R\}\,;\{b=b_0\}\,;\mathit{magic} \\
= & \{\ \text{property } \bot_a{\times}J \text{ terminating} \Rightarrow (\mathit{magic} = \bot_a{\times}J\,;\mathit{magic})\ \} \\
& \{R\}\,;\{b=b_0\}\,;\bot_a{\times}J\,;\mathit{magic} \\
\sqsubseteq & \{\ \bot_a{\times}J \text{ independent of } b\ \} \\
& \{R\}\,;\bot_a{\times}J\,;\{b=b_0\}\,;\mathit{magic} \\
\sqsubseteq & \{\ \{R\}{\sqsubseteq}\{q_1 \cap q_2\}\ ;\{-c\}\ ;\{+a\}\ ;\{R_2\}\text{, def. of extension}\ \} \\
& \{q_1 \cap q_2\}\,;\{-c\}\,;\{+a\}\,;\{R_2\}\,;\{-a\}\,;J\,;[+a]\,;\{b=b_0\}\,;\mathit{magic} \\
\sqsubseteq & \{\ \text{independence, } \{+a\}\,;\{-a\} = \mathit{skip}\text{, (10)}\ \} \\
& \{q_1 \cap q_2\}\,;\{-c\}\,;L\,;\{R_2\}\,;[+a]\,;\{b=b_0\}\,;\mathit{magic} \\
\sqsubseteq & \{\ \text{derivation below}\ \} \\
& C\,;\{q_1 \cap q_2\}\,;\{-c\}\,;\{R_2\}\,;[+a]\,;\{b=b_0\}\,;\mathit{magic} \\
\sqsubseteq & \{\ [+a] \sqsubseteq \{+a\}\text{, independence, merge}\ \} \\
& C\,;\{q_1 \cap q_2\}\,;\{(+a-c)\,;R_2\}\,;\{b=b_0\}\,;\mathit{magic}
\end{array}
$$

Now we combine both derivations:

$$
\begin{array}{ll}
& \{R\}\,;\{b=b_0\}\,;A \\
\sqsubseteq & \{\ \text{above derivations, lattice property}\ \} \\
& (C\,;\{(+b-d)\,;R_1\}\,;\{b=b_0\}) \sqcap (C\,;\{q_1 \cap q_2\}\,;\{(+a-c)\,;R_2\}\,;\{b=b_0\}\,;\mathit{magic}) \\
= & \{\ C \text{ conjunctive, property } S_1 \sqcap (\{q\}\,;S_2) = \{q\}\,;(S_1 \sqcap S_2)\ \} \\
& C\,;\{q_1 \cap q_2\}\,;(\{(+b-d)\,;R_1\}\,;\{b=b_0\} \sqcap \{(+a-c)\,;R_2\}\,;\{b=b_0\}\,;\mathit{magic}) \\
\sqsubseteq & \{\ \text{merge, see comment below}\ \} \\
& C\,;\{q_1 \cap q_2\}\,;\{(Rr_1 \cap Rr_2)\,;Ri\}
\end{array}
$$

Finally, the following proof rule for removing specification constants

$$
\frac{\{R\}\,;\{a=a_0\}\,;S \sqsubseteq \{R\}\,;T}{\{R\}\,;S \sqsubseteq \{R\}\,;T}
$$

and the last derivation yield the required condition:

$$\{R\}\,;A \sqsubseteq C\,;\{R\}$$

Note that the following property was used to justify one step in the second derivation:

$$\begin{array}{ll}
 & \{q_1 \cap q_2\}\,;\{-c\}\,;L \\
= & \quad \{ \text{ split assertion, } q_2 \text{ independent of } c \ \} \\
 & \{q_1\}\,;\{-c\}\,;\{q_2\}\,;L \\
\sqsubseteq & \quad \{ \text{ assumption (9) } \} \\
 & \{q_1\}\,;\{-c\}\,;L\,;\{q_2\} \\
\sqsubseteq & \quad \{ \text{ assumption (11) with shunting } S\,;[+a] \sqsubseteq S' \Rightarrow S \sqsubseteq S'\,;\{-a\} \ \} \\
 & C\,;\{q_1\}\,;\{-c\}\,;\{q_2\} \\
= & \quad \{ \ q_2 \text{ independent of } c\text{, merge assertions } \} \\
 & C\,;\{q_1 \cap q_2\}\,;\{-c\}
\end{array}$$

Also, note that the merge property

$$\{(+b-d)\,;R_1\}\,;\{b=b_0\} \sqcap \{(+a-c)\,;R_2\}\,;\{b=b_0\}\,;\mathit{magic} \sqsubseteq \{(Rr_1 \cap Rr_2)\,;Ri\}$$

relies upon the fact that *magic* makes any postcondition irrelevant, thus, the angelic update $\{(+a-c)\,;R_2\}$ is free to choose any b as long as the relation Rr_2 holds. Furthermore, the assertion $\{b = b_0\}$ in both alternatives of the demonic choice guarantees that the same b is chosen in any case. Formally, the property can be checked by expanding definitions.

Since the initialisation and exit conditions are rather obvious, we omit their proofs. $\square$

Next, we look more closely at data refinement of rely actions via simulation relations of the form used in Theorem 9. We show that the refinement condition can be expressed in terms of two simpler ones — that of data refinement with respect to the interface variables and a *noninterference* condition.

Lemma 10 *Assume that rely action I is terminating and K conjunctive, both independent of a and c. Let $R = Rr\,;Ri$ be a simulation relation such that $Rr = (+a - c \mid rr)$ and Ri as before. Assume that $\{Ri\}\,;I \sqsubseteq K\,;\{Ri\}$. Then*

$$\{R\}\,;I \sqsubseteq K\,;\{R\} \;\equiv\; (\forall a, c \bullet \{rr.a.c \cap \{Ri\}.\mathit{true}\}\,;K \sqsubseteq K\,;\{rr.a.c\})$$

PROOF. First, note that the following property is valid for $R = (+a - c \mid r)$ and monotonic predicate transformers S, T that are independent of a and c:

$$\{R\}\,;S \sqsubseteq T\,;\{R\} \;\equiv\; (\forall a, c \bullet \{r.a.c\}\,;S \sqsubseteq T\,;\{r.a.c\}) \qquad (12)$$

Its proof is heavily based on various program variables properties; since this topic is out of the scope of our paper, we omit it here.

Then, in the forward direction, we have:

$$
\begin{array}{ll}
 & \{R\}\,;I \sqsubseteq K\,;\{R\} \\
\equiv & \{\text{ def. of } R\text{, split, shunting }\} \\
 & \{Rr\}\,;\{Ri\}\,;I\,;[Ri^{-1}] \sqsubseteq K\,;\{Rr\} \\
\equiv & \{\text{ property (12) }\} \\
 & \forall a,c \bullet \{rr.a.c\}\,;\{Ri\}\,;I\,;[Ri^{-1}] \sqsubseteq K\,;\{rr.a.c\} \\
\Rightarrow & \{\text{ definitions, specialise } q := true\text{, } I \text{ terminating }\} \\
 & \forall a,c \bullet rr.a.c \cap \{Ri\}.true \subseteq K.(rr.a.c) \\
\Rightarrow & \{\ K \text{ conjunctive, correctness as refinement }\} \\
 & \forall a,c \bullet \{rr.a.c \cap \{Ri\}.true\}\,;K \sqsubseteq K\,;\{rr.a.c\}
\end{array}
$$

For the backward direction, let us assume

$$\forall a,c \bullet \{rr.a.c \cap \{Ri\}.true\}\,;K \sqsubseteq K\,;\{rr.a.c\}$$

Then:

$$
\begin{array}{ll}
 & \{rr.a.c\}\,;\{Ri\}\,;I\,;[Ri^{-1}] \\
\sqsubseteq & \{\ \{Ri\} = \{\{Ri\}.true\}\,;\{Ri\}\text{, data refinement assumption }\} \\
 & \{rr.a.c \cap \{Ri\}.true\}\,;K \\
\sqsubseteq & \{\text{ above assumption }\} \\
 & K\,;\{rr.a.c\}
\end{array}
$$

Finally, using the last result, we get:

$$
\begin{array}{ll}
 & \forall a,c \bullet \{rr.a.c\}\,;\{Ri\}\,;I\,;[Ri^{-1}] \sqsubseteq K\,;\{rr.a.c\} \\
\equiv & \{\text{ property (12) }\} \\
 & \{Rr\}\,;\{Ri\}\,;I\,;[Ri^{-1}] \sqsubseteq K\,;\{Rr\} \\
\equiv & \{\text{ shunting, merge, def. of } R\ \} \\
 & \{R\}\,;I \sqsubseteq K\,;\{R\}
\end{array}
$$

□

The lemma provides an intuitive interpretation for rely action refinements via relations $R = Rr\,;Ri$: if I is data refined via Ri to K, then the latter must not interfere with Rr for the refinement $\{R\}\,;I \sqsubseteq K\,;\{R\}$ to be valid.

Note that data refinement of rely actions allows us to strengthen environment assumptions. This might be needed for refinement of abstract interaction in the case when the simulation relation does/can not record some translation details.

Therefore, compatibility conditions are expressed in terms of the concrete systems in Theorem 9. This means that establishing them might require a lot of implementation related details. One way to deal with this problem is to introduce a notion symmetric to rely actions — that of a *guarantee* action. Such an action would be the abstract characterisation of an action system as regards its interaction with the environment. Compatibility can then be expressed in more abstract terms of rely/guarantee actions.

Instead, in this paper, we concentrate on a more component-oriented approach. Intuitively, the idea is to use the 'least' refinement of an abstract rely action as a concrete one. Then under certain conditions compatibility between abstract action systems is preserved at the concrete level.

4.3 Preserving compatibility

As mentioned earlier, the least data refinement of S is determined by the simulation relation. Thus, the simulation relation Ri, introduced earlier, is used to calculate a concrete rely action. Furthermore, refining shared-variable interaction between systems usually means that the concrete systems have more restricted access to the interface variables. In our example, the abstract sender and receiver both update the boolean variable b. However, their refinements read-only, respectively, variable ack and variables req, rdy. More generally, we consider systems that assume that their environment preserves some expression e. Accordingly, concrete rely actions are not simply the least data refinements via the simulation relation Ri but they also preserve some expression e.

More precisely, let us assume that an abstract rely action J is a universally conjunctive predicate transformer $J = [J]$. For simplicity, the same name here denotes a rely action and the corresponding state relation. Suppose that the concrete environment is expected to preserve a state expression e. Then, the concrete rely action L can defined to be the least universally conjunctive data refinement of J via Ri that also preserves the expression e, written $J \downarrow^e \{Ri\}$. Note that, for any state relation P and state expression e, $[P]$ preserves e, iff $P = P \cap Id.e$. Here the relation $Id.e$ is simply an identity with respect to the expression e: $Id.e.\sigma.\gamma \mathrel{\hat{=}} (e.\gamma = e.\sigma)$. Then, we have the following property:

$$J \downarrow^e \{Ri\} = [(Ri \setminus J\,;Ri^{-1}) \cap Id.e] \tag{13}$$

Example 11 (refining rely actions) *Let* $K_{\text{send}} \mathrel{\hat{=}} I_{\text{send}} \downarrow^{e_1} \{IR\}$*, where* $e_1 = (req, rdy, d_1)$*. Thus,* K_{send} *reads-only the interface variables* req*,* rdy *and* d_1*. We use Lemma 10 to demonstrate that data refinement between* I_{send} *and* K_{send} *via* R_1 *is indeed valid.*

By definition, K_{send} *is data refinement of* I_{send} *via* IR*. By Lemma 10, we only need to establish that* K_{send} *does not interfere with* R_{send}*. Since the variables* rdy *and* d_1 *are read-only by the rely action, noninterference is obvious.*

Now we are ready to formulate a theorem that shows how compatibility on the abstract level induces that on the concrete level:

Theorem 12 *Let $\mathcal{A}$, $\mathcal{I}$ and $\mathcal{K}$ be as in Theorem 9. Assume that the context-sensitive simulation*

$$\mathcal{I}[\mathcal{A}] \leq_R \mathcal{K}[\mathcal{C}]$$

holds for conjunctive K and simulation relation $R = Rr\,; Ri$ such that $Rr = (+a - c \mid r)$ and Ri as before. Suppose also that the abstract compatibility is valid:

$$\mathcal{I}[\perp_a \times \mathcal{J}] \leq_q \mathcal{I}[\mathcal{A}]$$

Here, $\perp_a \times \mathcal{J} = \mathsf{var}\ a \bullet\ (\perp_a \times J)^\omega$, and $q = qq \cap qi$ is such that predicate qi is a part of Ri while qq is independent of the interface variables ab. Finally, assume that $\mathcal{C}$ preserves a state expression e. Then the concrete action system $\mathcal{C}$ within the context $\mathcal{K}$ is compatible via $\{R\}.q$ with the context $\mathcal{L}$:

$$\mathcal{K}[\perp_c \times \mathcal{L}] \leq_{\{R\}.q} \mathcal{K}[\mathcal{C}]$$

where $L \;\hat{=}\; J \downarrow^e \{Ri\}$ and $\perp_c \times \mathcal{L} \;\hat{=}\; \mathsf{var}\ c \bullet\ (\perp_c \times L)^\omega$.

PROOF. See Appendix A. $\square$

Intuitively, the theorem states that once compatibility between an action system and a context has been established, it is preserved by context-sensitive simulations, provided that the concrete rely action is the least universally conjunctive data refinement of the abstract one.

Now, the second composition theorem shows when context-sensitive simulations and compatibility between abstract action systems are sufficient to deduce simulation between abstract and concrete parallel compositions. We assume that rely actions are universally conjunctive.

Theorem 13 *Let $\mathcal{A}$, $\mathcal{B}$, $\mathcal{I}$, $\mathcal{J}$ and Ri, Rr_j be as in Theorem 9. Assume that $\mathcal{A}$ and $\mathcal{B}$ are compatible with the corresponding contexts via invariants $q_j = qq_j \cap qi$. Assume also that concrete action systems $\mathcal{C}$, $\mathcal{D}$ preserve expressions e_2 and e_1, respectively. Let concrete rely actions be $K \;\hat{=}\; I \downarrow^{e_1} \{Ri\}$ and $L \;\hat{=}\; J \downarrow^{e_2} \{Ri\}$. Suppose the following context-sensitive simulations hold:*

$$\mathcal{I}[\mathcal{A}] \leq_{R_1} \mathcal{K}[\mathcal{C}]$$
$$\mathcal{J}[\mathcal{B}] \leq_{R_2} \mathcal{L}[\mathcal{D}]$$

where $R_j = Rr_j\,; Ri$. Let $R = |\{R_1\}.q_1 \cap \{R_2\}.q_2|\,;(Rr_1 \cap Rr_2)\,; Ri$. Then

$$\mathsf{var}\ ab \mid p_{ab} \bullet\ \mathcal{A} \| \mathcal{B}\ \leq_R\ \mathsf{var}\ cd \mid p_{cd} \bullet\ \mathcal{C} \| \mathcal{D}$$

PROOF. From Theorem 12 we have that $\mathcal{C}$ is compatible with the context $\mathcal{L}$ via invariant $\{R_1\}.q_1$:

$$\mathcal{K}[\perp_c \times \mathcal{L}] \leq_{\{R_1\}.q_1} \mathcal{K}[\mathcal{C}]$$

Similar condition holds for $\mathcal{D}$ and $\mathcal{K}$ as well. Then the required conclusion follows from Theorem 9. $\square$

5 Context-sensitive simulation and stuttering

So far, we have been considering simulations where each state change specified by a concrete action has its counterpart on the abstract level. In this section, our approach is extended to handle stuttering-insensitive simulations of action systems. Composition theorems for context-sensitive simulations of action systems with stuttering are presented.

5.1 Context-sensitive simulation with stuttering

5.1.1 Stuttering-insensitive simulations

Informally, stuttering is an event occuring in a system that is invisible to the observer of that system. Thus, any state change that does not affect the observable variables of an action system is an internal event invisible from outside. We say that an action A is a *stuttering* action, iff it is terminating and independent of the observable variables of an action system.

The reason why stuttering actions are distinguished is their special treatment when refining action systems. In particular, refinement can introduce state changes that are invisible from outside and are matched by skip transitions on the abstract level. However, simulations that we have been considering so far treat all actions uniformly. Before considering more general notions of simulation, we first note that any action A can be written as a demonic choice, $A = \tilde{A} \sqcap \bar{A}$, where $\bar{A}$ is the stuttering action, and $\tilde{A}$ is some action which we will call the *change* action. Since *magic* is a stuttering action, a trivial case of such decomposition would be $\tilde{A} = A$ and $\bar{A} = magic$.

Now, let A and C be actions and R be a simulation relation. Instead of data refinement $\{R\} \, ; A \sqsubseteq C \, ; \{R\}$, we consider separate refinements between the change and stuttering parts of A and C. Thus, let $\{R\} \, ; \tilde{A} \sqsubseteq \tilde{C} \, ; \{R\}$ be valid as before. However, for stuttering actions, two additional conditions are required:

$$\{R\} \, ; (\bar{A} \sqcap skip) \sqsubseteq \bar{C} \, ; \{R\} \tag{14}$$

$$\{R\}.\mu.\bar{A} \subseteq \mu.\bar{C} \tag{15}$$

They reflect the special treatment of stuttering actions. Namely, a concrete stuttering action can refine an abstract stuttering one in some states and a skip action in other states according to (14). Most frequently, the corresponding condition is of the form $\{R\} \sqsubseteq \bar{C} \, ; \{R\}$ which is a special case of (14) with $\bar{A} = magic$. The main advantage of (14) is that the simulation relation with such refinement condition for stuttering actions is transitive which is necessary for stepwise refinement of action systems. Finally, to guarantee that the effect of stuttering actions is indeed invisible, one must ensure that their iteration (stuttering iteration) can not continue forever. Thus, the condition (15) states that the stuttering iteration in concrete systems terminates, unless the corresponding iteration was already nonterminating at the abstract level. We refer to (15) as a *weak* stuttering termination condition.

5.1.2 Stuttering and communication

Since we consider the interaction between system parts via interface variables as an invisible event, the involved actions can be viewed as stuttering ones provided they do not change observable variables. Thus, stuttering actions can modify local and interface variables. Since interface variables are shared by action systems, their replacement makes it possible to refine interaction between system components. In particular, atomicity of such interaction can be refined by decomposing an atomic action into several actions. In this way, state changes are introduced at the concrete level that are matched by skip transitions on the abstract one. Since it makes no difference which system part performs such state change, a newly introduced action may be delegated from the component specified by an abstract action system to another component, the potential environment of a concrete system. An external exit condition as a part of the context is used to keep track of the states associated with such a delegation. Its meaning is explained below.

5.1.3 Simulation

Now, we are ready to define the notion of context-sensitive simulation for action systems with stuttering. We start by introducing a weaker notion — that of context-sensitive pre-simulation. Let an abstract and concrete action system be, respectively, $\mathcal{A} = \mathsf{var}\ a \mid p_a \bullet\ A^\omega\ ; [h_A]$ and $\mathcal{C} = \mathsf{var}\ c \mid p_c \bullet\ C^\omega\ ; [h_C]$. Let the interface variables of $\mathcal{A}$ and $\mathcal{C}$ be ia and ic. Assume that p_a and p_c are independent of ia and ic.

Definition 14 *Let the contexts $\mathcal{I}$ and $\mathcal{K}$ be, respectively:*

$$\lambda\mathcal{X} \bullet \mathsf{var}\ ia \mid p_{ia} \bullet\ (I^\omega; [h_I]) \| \mathcal{X}$$
$$\lambda\mathcal{X} \bullet \mathsf{var}\ ic \mid p_{ic} \bullet\ (K^\omega; [h_K]) \| \mathcal{X}$$

Let the simulation relation be $R = (+a, ia - c, ic \mid r)$. Then $\mathcal{C}$ is a context-sensitive pre-simulation *of $\mathcal{A}$ via R, written $\mathcal{I}[\mathcal{A}] \preceq_R^\natural \mathcal{K}[\mathcal{C}]$, iff*

$$\{R\}\,;\tilde{I} \sqsubseteq \tilde{K}\,;\{R\}$$
$$p_{ic} \cap p_c \subseteq \{R\}.(p_{ia} \cap p_a)$$
$$\{R\}\,;\tilde{A} \sqsubseteq \tilde{C}\,;\{R\}$$
$$\{R\}\,;[h_A \cap h_I] \sqsubseteq [h_C \cap h_K]\,;\{R\} \tag{16}$$
$$\{R\}\,;(\bar{I} \sqcap skip) \sqsubseteq \bar{K}\,;\{R\} \tag{17}$$
$$\{R\}\,;(\bar{A} \sqcap skip) \sqsubseteq \bar{C}\,;\{R\} \tag{18}$$

The first three conditions correspond to the similar ones in Definition 4. Condition (16) generalises the corresponding condition from the same definition by adding external exit predicates. To explain its meaning, we first assume that $h_I = true$. Now, suppose that $\neg h_K$ is such that the exit condition from Definition 4, $\{R\};[h_A] \sqsubseteq [h_C];\{R\}$, does not hold. Intuitively this means that, for some states in $\neg h_A$, there exist no related states in $\neg h_C$ with respect to R. In other words, termination at the concrete level is possible when that at the abstract level is not. To prevent this, (16) makes use of h_K; the environment of $\mathcal{C}$ is required to continue when h_K does not hold. In practise, external exit predicates are needed when the concrete stuttering actions associated with one abstract action are distributed into several component systems. Since $\mathcal{C}$ may be refined further, condition (16) permits arbitrary exit predicates in the abstract context. Finally, conditions (17) and (18) are needed for refinement of the stuttering parts of I and A.

Example 15 (external exit conditions) *Now, let us consider a concrete receiver,* $\mathcal{R}eceiver' \;\hat{=}\; (Recv_1' \sqcap Recv_2')^{\omega}\,;[h'_{\mathrm{recv}}]$, *which consists of the following:*

$$\begin{array}{rcl} Recv_1' & \hat{=} & [(req \not\equiv ack) \wedge \neg rdy]\,; \\ & & [d_2, ack := d_2', ack' \mid (d_2' = d_2 :: val) \wedge (ack' \not\equiv ack)] \\ Recv_2' & \hat{=} & [(req \not\equiv ack) \wedge rdy]\,; \\ & & [d_2, ack := d_2', ack' \mid consume.d_2 \wedge (ack' = \neg ack) \wedge (|d_2'| = 0)] \\ h'_{\mathrm{recv}} & \hat{=} & req = ack \end{array}$$

It has a variable, sequence d_2, that accumulates the transmitted data until the whole data package has been read. As far as $\mathcal{R}eceiver'$ concerned, the exit condition h'_{recv} allows it to terminate, provided there is no request from the environment, i.e. $req = ack$. On the other hand, the abstract receiver could not exit unless the whole data package has been transmitted. To match this constraint at the concrete level, an external exit condition is used as a part of the receiver context:

$$h_L' \;\hat{=}\; rdy$$

Thus, the environment of $\mathcal{R}eceiver'$ is required to continue when rdy is false, i.e. it must be enabled in such states.

Finally, we extend Definition 14 by adding a condition that relates the stuttering iteration in the abstract and concrete systems. In fact, we will use two conditions which lead to two notions of context-sensitive simulations for the action systems with stuttering. Actually, this is the main reason why an intermediate notion of pre-simulation has been introduced in the first place.

Definition 16 *Let $\mathcal{A}$, $\mathcal{C}$ and $\mathcal{I}$, $\mathcal{K}$ be as before. Assume that $\mathcal{I}[\mathcal{A}] \preceq_R^{\natural} \mathcal{K}[\mathcal{C}]$ is valid. Then we say that $\mathcal{C}$ is a* weak context-sensitive simulation *of $\mathcal{A}$ via R, written $\mathcal{I}[\mathcal{A}] \leq_R^{\natural} \mathcal{K}[\mathcal{C}]$, iff condition (15) holds. We say that $\mathcal{C}$ is a* strong context-sensitive simulation *of $\mathcal{A}$ via R, written $\mathcal{I}[\mathcal{A}] \lesssim_R^{\natural} \mathcal{K}[\mathcal{C}]$, iff*

$$\{R\}.\mu.\bar{A} \subseteq \mu.(\bar{K}^*;\bar{C}) \tag{19}$$

We refer to (19) as a *strong* stuttering termination condition. Since (15) in no way takes into account possible environment interferences, weak context-sensitive simulation is non-compositional. Intuitively, the reasons for this are as follows: (a) a concrete stuttering action may be a refinement of the skip action, and (b) stuttering actions are allowed to update interface variables that are shared between the composed systems. This makes it possible for the stuttering action of one component to interfere with the termination of stuttering in the other component. As a remedy for this, (19) stipulates that stuttering in a concrete system is to terminate in spite of any finite environment stuttering $\bar{K}^*$ that may proceed each execution of $\bar{C}$. Such requirement ensures that strong context-sensitive simulation of action systems is compositional. Further justifications for (19) are given in Section 6.2. In the next section, we also show that (19) does not have to be symmetric in the sense that stuttering-insensitive simulation is retained between the parallel compositions even in the case when only weak context-sensitive simulation holds for one of the component systems.

Finally, the following lemma shows that strong context-sensitive simulation entails the weak one.

Lemma 17 *Let R be a simulation relation such that $\mathcal{I}[\mathcal{A}] \lesssim^{\natural}_{R} \mathcal{K}[\mathcal{C}]$. Then*

$$\mathcal{I}[\mathcal{A}] \leq^{\natural}_{R} \mathcal{K}[\mathcal{C}]$$

PROOF. Follows from the fact that (19) implies (15). $\square$

Example 18 (stuttering in the concrete sender) *Now, we consider the simulation conditions related to the stuttering actions of $\mathcal{S}ender'$. Since there are no stuttering actions in the abstract sender, the following refinements are to be established:*

$$\{R_1\}\,;\,skip \sqsubseteq Send'_2\,;\,\{R_1\}$$
$$\{R_1\}\,;\,skip \sqsubseteq Send'_3\,;\,\{R_1\}$$

Also, we get the following stuttering termination condition:

$$\{R_1\}.true \quad \subseteq \quad \mu.(K^*_{\text{send}};(Send'_2 \sqcap Send'_3))$$

In fixpoint theory, such conditions are usually proved by using variant functions. In this case, let $|d_1| + (\neg rdy \to 1 \mid 0)$ be a variant function. It is obviously decreased by both stuttering actions. Indeed, $Send'_2$ removes an element from d_1, while $Send'_3$ sets rdy to $false$. Moreover, d_1 and rdy are read-only by the rely action K_{send} which, therefore, preserves the value of the variant function. Note that K_{send} is also a stuttering action. Hence, we make no distinction between its change and stuttering parts.

Finally, there is no need to introduce an external exit predicate for the concrete sender, since we have already checked the simpler exit condition from Definition 4. Thus, strong context-sensitive simulation between the sender systems holds:

$$\mathcal{I}_{\text{send}}[\mathcal{S}ender] \lesssim^{\natural}_{R_1} \mathcal{K}_{\text{send}}[\mathcal{S}ender']$$

6 Composing stuttering-insensitive simulations

Here, we generalise the results of Section 4 for the action systems with stuttering.

6.1 Composition theorems

First, one has to adapt the notion of compatibility to take into account stuttering actions.

Definition 19 *Let $\mathcal{B}$, $\mathcal{J}$ be an action system and its context, respectively. Assume that $\mathcal{B}$ is compatible via q with a context $\mathcal{I}$. Then, the action system $\mathcal{B}$ within the context $\mathcal{J}$ is* stuttering-compatible *via q with the context $\mathcal{I}$, iff*

(a) $\{q\} \,;\, \tilde{I} \sqsubseteq \tilde{B} \,;\, \{q\}$,

(b) $\{q\} \,;\, \bar{I} \sqsubseteq \bar{B} \,;\, \{q\}$

Note that the simple compatibility already ensures the refinement $\{q\} \,; I \sqsubseteq B \,; \{q\}$. However, to make use of the strong stuttering termination condition (19), one needs refinement between the stuttering parts of I and B. Thus, the new definition has two separate conditions for the change and stuttering actions. We also have a stuttering-compatibility lemma similar to Lemma 7:

Lemma 20 *Let $\mathcal{B} = \mathsf{var}\ b \mid p_b \bullet\ B^\omega \,;\, [h_B]$ and $\mathcal{J}$ be an action system and its context, respectively. Assume that ab are interface variables initialised according to p_{ab}. Let $\mathcal{I}$ be another context. Let q be a predicate. Then $\mathcal{B}$ within the context $\mathcal{J}$ is stuttering-compatible via q with $\mathcal{I}$, iff:*

(a) $p_{ab} \cap p_b \subseteq q$, *and* $\{q\} \,;\, J \sqsubseteq J \,;\, \{q\}$,

(b) $\{q\} \,;\, \bot_b \times \tilde{I} \sqsubseteq \tilde{B} \,;\, \{q\}$ *and* $\{q\} \,;\, \bot_b \times \bar{I} \sqsubseteq \bar{B} \,;\, \{q\}$,

(c) $\{q\} \,;\, [h_I] \sqsubseteq [h_B] \,;\, \{q\}$.

PROOF. Using the definitions of stuttering-compatibility and action system simulation. □

Intuitively, condition (c) requires $\mathcal{B}$ to continue execution unless the corresponding context $\mathcal{I}$ permits termination.

Now, we follow the same path as in Section 4 and present two composition theorems for the action systems with stuttering. We start from the case where stuttering-compatibility is established at the concrete level.

Theorem 21 *Let action systems $\mathcal{A}$, $\mathcal{B}$ share interface variables ab, and let $\mathcal{C}$, $\mathcal{D}$ share interface variables cd. Let*

$$\mathcal{I} = (\lambda \mathcal{X} \bullet \mathsf{var}\ ab \mid p_{ab} \bullet\ I^\omega \| \mathcal{X})$$
$$\mathcal{J} = (\lambda \mathcal{X} \bullet \mathsf{var}\ ab \mid p_{ab} \bullet\ J^\omega \| \mathcal{X})$$

be the contexts of $\mathcal{A}$ and $\mathcal{B}$ such that I and J are terminating. Assume that the concrete contexts $\mathcal{K}$ and $\mathcal{L}$ are, respectively:

$$\lambda\mathcal{X} \bullet \mathsf{var}\ cd \mid p_{cd} \bullet\ (K^{\omega}; [h_K]) \,\|\, \mathcal{X}$$
$$\lambda\mathcal{X} \bullet \mathsf{var}\ cd \mid p_{cd} \bullet\ (L^{\omega}; [h_L]) \,\|\, \mathcal{X}$$

Suppose that the following context-sensitive simulations hold:

$$\mathcal{I}[\mathcal{A}] \lesssim^{\natural}_{R_1} \mathcal{K}[\mathcal{C}]$$
$$\mathcal{J}[\mathcal{A}] \leq^{\natural}_{R_2} \mathcal{L}[\mathcal{D}]$$

where R_j are simulation relations as before. Assume also that $\mathcal{C}$ and $\mathcal{D}$ are stuttering-compatible with the corresponding contexts via some invariants q_j. Then

$$\mathsf{var}\ ab \mid p_{ab} \bullet\ \mathcal{A} \| \mathcal{B}\ \leq^{\natural}_{R}\ \mathsf{var}\ cd \mid p_{cd} \bullet\ \mathcal{C} \| \mathcal{D}$$

where $R = |q_1 \cap q_2| \,;(Rr_1 \cap Rr_2) \,; Ri$.

Note, that the abstract contexts do not include exit conditions. As explained earlier, external exit predicates are needed when concrete stuttering actions are delegated from a concrete system to its potential environment. Since, in this theorem, we view the abstract systems as initial specifications, and not as refinements of even more abstract action systems, the external exit predicates are assumed to be *true*. Note also that strong context-sensitive simulation is required only for one component; for the other one, the weak context-sensitive simulation is sufficient.

PROOF. Five conditions arising from Definition 14 and Definition 16 are to be established. The initialisation condition and the refinement condition for the change action are proved in the same way as in Theorem 9. For the exit condition we first note the following property of data refinement between guards:

$$(\{R\} \,; [p] \sqsubseteq [p'] \,; \{R\}) \qquad \equiv \qquad (\{R\}.(\neg\, p) \subseteq \neg\, p') \tag{20}$$

Then applying it to the exit conditions of the parallel compositions we get

$$\begin{array}{ll}
 & \{R\} \,; [h_A \cap h_B] \sqsubseteq [h_C \cap h_D] \,; \{R\} \\
= & \quad \{ \text{ above property } \} \\
 & \{R\}.\neg(h_A \cap h_B) \subseteq \neg(h_C \cap h_D) \\
= & \quad \{ \text{ distributivity, lattice prop. } \} \\
 & (\{R\}.(\neg\, h_A) \subseteq \neg\, h_C \cup \neg\, h_D) \quad \wedge \quad (\{R\}.(\neg\, h_B) \subseteq \neg\, h_C \cup \neg\, h_D)
\end{array}$$

We show how the first part of the last condition is established. The second part is proved in the same way:

$$
\begin{array}{ll}
 & \{R\}.(\neg\, h_A) \\
\subseteq & \{\text{ def. of } R \text{, monotonicity, split update }\} \\
 & \{q_2\}.(\{R_1\}.(\neg\, h_A)) \\
\subseteq & \{\text{ exit condition for simulation between } \mathcal{A} \text{ and } \mathcal{C}\text{, (20) }\} \\
 & \{q_2\}.(\neg\, h_C \cup \neg\, h_K) \\
\subseteq & \{\text{ distributivity, compatiblity between } \mathcal{K} \text{ and } \mathcal{D}\text{, Lemma 20, (20) }\} \\
 & \{q_2\}.(\neg\, h_C) \cup \neg\, h_D \\
\subseteq & \{\text{ def. of assertion, lattice prop. }\} \\
 & \neg\, h_C \cup \neg\, h_D
\end{array}
$$

The fourth condition is the refinement condition (14) for the stuttering actions:

$$\{R\}\,;((\bar{A} \sqcap \bar{B}) \sqcap skip) \sqsubseteq (\bar{C} \sqcap \bar{D})\,;\{R\}$$

Again, the derivations for the action A in the proof of Theorem 9 can be used by choosing $A \mathrel{\hat{=}} \bar{A} \sqcap skip$. Finally, the stuttering termination (15) is formulated and proved as Lemma 27 in the following section. □

As in the case of context-sensitive simulation without stuttering, compatibility at the abstract level induces that at the concrete level. However, Theorem 21 assumes that stuttering-compatibility holds at the concrete level. Since the latter involves the exit predicates of concrete contexts and these do not have counterparts at the abstract level, context-sensitive simulations do not automatically entail stuttering-compatibility between the refined systems. Thus, additional conditions for the exit predicates of concrete contexts are required. In fact, the required condition is the last one from Lemma 20. More formally, let J be such that $\tilde{J} = [J_1]$, $\bar{J} = [J_2]$.

Theorem 22 *Let $\mathcal{A}$, $\mathcal{I}$ and $\mathcal{K}$ be as in Theorem 21. Assume that the context-sensitive simulation*

$$\mathcal{I}[\mathcal{A}] \leq_R \mathcal{K}[\mathcal{C}]$$

holds for conjunctive K, $R = Rr\,;Ri$, $Rr = (+a - c \mid r)$, and Ri as before. Suppose also that the abstract compatibility is valid:

$$\mathcal{I}[\bot_a \times \mathcal{J}] \leq_q \mathcal{I}[\mathcal{A}]$$

Here, $\bot_a \times \mathcal{J} = \mathsf{var}\ a \bullet\ (\bot_a \times J)^{\omega}$, and $q = qq \cap qi$ is such that predicate qi is a part of Ri, while qq is independent of the interface variables ab. Assume that $\mathcal{C}$ preserves a state expression e. Finally, assume that the exit predicate of $\mathcal{K}$ satisfies the following condition:

$$\{\{R\}.q\}\,;[h_L] \sqsubseteq [h_C]\,;\{\{R\}.q\}$$

Then the concrete action system $\mathcal{C}$ within the context $\mathcal{K}$ is stuttering-compatible via $\{R\}.q$ with the context $\mathcal{L}$ such that $\tilde{L} \mathrel{\hat{=}} \tilde{J} \downarrow^e \{Ri\}$, $\bar{L} \mathrel{\hat{=}} (\bar{J} \sqcap skip) \downarrow^e \{Ri\}$, and $\bot_c \times \mathcal{L} \mathrel{\hat{=}} \mathsf{var}\ c \bullet\ (\bot_c \times L)^{\omega}$.

PROOF. The proof is essentially the same as for Theorem 12. By Lemma 20 we have to establish the following conditions:

$$\{\{R\}.q\}\,;K \sqsubseteq K\,;\{\{R\}.q\}$$
$$\{\{R\}.q\}\,;\bot_a{\times}\tilde{L} \sqsubseteq \tilde{C}\,;\{\{R\}.q\}, \quad \{\{R\}.q\}\,;\bot_a{\times}\bar{L} \sqsubseteq \bar{C}\,;\{\{R\}.q\}$$
$$\{\{R\}.q\}\,;[h_L] \sqsubseteq [h_C]\,;\{\{R\}.q\}$$

For the first condition, substitute $\tilde{I} \sqcap \bar{I} \sqcap skip$ for I in the corresponding derivation. Refinement between the change actions is derived in exactly the same way as in Theorem 12. For the stuttering actions substitute $\bar{J} \sqcap skip$ and $\bar{A} \sqcap skip$ for J, A, respectively, and note that the property $[P] \sqcap skip = [P \cup Id.v]$ holds for any relation P. Here $Id.v$ is an identity relation on the global variables v. The last condition is one of the theorem assumptions. □

The second composition theorem shows when context-sensitive simulations and compatibility between abstract action systems are sufficient to deduce stuttering-insensitive simulation between the abstract and concrete parallel compositions.

Theorem 23 *Let $\mathcal{A}$, $\mathcal{B}$, $\mathcal{I}$, $\mathcal{J}$ and Ri, Rr_j be as in Theorem 21. Assume that $\mathcal{A}$ and $\mathcal{B}$ are stuttering-compatible with the corresponding contexts via the invariants $q_j = qq_j \cap qi$. Assume also that concrete action systems $\mathcal{C}$, $\mathcal{D}$ preserve expressions e_2 and e_1, respectively. Let concrete rely actions K and L be such that $\tilde{K} = \tilde{I} \downarrow^{e_1} \{Ri\}$, $\bar{K} = (\bar{I} \sqcap skip) \downarrow^{e_1} \{Ri\}$, and $\tilde{L} = \tilde{J} \downarrow^{e_2} \{Ri\}$, $\bar{L} = (\bar{J} \sqcap skip) \downarrow^{e_2} \{Ri\}$. Suppose that the following context-sensitive simulations hold:*

$$\mathcal{I}[\mathcal{A}] \lesssim^{\natural}_{R_1} \mathcal{K}[\mathcal{C}]$$
$$\mathcal{J}[\mathcal{B}] \leq^{\natural}_{R_2} \mathcal{L}[\mathcal{D}]$$

Here $R_j = Rr_j\,;Ri$, and the concrete contexts $\mathcal{K}$ and $\mathcal{L}$ are, respectively:

$$\lambda\mathcal{X} \bullet \mathsf{var}\ cd \mid p_{cd} \bullet\ (K^{\omega};[h_K]) \,\|\, \mathcal{X}$$
$$\lambda\mathcal{X} \bullet \mathsf{var}\ cd \mid p_{cd} \bullet\ (L^{\omega};[h_L]) \,\|\, \mathcal{X}$$

Assume also that the exit conditions of $\mathcal{K}$ and $\mathcal{L}$ are refined by those of the action systems:

$$\{\{R_1\}.q_1\}\,;[h_L] \sqsubseteq [h_C]\,;\{\{R_1\}.q_1\}$$
$$\{\{R_2\}.q_2\}\,;[h_K] \sqsubseteq [h_D]\,;\{\{R_2\}.q_2\}$$

Let $R = |\{R_1\}.q_1 \cap \{R_2\}.q_2|\,;(Rr_1 \cap Rr_2)\,;Ri$. Then

$$\mathsf{var}\ ab \mid p_{ab} \bullet\ \mathcal{A}\|\mathcal{B} \ \leq^{\natural}_{R}\ \mathsf{var}\ cd \mid p_{cd} \bullet\ \mathcal{C}\|\mathcal{D}$$

PROOF. Using Lemma 20 and Theorem 21. □

We note that strong context-sensitive simulations can be used for both component systems in the assumptions of Theorems 21 and 23. This follows from Lemma 17.

Example 24 (composing simulations) *Now, we use Theorem 23 to establish simulation between the sender-receiver systems. Summarising Examples 8, 18, we have:*

$$\mathcal{I}_{\text{send}}[\mathcal{J}_{\text{recv}}] \leq_{true} \mathcal{I}_{\text{send}}[\mathcal{S}ender]$$
$$\mathcal{J}_{\text{recv}}[\mathcal{I}_{\text{send}}] \leq_{true} \mathcal{J}_{\text{recv}}[\mathcal{R}eceiver]$$
$$\mathcal{I}_{\text{send}}[\mathcal{S}ender] \lesssim^{\natural}_{R_1} \mathcal{K}_{\text{send}}[\mathcal{S}ender']$$

Next, simulation between the receiver systems is established. We use IR as a simulation relation and consider $Recv'_1$ as a stuttering action. Then, the following refinements are valid for the abstract and concrete receivers:

$$\begin{array}{rcl} \{IR\}\,;skip & \sqsubseteq & Recv'_1\,;\{IR\} \\ \{IR\}\,;Recv & \sqsubseteq & Recv'_2\,;\{IR\} \\ \{IR\}\,;[h_{\text{recv}}] & \sqsubseteq & [h'_{\text{recv}} \cap h'_L]\,;\{IR\} \end{array}$$

Since weak context-sensitive simulation is sufficient for the composition theorem, we get the following stuttering termination condition:

$$\{IR\}.true \quad \subseteq \quad \mu.(Recv'_1)$$

It obviously holds, since $Recv'_1$ disables itself. Then, we get $\mathcal{R}eceiver \leq^{\natural}_{IR} \mathcal{R}eceiver'$. Also, assuming that $L_{\text{recv}} = K_{\text{recv}} \downarrow^{e_2} \{IR\}$, refinement between the abstract and concrete rely actions holds by definition. Thus, weak context-sensitive simulation is valid for the receiver systems:

$$\mathcal{K}_{\text{recv}}[Receiver] \leq^{\natural}_{IR} \mathcal{L}_{\text{recv}}[Receiver']$$

Finally, recall that $h'_{\text{send}} = rdy \wedge a$, while $h'_L = rdy$. According to (20), the theorem assumption about exit conditions is satisfied:

$$\{\{IR\}.true\}\,;[h'_L] \sqsubseteq [h'_{\text{send}}]\,;\{\{IR\}.true\}$$

Since $\mathcal{S}ender'$ does not have an external exit predicate, the corresponding condition is trivial.

Summarising, we derive from Theorem 23 the following simulation between our sender-receiver systems:

$$\mathsf{var}\ ia \mid pi \bullet\ \mathcal{S}ender \| \mathcal{R}eceiver \ \ \leq^{\natural}_{R_1}\ \mathsf{var}\ ic \mid pi' \bullet\ \mathcal{S}ender' \| \mathcal{R}eceiver'$$

where $ia = (b, chan)$ and $ic = (req, ack, rdy, val, d_1, d_2)$.

6.2 Termination of stuttering

In this section, we consider in more detail conditions that allow to deduce termination of the stuttering iteration of a composite system from the stuttering termination conditions of individual systems. In other words, we are dealing with the compositionality property for stuttering termination conditions. Such consideration ought to justify the notions of the weak and strong stuttering termination introduced in the previous section.

We start from a simple example which reveals the problem under consideration. Let $B_1 \mathrel{\hat{=}} [b]; b{:=}F$ and $B_2 \mathrel{\hat{=}} [\neg b]; b{:=}T$ be stuttering actions in two concrete action systems. We assume that b is an interface variable shared by both systems. Suppose that the corresponding abstract systems did not have stuttering actions which is an equivalent way of saying that both abstract stuttering actions were *magic*. Let $\{R_i\}$ be the corresponding abstraction statements. Then both weak stuttering termination conditions $\{R_i\}.true \subseteq \mu.B_i$ hold, since each action B_i can be executed only once, and the execution of B_i simply falsifies its guard. However, the stuttering action in the composite system is $B_1 \sqcap B_2$. It is easy to see that the latter may be executed forever, since B_1 enables B_2 and vice versa. Formally, this means that

$$\mu.(B_1 \sqcap B_2) = false$$

which implies that the weak stuttering termination condition for the composite system does not hold.

The example demonstrates that weak stuttering termination is not preserved by the parallel composition of action systems. The reason being that termination condition (15) of one system does not take into account possible interference of the stuttering action from the other system. The problem is remedied by strengthening the termination condition to (19). Intuitively, the latter is satisfied only if the iteration of a stuttering action terminates when interleaved with the finite sequences of a rely action that stands for the stuttering action of another component. Formally, the strong termination condition is justified by the following two theorems.

Theorem 25 *Let S and T be conjunctive and monotonic predicate transformers, respectively. Then*

(a) $\mu.(S \sqcap T) = \mu.(S^\omega; T)$,

(b) $\mu.(S \sqcap T) = \mu.(S^*; T) \cap (S \sqcap T)^*.\mu.S$.

The first part of the theorem shows how the demonic choice operator can be removed from the fixpoint operator by replacing it with sequential composition and iteration. The following proofs rely on various properties of iteration constructs [5].

PROOF. First, we note that the following holds for the strong iteration of a monotonic predicate transformer U:

$$U^\omega.q = (\mu x \bullet U.x \cap q), \qquad U^\omega.true = \mu.U \tag{21}$$

Then (a) is derived as follows:

$$
\begin{array}{ll}
 & \mu.(S \sqcap T) \\
= & \{ \text{ (21) } \} \\
 & (S \sqcap T)^{\omega}.true \\
= & \{ \text{ decomposition } (S \sqcap T)^{\omega} = (S^{\omega}; T)^{\omega}; S^{\omega} \ \} \\
 & ((S^{\omega}; T)^{\omega}; S^{\omega}).true \\
= & \{ \text{ def. of composition, (21) } \} \\
 & (\mu x \bullet (S^{\omega}; T).x \cap S^{\omega}.true) \\
= & \{ \text{ def. of composition, } S^{\omega} \text{ conjunctive } \} \\
 & (\mu x \bullet S^{\omega}.(T.x \cap true)) \\
= & \{ \text{ lattice prop., def. of composition, } \eta \text{ conversion } \} \\
 & \mu.(S^{\omega}; T)
\end{array}
$$

For the second part we first make use of the *infinite iteration* statement defined as $U^{\infty} \ \hat{=}\ (\mu X \bullet U; X)$:

$$
\begin{array}{ll}
 & \mu.(S \sqcap T) \\
= & \{ \text{ (a) } \} \\
 & \mu.(S^{\omega}; T) \\
= & \{ \ U^{\omega} = U^{*} \sqcap U^{\infty} \text{ for conjunctive } U \text{, distributivity } \} \\
 & \mu.(S^{*}; T \sqcap S^{\infty}; T) \\
= & \{ \ S^{\infty}; T = S^{\infty} \text{ for any } T \text{, above derivation } (S^{*}; T \text{ conjunctive}) \ \} \\
 & \mu.((S^{*}; T)^{\omega}; S^{\infty}) \\
= & \{ \text{ second then first part of (21) } \} \\
 & (\mu x \bullet ((S^{*}; T)^{\omega}; S^{\infty}).x \cap true) \\
= & \{ \text{ def. of composition, (21) } \} \\
 & (\mu x \bullet (S^{*}; T)^{\omega}.(\mu.S)) \\
= & \{ \text{ fixpoint of constant function } \} \\
 & (S^{*}; T)^{\omega}.(\mu.S)
\end{array}
$$

Finally, the following derivation gives (b):

$$
\begin{array}{ll}
 & (S^{*}; T)^{\omega}.(\mu.S) \\
= & \{ \text{ (21), def. of composition } \} \\
 & ((S^{*}; T)^{\omega}; S^{\omega}).true \\
= & \{ \ U^{\omega} = \{\mu.U\}; U^{*} \text{ and } U^{\omega} = U^{*}; \{\mu.U\} \text{ for conjunctive } U \ \} \\
 & (\{\mu.(S^{*}; T)\}; (S^{*}; T)^{*}; S^{*}; \{\mu.S\}).true \\
= & \{ \text{ decomposition } (S \sqcap T)^{*} = (S^{*}; T)^{*}; S^{*} \ \}
\end{array}
$$

$$
\begin{array}{ll}
& (\{\mu.(S^*;T)\};(S\sqcap T)^*;\{\mu.S\}).true \\
= & \quad \{ \text{ def. of composition and assertion, lattice prop. } \} \\
& \mu.(S^*;T)\cap(S\sqcap T)^*.(\mu.S)
\end{array}
$$

□

The following corollary of the theorem explains our choice of weak and strong stuttering termination conditions.

Corollary 26 *Assume that predicate transformer S is conjunctive and T monotonic. Suppose that the following conditions are satisfied:*

(a) $p\subseteq(S\sqcap T).p$,

(b) $p\subseteq\mu.S$,

(c) $p\subseteq\mu.(S^*;T)$

where p is a predicate. Then $p\subseteq\mu.(S\sqcap T)$.

PROOF. We have

$$
\begin{array}{ll}
& p\subseteq\mu.(S\sqcap T) \\
= & \quad \{ \text{ Theorem 25(b) } \} \\
& p\subseteq\mu.(S^*;T)\cap(S\sqcap T)^*.(\mu.S) \\
= & \quad \{ \text{ lattice prop., assumption (c) } \} \\
& p\subseteq(S\sqcap T)^*.(\mu.S) \\
\Leftarrow & \quad \{ \text{ transitivity } \} \\
& (p\subseteq(S\sqcap T)^*.p) \quad\wedge\quad ((S\sqcap T)^*.p\subseteq(S\sqcap T)^*.(\mu.S)) \\
= & \quad \{ \ (S\sqcap T)^* \text{ monotonic, assumption (b) } \} \\
& (p\subseteq(S\sqcap T)^*.p) \\
\Leftarrow & \quad \{ \text{ correctness of weak iteration } \} \\
& (p\subseteq(S\sqcap T).p) \\
= & \quad \{ \text{ assumption (a) } \} \\
& \mathsf{T}
\end{array}
$$

□

The corollary also illustratesthe basic idea why strong termination requirement for one component is sufficient to derive the stuttering termination condition of a composite system as stated by Theorem 21. Using rely actions, the same idea is restated in a compositional manner by the lemma below which is used in the proof of Theorem 21.

Lemma 27 *Assume that the action systems $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$, $\mathcal{D}$ with the contexts $\mathcal{I}$, $\mathcal{J}$, $\mathcal{K}$, $\mathcal{L}$, and the simulation relations R_j, R are as in Theorem 21. Suppose also that the following conditions hold:*

(a) $\{R\}\,;((\bar{A}\sqcap\bar{B})\sqcap \mathit{skip}) \sqsubseteq (\bar{C}\sqcap\bar{D})\,;\{R\}$,

(b) $\{q_1\}\,;\perp_c\times\bar{L}\sqsubseteq\bar{C}\,;\{q_1\}$ *and* $\{q_2\}\,;\bar{L}\sqsubseteq\bar{L}\,;\{q_2\}$,

(c) $\{q_2\}\,;\perp_d\times\bar{K}\sqsubseteq\bar{D}\,;\{q_2\}$ *and* $\{q_1\}\,;\bar{K}\sqsubseteq\bar{K}\,;\{q_1\}$,

(d) $\{R_1\}.(\mu.\bar{A})\subseteq\mu.\bar{C}$,

(e) $\{R_2\}.(\mu.\bar{B})\subseteq\mu.(\bar{L}^*;\bar{D})$.

Then $\{R\}.(\mu.(\bar{A}\sqcap\bar{B}))\subseteq\mu.(\bar{C}\sqcap\bar{D})$.

PROOF. See Appendix 27. □

7 Conclusions and related work

7.1 Discussion

We have presented an approach to component-oriented development of action systems. Compositionality of refinements is achieved by considering action systems together with their context. A rely action can be viewed as an abstract characterisation of the environment as regards its interaction with the system. Several relations of context-sensitive simulation have been introduced for action systems. They are defined in a way that supports compositional reasoning — simulation between the parallel compositions of abstract and concrete action systems is valid provided context-sensitive simulations for both components have been established.

Our approach is geared to support refinement of interaction between systems that communicate through shared variables. It handles very general refinements of such interfaces — abstract interface variables can be replaced by concrete ones, and abstract communication protocols can be refined by more realistic ones. Furthermore, interface refinement may introduce new actions and/or refine atomicity of interaction between systems. In these cases, context-sensitive simulation of action systems with stuttering guarantees a liveness property — no infinite stuttering may occur in the concrete parallel composition.

We have not considered more advanced liveness properties of interaction between a system and its environment. To encode such properties, temporal predicate transformers [22, 24] can be used as rely actions, however, this is a future work. When formulating composition theorems, we have also assumed that both component systems have exactly the same interface variables. In reality, however, it is natural to expect only partly overlapping component interfaces. Our results are easily generalised to handle such situations. The basic idea is to partition interface variables while associating a rely action with each group of variables. Then compatibility between a system and a context can be checked in the same way as described in this paper by considering only the common interface variables (and the corresponding rely action).

The ideas presented here has originated while considering the design of asynchronous circuits using the action system framework [25, 26]. Such circuits are usually built from library components that communicate with each other via a standard protocol (four-phase handshake in our case). It turns out that context-sensitive simulation of action systems suites well for the development of asynchronous circuit components. We think that similar approaches are also applicable when developing other kinds of reusable concurrent components whose interaction is based on established communication protocols. However, more case studies are needed to identify weak and strong points of our approach.

Our emphasis was on stepwise refinement of interaction between action systems. Note that each refinement of the initial specification can in turn be viewed as a more concrete specification of the same component. As argued in [11], supplying a component with specifications at various levels of abstraction facilitates its reuse. The designer who puts components together can choose the most convenient one so that the process of matching it with the required specification is as simple as possible, perhaps even automatic.

We have considered action systems and their context-sensitive simulation within the refinement calculus. The latter identifies actions with conjunctive predicate transformers, while action systems are endowed trace semantics. The refinement calculus is a conservative extension of higher-order logic and, consequently, preserves its soundness. Moreover, since there is no distinction between syntax and semantics of actions in the refinement calculus, it does not require a separate proof theory. The presented formalism is intended to serve as a basis for developing a tool based on a theorem prover. In fact, we have already used the Refinement Calculator [9, 28] to mechanise our approach and to verify the example used in this paper.

7.2 Related work

Our approach is related to several areas of research. The first one is rely-guarantee based reasoning for shared-variable concurrency introduced in [16, 19] and later developed, for example, in [30, 31, 18]. Their work is based on Hoare-style proof systems, adapted to the concurrent setting and oriented to top-down development of programs with respect to open system specifications. Rely-guarantee paradigm and refinement of transition based specifications were combined in [17]. All these approaches concentrate on stepwise decomposition of specifications and, therefore, do not really deal with refinement of interaction as such. The interaction between systems is essentially fixed when a specification is decomposed into several component specifications. As a consequence, refinement relations reflect an open system view of specifications: refinement means weakening assumptions about the environment of a system. On the other hand, interaction involves at least two *cooperating* systems. Thus, in general, refinement of interaction entails strengthening obligations of both parts. This relates to an essentially closed view of specifications which is reflected in our definition of context-sensitive simulation. The latter resembles the context-sensitive refinement relation, which incorporates rely/guarantee reasoning, introduced by Dingel [14, 15]. Dingel's approach is more general and supports both shared-variable and message-passing concurrency. Furthermore, he develops a refinement calculus for a wide-spectrum specification language based on a compositional trace semantics. On the other hand, Dingel's refinement relation presumes that the environment of a specification is fixed. As a result, his approach does not deal with refinement of interaction.

Our context-sensitive simulation is related to the notions of conditional implementation [1] and conditional refinement[32, 8]. The latter is a generalisation of interface refinement which has been thoroughly investigated in the setting of streams and asynchronous message passing [7]. The basic idea of conditional refinement is similar, however: environment constraints can be strengthened by refinement. Besides that we work in a shared-variable setting, the main difference in our approach is as follows. Context-sensitive simulation involves two (abstract and concrete) rely actions that are related via the simulation relation. Whereas, in conditional refinement, additional assumptions about the environment are recorded as a separate condition either at the abstract [8] or the concrete [32] level.

Our approach is also reminiscent of lazy composition [29]. Since Shankar uses the latter for compositional verification, an abstract characterisation of the environment is verified to preserve component properties. Whereas, in our case, context-sensitive simulation ensures that a rely action does not interfere with component refinement.

Xu suggested using rely/guarantee conditions within the action system framework [34]. However, his approach was essentially geared to action system decomposition through refinement. He also has not considered stuttering-insensitive refinement. Compositionality of action system refinement has been investigated by Back and Wright [6]. They assume that an explicit environment system is given with its guarantee condition and invariant. The idea is similar, since a guarantee condition together with an invariant can be translated to a rely action in our approach. However, their assumption of the explicit environment does not permit a component-oriented approach to action system development. Furthermore, data refinement is restricted to the local variables of an action system; as a consequence, system interfaces can not be refined.

Also using weakest precondition predicate transformers, [23] considered interface refinement for sequential object-oriented programs. They handle it by establishing simulation between the corresponding abstract and concrete methods and using for this additional simulation relations for method parameters. Thus, their approach is essentially restricted to data refinement of method parameters. In the spirit of rely/guarantee approaches, [10] uses access roles to achieve compositionality in specifications of sequential systems with shared components. They do not consider refinement of roles, however.

In a different approach to action systems, Kurki-Suonio has dealt with compositional refinement within a closed system setting [20]. The main difference is that the refinement relation investigated there rests on superposition refinement. This means that new variables and actions modifying these can be introduced in refinement but already existing ones can not be removed from a system. Due to this, compositionality conditions become simpler.

Acknowledgements

I would like to thank Kaisa Sere for helpful discussions about various aspects of this work and Linas Laibinis for his advice on the axiomatic model of program variables.

Appendices

A Proof of Theorem 12

Expanding definitions and applying Lemma 7, we get the following from the assumptions of the theorem:

$$\{R\}\,;I \sqsubseteq K\,;\{R\} \tag{22}$$
$$\{R\}\,;A \sqsubseteq C\,;\{R\} \tag{23}$$
$$\{q\}\,;I \sqsubseteq I\,;\{q\} \tag{24}$$
$$\{q\}\,;\bot_a{\times}J \sqsubseteq A\,;\{q\} \tag{25}$$

Again, by Lemma 7, we have to establishes the following two conditions to prove the theorem:

$$\{\{R\}.q\}\,;K \sqsubseteq K\,;\{\{R\}.q\}$$
$$\{\{R\}.q\}\,;\bot_a{\times}L \sqsubseteq C\,;\{\{R\}.q\}$$

For the first condition, we have:

$$\begin{array}{ll}
 & \{q\}\,;I \sqsubseteq I\,;\{q\} \\
\Rightarrow & \{\text{ I is terminating }\} \\
 & q \subseteq I.q \\
\Rightarrow & \{\text{ monotonicity }\} \\
 & \{R\}.q \subseteq \{R\}.(I.q) \\
\Rightarrow & \{\text{ refinement (22) }\} \\
 & \{R\}.q \subseteq K.(\{R\}.q) \\
\Rightarrow & \{\text{ K is conjunctive }\} \\
 & \{\{R\}.q\}\,;K \sqsubseteq K\,;\{\{R\}.q\}
\end{array}$$

The second condition can be split into two parts:

$$\{\{R\}.q\}\,;C \sqsubseteq C\,;\{\{R\}.q\}$$
$$\{\{R\}.q\}\,;\bot_a{\times}L \sqsubseteq C$$

The first part follows from a derivation similar to the above one. For the second part, we note that actions are conjunctive predicate transformers, thus, C can be written as $\{p'\}\,;[P']$ for some predicate p' and relation P'. Then the refinement condition to be established is equivalent to the following one:

$$\{R\}.q \subseteq p' \quad\wedge\quad |\{R\}.q|\,;P'\,;(-c) \subseteq (-c)\,;((Ri \setminus J\,;Ri^{-1}) \cap Id.e) \tag{26}$$

To prove this, we rewrite assumptions of the theorem. For (23) we have:

$$\begin{array}{ll}
 & \{R\}\,;A \sqsubseteq C\,;\{R\} \\
\Rightarrow & \{\text{ monotonicity }\} \\
 & \{R\}\,;\{q\}\,;A \sqsubseteq C \\
\equiv & \{\text{ data refinement of conjunctive pred. transformers }\} \\
 & (\{R\}.(q \cap p) \subseteq p') \quad\wedge\quad (P' \subseteq R\,;|q| \setminus |p|\,;P\,;R^{-1})
\end{array} \tag{27}$$

Similarly, for (25):

$$\begin{array}{ll}
 & \{q\}\,;\bot_a{\times}J \sqsubseteq A\,;\{q\} \\
\Rightarrow & \{\text{ monotonicity }\} \\
 & \{q\}\,;\bot_a{\times}J \sqsubseteq A \\
\equiv & \{\text{ refinement of conjunctive pred. transformers }\} \\
 & (q \subseteq p) \quad\wedge\quad (|q|\,;P \subseteq (-a)\,;J\,;(+a))
\end{array} \tag{28}$$

Now, the first conjunct ($\{R\}.q \subseteq p'$) from (26) easily follows from (27) and (28). For the remaining one, we have:

$$\begin{array}{ll}
 & P'\,;(-c) \\
\subseteq & \{\text{ (27), monotonicity }\} \\
 & (R\,;|q| \setminus |p|\,;P\,;R^{-1})\,;(-c) \\
= & \{\text{ property: } q \subseteq p \Rightarrow (R\,;|q| \setminus |p|\,;S = R\,;|q| \setminus |q|\,;S) \text{ }\} \\
 & (R\,;|q| \setminus |q|\,;P\,;R^{-1})\,;(-c) \\
\subseteq & \{\text{ (28), monotonicity }\} \\
 & (R\,;|q| \setminus (-a)\,;J\,;(+a)\,;R^{-1})\,;(-c) \\
= & \{\text{ } (-c) \text{ deterministic }\} \\
 & R\,;|q| \setminus (-a)\,;J\,;(+a)\,;R^{-1}\,;(-c) \\
= & \{\text{ } R^{-1} = Ri^{-1}\,;Rr^{-1}\text{, independence }\} \\
 & R\,;|q| \setminus (-a)\,;J\,;Ri^{-1}\,;(+a)\,;Rr^{-1}\,;(-c)
\end{array}$$

$$
\begin{array}{ll}
\subseteq & \{\ Rr^{-1} \subseteq (-a)\,;(+c),\ \text{monotonicity}\ \} \\
& R\,;|q|\setminus(-a)\,;J\,;Ri^{-1}\,;(+a)\,;(-a)\,;(+c)\,;(-c) \\
= & \{\ (+x)\,;(-x)\ \text{is an identity relation}\ \} \\
& R\,;|q|\setminus(-a)\,;J\,;Ri^{-1} \\
= & \{\ (-a)\ \text{deterministic}\ \} \\
& R\,;|q|\,;(-a)\setminus J\,;Ri^{-1} \\
= & \{\ R = Rr\,;Ri,\ q = |qq|\,;|qi|,\ \text{independence}\ \} \\
& Rr\,;|qq|\,;(-a)\,;Ri\,;|qi|\setminus J\,;Ri^{-1} \\
= & \{\ Ri = Ri\,;|qi|,\ \text{property}\ Q\,;R\setminus S = Q\setminus R\setminus S\ \} \\
& Rr\,;|qq|\,;(-a)\setminus Ri\setminus J\,;Ri^{-1} \\
= & \{\ \text{property}\ Rr\,;|qq|\,;(-a) = |\{Rr\}.qq|\,;(-c),\ \} \\
& |\{Rr\}.qq|\,;(-c)\setminus Ri\setminus J\,;Ri^{-1}
\end{array}
$$

Finally, the following derivation gives (26), thus finishing the proof:

$$
\begin{array}{ll}
& |\{R\}.q|\,;P'\,;(-c) \\
= & \{\ C\ \text{preserves}\ e,\ \text{distributivity}\ \} \\
& |\{R\}.q|\,;P'\,;(-c)\ \cap\ |\{R\}.q|\,;Id.e\,;(-c) \\
\subseteq & \{\ \text{above derivation},\ P'\ \text{independent of}\ c\ \} \\
& |\{R\}.q|\,;(|\{Rr\}.qq|\,;(-c)\setminus Ri\setminus J\,;Ri^{-1})\ \cap\ |\{R\}.q|\,;(-c)\,;Id.e \\
= & \{\ \{R\}.q \subseteq \{Rr\}.qq,\ \text{property}\ q \subseteq p \Rightarrow (|q|\setminus|p|\,;S = |q|\setminus S)\ \} \\
& |\{R\}.q|\,;((-c)\setminus Ri\setminus J\,;Ri^{-1})\ \cap\ |\{R\}.q|\,;(-c)\,;Id.e \\
\subseteq & \{\ |\{R\}.q| \subseteq Id,\ \text{monotonicity}\ \} \\
& (-c)\setminus Ri\setminus J\,;Ri^{-1}\ \cap\ (-c)\,;Id.e \\
= & \{\ (-c)\ \text{deterministic} \Rightarrow (-c)\setminus R = (-c)\,;R\ \} \\
& (-c)\,;(Ri\setminus J\,;Ri^{-1})\ \cap\ (-c)\,;Id.e \\
= & \{\ \text{distributivity}\ \} \\
& (-c)\,;((Ri\setminus J\,;Ri^{-1})\cap Id.e)
\end{array}
$$

□

B Proof of Lemma 27

To prove the lemma, Corollary 26 can be used by choosing $p := \{R\}.(\mu.(\bar{A}\sqcap\bar{B}))$. For this we have to establish three conditions. For the first one, we have:

$$
\begin{array}{ll}
& \mathsf{T} \\
= & \{\ \text{assumption (a)}\ \} \\
& \{R\}\,;((\bar{A}\sqcap\bar{B})\sqcap skip) \sqsubseteq (\bar{C}\sqcap\bar{D})\,;\{R\}
\end{array}
$$

$$
\begin{array}{ll}
\Rightarrow & \{ \text{ def. of refinement, specialise with } p \ \} \\
& (\{R\} \,;\, ((\bar{A} \sqcap \bar{B}) \sqcap skip)).p \subseteq ((\bar{C} \sqcap \bar{D}) \,;\, \{R\}).p \\
= & \{ \text{ choice of } p \text{, definitions } \} \\
& p \subseteq (\bar{C} \sqcap \bar{D}).p
\end{array}
$$

The second condition is derived as follows:

$$
\begin{array}{ll}
& p \subseteq \mu.\bar{C} \\
= & \{ \text{ substitute } p \ \} \\
& \{R\}.(\mu.(\bar{A} \sqcap \bar{B})) \subseteq \mu.\bar{C} \\
\Leftarrow & \{ \text{ assumption (d), transitivity of } \subseteq \ \} \\
& \{R\}.(\mu.(\bar{A} \sqcap \bar{B})) \subseteq \{R_1\}.(\mu.\bar{A}) \\
\Leftarrow & \{ \ \{R\} \text{ and } \mu \text{ are monotonic, } \bar{A} \sqcap \bar{B} \sqsubseteq \bar{A} \text{, transitivity of } \subseteq \ \} \\
& \{R\}.(\mu.\bar{A}) \subseteq \{R_1\}.(\mu.\bar{A}) \\
\Leftarrow & \{ \ \{R\} \sqsubseteq \{R_1\} \text{ specialised with } \mu.\bar{A} \text{, transitivity of } \subseteq \ \} \\
& \mathsf{T}
\end{array}
$$

Finally, the last condition $p \subseteq \mu.(\bar{C}^*; \bar{D})$ can be split into two parts. For the first one, we have:

$$
\begin{array}{ll}
& p \subseteq q_1 \cap q_2 \cap \mu.(\bar{L}^*; \bar{D}) \circ \mathsf{del}.c \\
= & \{ \ \{R\}.q \subseteq q_1 \cap q_2 \text{ for any } q \text{, lattice prop. } \} \\
& p \subseteq \mu.(\bar{L}^*; \bar{D}) \circ \mathsf{del}.c \\
\Leftarrow & \{ \text{ substitute } p \text{, see previous derivation, transitivity of } \subseteq \ \} \\
& \{R_2\}.(\mu.\bar{B}) \subseteq \mu.(\bar{L}^*; \bar{D}) \circ \mathsf{del}.c \\
= & \{ \ q \text{ independent of } c \quad \Rightarrow \quad q \circ \mathsf{del}.c = q \quad \text{for any } q \ \} \\
& \{R_2\}.(\mu.\bar{B}) \subseteq \mu.(\bar{L}^*; \bar{D}) \\
\Leftarrow & \{ \text{ assumption (e) } \} \\
& \mathsf{T}
\end{array}
$$

The second part is derived as follows:

$$
\begin{array}{ll}
& q_1 \cap q_2 \cap \mu.(\bar{L}^*; \bar{D}) \circ \mathsf{del}.c \subseteq \mu.(\bar{C}^*; \bar{D}) \\
= & \{ \text{ see derivation below } \} \\
& q_1 \cap q_2 \cap \mu.(\bot_c \times (\bar{L}^*; \bar{D})) \subseteq \mu.(\bar{C}^*; \bar{D}) \\
\Leftarrow & \{ \text{ property } \{q\}; S \sqsubseteq T; \{q\} \Rightarrow q \cap \mu.S \subseteq \mu.T \ \} \\
& \{q_1 \cap q_2\} \,;\, (\bot_c \times (\bar{L}^*; \bar{D})) \sqsubseteq \bar{C}^* \,;\, \bar{D} \,;\, \{q_1 \cap q_2\} \\
= & \{ \text{ extension distributes into composition and iteration } \} \\
& \{q_1 \cap q_2\} \,;\, (\bot_c \times \bar{L})^* \,;\, \bot_c \times \bar{D} \sqsubseteq \bar{C}^* \,;\, \bar{D} \,;\, \{q_1 \cap q_2\} \\
\Leftarrow & \{ \text{ (data) refinement of composition } \}
\end{array}
$$

$$
\begin{array}{ll}
 & (\{q_1 \cap q_2\}\,;(\bot_c{\times}\bar{L})^* \sqsubseteq \bar{C}^*\,;\{q_1 \cap q_2\}) \quad \wedge \quad (\{q_1 \cap q_2\}\,;\bot_c{\times}\bar{D} \sqsubseteq \bar{D}\,;\{q_1 \cap q_2\}) \\
\Leftarrow & \{ \text{ (data) refinement of iteration } \} \\
 & (\{q_1 \cap q_2\}\,;(\bot_c{\times}\bar{L}) \sqsubseteq \bar{C}\,;\{q_1 \cap q_2\}) \quad \wedge \quad (\{q_1 \cap q_2\}\,;\bot_c{\times}\bar{D} \sqsubseteq \bar{D}\,;\{q_1 \cap q_2\}) \\
= & \{ \text{ see the last derivation in Theorem 9 } \} \\
 & (\{q_1 \cap q_2\}\,;\bot_c{\times}\bar{D} \sqsubseteq \bar{D}\,;\{q_1 \cap q_2\}) \\
\Leftarrow & \{ \text{ demonic extension refined by } \mathit{skip} \text{ on } c \ \} \\
 & (\{q_1 \cap q_2\}\,;\bar{D} \sqsubseteq \bar{D}\,;\{q_1 \cap q_2\}) \\
\Leftarrow & \{ \ \bot_d{\times}\bar{K} \text{ terminating, } \bar{D} \text{ conjunctive } \} \\
 & (\{q_1 \cap q_2\}\,;\bot_d{\times}\bar{K} \sqsubseteq \bar{D}\,;\{q_1 \cap q_2\}) \\
\Leftarrow & \{ \text{ as above for } \bot_c{\times}\bar{D} \text{ and } \bar{C} \ \} \\
 & \mathsf{T}
\end{array}
$$

This proves the lemma. Note that the first step in previous derivation is justified by the following property:

$$
\begin{array}{ll}
 & \mu.S \circ \mathsf{del}.x = \mu.(\bot_x{\times}.S) \\
= & \{ \text{ definition of extension } \} \\
 & \mu.S \circ \mathsf{del}.x = \mu.(\{-x\};S;[+x]) \\
= & \{ \text{ rolling rule } \mu.(S;T) = S.(\mu.(T;S)) \text{ for monotonic } S \text{ and } T \ \} \\
 & \mu.S \circ \mathsf{del}.x = \{-x\}.(\mu.(S;[+x];\{-x\})) \\
= & \{ \text{ property } [+x];\{-x\} = \mathit{skip} \text{ then def. of } \{-x\} \ \} \\
 & \mathsf{T}
\end{array}
$$

□

References

[1] M. Abadi, L. Lamport, Conjoining specifications, ACM Transactions on Programming Languages and Systems 17 (3) (1995) 507–534.

[2] R. J. R. Back, R. Kurki-Suonio, Distributed cooperation with action systems, ACM Transactions on Programming Languages and Systems 10 (4) (1988) 513–554.

[3] R. J. R. Back, K. Sere, Stepwise refinement of action systems, Structured Programming 12 (1991) 17–30.

[4] R. J. R. Back, J. von Wright, Trace refinement of action systems, in: B. Jonsson, J. Parrow (Eds.), CONCUR '94: Concurrency Theory, 5th International Conference, Vol. 836 of Lecture Notes in Computer Science, Springer-Verlag, Uppsala, Sweden, 1994, pp. 367–384.

[5] R.-J. Back, J. von Wright, Refinement Calculus: A Systematic Introduction, Graduate Texts in Computer Science, Springer-Verlag, 1998.

[6] R.-J. Back, J. von Wright, Compositional action system refinement, in: J. Derrick, E. Boiten, J. Woodcock, J. von Wright (Eds.), Electronic Notes in Theoretical Computer Science, Vol. 70, Elsevier, 2002.

[7] M. Broy, Compositional refinement of interactive systems, Journal of the ACM 44 (6) (1997) 850–891.

[8] M. Broy, K. Stølen, Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement, Monographs in Computer Science, Springer-Verlag, 2001.

[9] M. J. Butler, J. Grundy, T. Långbacka, R. Rukšėnas, J. von Wright, The Refinement Calculator: Proof support for program refinement, in: L. J. Groves, S. Reeves (Eds.), Formal Methods Pacific'97, Springer Series in Discrete Mathematics and Theoretical Computer Science, Springer-Verlag, Wellington, New Zealand, 1997.

[10] M. Büchi, R. Back, Compositional symmetric sharing in B, in: J. M. Wing, J. Woodcock, J. Davies (Eds.), Proceedings of FM'99: World Congress on Formal Methods, Vol. 1708 of Lecture Notes in Computer Science (Springer-Verlag), Springer Verlag, 1999, pp. 431–451.

[11] M. Charpentier, K. M. Chandy, Towards a compositional approach to the design and verification of distributed systems, in: J. M. Wing, J. Woodcock, J. Davies (Eds.), FM'99—Formal Methods, Volume I, Vol. 1708 of Lecture Notes in Computer Science, Springer, 1999, pp. 570–589.

[12] K. M. Chandy, J. Misra, Parallel Program Design: A Foundation, Addison-Wesley, Reading, MA, 1988, university of Texas–Austin.

[13] E. W. Dijkstra, A Discipline of Programming, Prentice Hall Series in Automatic Computation, Prentice Hall, 1976.

[14] J. Dingel, A trace-based refinement calculus for shared-variable parallel programs, in: Proceedings of the Seventh International Conference on Algebraic Methodology and Software Technology (AMAST '98), Vol. 1548 of Lecture Notes in Computer Science, Springer Verlag, Amazonia, Brazil, 1999, pp. 231–247.

[15] J. Dingel, Systematic parallel programming, Ph.d. thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, USA (1999).

[16] N. Francez, A. Pnueli, A proof method for cyclic programs, Acta Informatica 9 (2) (1978) 133–157.

[17] P. Gronning, T. Q. Nielsen, H. H. Lovengreen, Refinement and composition of transition-based rely-guarantee specifications with auxiliary variables, in: Foundations of Software Technology and Theoretical Computer Science, Vol. 472 of Lecture Notes in Computer Science, Springer, Berlin - Heidelberg - New York, 1990, pp. 332–348.

[18] H. Jifeng, X. Qiwen, A theory of state-based parallel programming by refinement, in: Proc. 1991 Refinement Workshop, Cambridge, 1991.

[19] C. B. Jones, Tentative steps towards a development method for interfering programs, Acta Informatica 4 (5) (1983) 596–619.

[20] R. Kurki-Suonio, Component and interface refinement in closed-system specifications, in: J. M. Wing, J. Woodcock, J. Davies (Eds.), FM'99—Formal Methods, Volume I, Vol. 1708 of Lecture Notes in Computer Science, Springer, 1999, pp. 134–154.

[21] L. Lamport, The temporal logic of actions, ACM Transactions on Programming Languages and Systems 16 (3) (1994) 872–923.

[22] J. J. Lukkien, Operational semantics and generalized weakest preconditions, Science of Computer Programming 22 (1–2) (1994) 137–155.

[23] A. Mikhajlova, E. Sekerinski, Class refinement and interface refinement in object-oriented programs, in: J. Fitzgerald, C. B. Jones, P. Lucas (Eds.), FME'97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997), Vol. 1313 of Lecture Notes in Computer Science, Springer-Verlag, 1997, pp. 82–101.

[24] J. M. Morris, Temporal predicate transformers and fair termination, Acta Informatica 27 (4) (1990) 287–313.

[25] J. Plosila, Self-timed circuit design - the action systems approach, Ph.d. thesis, Department of Applied Physics, University of Turku, Finland (1999).

[26] J. Plosila, R. Rukšėnas, K. Sere, Synthesis of delay-insensitive circuits, in: J. Grundy, M. Schwenke, T. Vickers (Eds.), International Refinement Workshop & Formal Methods Pacific '98, Springer Series in Discrete Mathematics and Theoretical Computer Science, Springer-Verlag, Canberra, Australia, 1998, pp. 286–305.

[27] W.-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, J. Zwiers, Concurrency Verification: Introduction to Compositional and Noncompositional Methods, Vol. 54 of Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 2001.

[28] R. Rukšėnas, J. von Wright, A tool for data refinement, in: J. Grundy, M. Newey (Eds.), Theorem Proving in Higher Order Logics: 11th International Conference, Vol. 1479 of Lecture Notes in Computer Science, Springer-Verlag, Canberra, Australia, 1998, pp. 423–441.

[29] N. Shankar, Lazy compositional verification, in: W.-P. de Roever, H. Langmaack, A. Pnueli (Eds.), Compositionality: The Significant Difference, Proceedings of the International Symposium COMPOS'97, Vol. 1536 of Lecture Notes in Computer Science, Springer Verlag, 1998, pp. 541–564.

[30] C. Stirling, A generalization of Owicki-Gries's Hoare logic for a concurrent while language, Theoretical Computer Science 58 (1-3) (1988) 347–359.

[31] K. Stølen, Method for the development of totally correct shared-state parallel programs, in: J. C. M. Baeten, J. F. Groote (Eds.), CONCUR '91: 2nd International Conference on Concurrency Theory, Vol. 527 of Lecture Notes in Computer Science, Springer-Verlag, Amsterdam, The Netherlands, 1991, pp. 510–525.

[32] K. Stølen, Refinement principles supporting the transition from asynchronous to synchronous communication, Science of Computer Programming 26 (1–3) (1996) 255–272.

[33] N. Wirth, Program development by stepwise refinement, Communications of the ACM 14 (4) (1971) 221–227.

[34] Q. Xu, On compositionality in refining concurrent systems, in: J. He, J. Cooke, P. Wallis (Eds.), BCS FACS 7th Refinement Worksho, Electronic Workshops in Computing, Springer-Verlag, 1996.

Turku Centre for Computer Science
Lemminkäisenkatu 14
FIN-20520 Turku
Finland

http://www.tucs.fi

University of Turku
- **Department of Information Technology**
- **Department of Mathematics**

Åbo Akademi University
- **Department of Computer Science**
- **Institute for Advanced Management Systems Research**

Turku School of Economics and Business Administration
- **Institute of Information Systems Science**