Using Coupled Simulations in Non-atomic Refinement

John Derrick[1] and Heike Wehrheim[2]

[1]Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, UK

[2]Universität Oldenburg, Fachbereich Informatik, 26111 Oldenburg, Germany

Abstract. Refinement is one of the most important techniques in formal system design, supporting stepwise development of systems from abstract specifications into more concrete implementations. *Non-atomic refinement* is employed when the level of granularity changes during a refinement step, i.e., whenever an abstract operation is refined into a *sequence* of concrete operations, as opposed to a single concrete operation. There has been some limited work on non-atomic refinement in Z, and the purpose of this paper is to extend this existing theory. In particular, we strengthen the proposed definition to exclude certain behaviours which only occur in the concrete specification but have no counterpart on the abstract level. To do this we use *coupled simulations*: the standard simulation relation is complemented by a second relation which guarantees the exclusion of undesired behaviour of the concrete system. These two relations have to agree at specific points (coupling condition), thus ensuring the desired close correspondence between abstract and concrete specification.

Keywords: Non-atomic refinement, action refinement, Z, coupled simulations.

1 Introduction

Refinement is one of the most important techniques in formal system design. Refinement supports the development on specifications on different levels of abstraction by supplying correctness criteria for moving between these levels. Thus a design may start with an abstract specification and use refinement to prove correctness of transformations to more concrete levels, which are closer to an actual implementation. In a state-based setting like Z [17], B [1] or VDM [11], refinement is concerned with proving correctness of changes made to either the state space or the operations within a specification, and is usually termed *data refinement* [5]. These data refinements are usually verified by using downward and upward simulations, which form a sound and jointly complete methodology for verifying refinements. In a process algebraic setting, like CSP [10], refinement (e.g., failure divergence refinement) compares the dynamic behaviour of processes: an implementation may not exhibit behaviour that the specification did not allow.

The basic assumption behind both data and failure divergence refinement is the *atomicity* of operations: during a refinement step the granularity of operations should be preserved; an abstract operation is always refined into *one* concrete operation. This assumption is, however, not always realistic. The necessity to split operations into a number of smaller steps naturally arises in system design. System models are developed at different levels of abstraction, and what seems to be atomic on an abstract level later often turns out to be composed out of several entities.

To tackle this a certain amount of existing work has been undertaken on *non-atomic refinement*, especially in the area of process algebras (therein called action refinement). Action refinement [2] allows the refinement of one operation (or action) into a complex process. Usually action refinement is defined via an *operator* in the process algebra as opposed to a *relation* between specifications. As a consequence the majority of work has been into considering whether a semantic equivalence which is a congruence with respect to action refinement can be found [3, 21, 20, 14].

There has also been limited work on non-atomic refinement in the state-based context [6]. In these approaches an abstract operation AOp is always refined into a sequence of concrete operations, say $COp_1 \fatsemi COp_2$. The simplest technique for treating such non-atomic refinements is to add a new operation *skip* to the abstract specification, and require that, apart from one, every concrete operation in the sequence refines *skip* and that the remaining operation refines AOp. This is, however, not possible in all cases. The effect of executing AOp may be distributed onto the whole sequence, for instance, inputs may be split onto several operations and outputs may have to be collected throughout the complete sequence.

Our starting point in this paper is therefore the definition of non-atomic refinement in [6] which also defines a downward simulation method to verify them. It already allows the more general case: neither COp_1 nor COp_2 have to match an abstract *skip* operation. Essentially, the (downward) simulation that is defined requires the following conditions to hold. (1) The execution of the sequence $COp_1 \fatsemi COp_2$ is equivalent (modulo a retrieve relation) to an execution of AOp; (2) Whenever the precondition of AOp is true so is that of COp_1, and (3) (immediately) after executing COp_1 the precondition of COp_2 is true. This guarantees that AOp is simulated by $COp_1 \fatsemi COp_2$ without requiring that any of the concrete operations matches an abstract *skip*.

However, there are also some limitations in this first approach to a truly non-atomic refinement. For some specifications the conditions are too liberal (and we will see examples of such specifications throughout the paper), and allow concrete specifications which show certain behaviours that lack an abstract counterpart. The two main difficulties which are not captured yet are the following: the sequence $COp_1 \fatsemi COp_2$ may be started "in the middle" (i.e., COp_2 can be executed without COp_1 having occurred) and it may not be completed (i.e. no COp_2 after a COp_1). The second case seems to be excluded by condition (3) of non-atomic refinement. However, this condition does not require that COp_2's precondition remains true until the actual execution.

In this paper, we therefore propose a new definition of *coupled* non-atomic refinement for Z which settles these issues. The basic idea is to complement the existing simulation relation for non-atomic refinement with a second simulation relation which excludes undesired behaviours. These two relations have to be *coupled* (i.e., agree) at specific points, thus ensuring a tight correspondence of concrete and abstract specification. The idea of using coupled simulations (originally from Parrow and Sjödin [12]) for non-atomic refinement (or, for the opposite direction, *contraction*) has been proposed by Rensink [13]. While there are technical differences to our definition (e.g., concerning the semantic model: state-based formalism vs. transition system) the basic principle is the same. The second simulation relation avoids incomplete sequences and starts in the middle, and it is coupled with the first relation on *idle* states, i.e., states, in which all refinement sequences of abstract operations are completed.

The ultimate aim of this work is to link these ideas to those arising in a behaviour-oriented setting. That is, the definition of non-atomic refinement can be related to action refinement as occurring in a failure-divergence semantics. The aim will then be to define non-atomic refinement for specifications written in a combination of CSP and a state-based language such as Z or Object-Z. There are a number of proposals for semantic integrations of these specification languages [15, 8, 9, 19, 16], and there has been much interest in definitions of refinement for these integrations, and ultimately we see our work as contributing to this study. However, here we will concentrate on the basic definition in a state-based language, leaving these additional issues for the future.

The paper is structured as follows. In Section 2 we discuss the basics of both standard and non-atomic refinement as verified by simulations. In Section 3 we introduce the idea of a coupled simulation and present a schema calculus formulation of it. Section 4 extends this formulation with input and output transformation, necessary for the most general case of non-atomic refinement. Finally, in Section 5 we conclude.

2 Background

The setting in which our work is placed is that provided by a state-based language such as Z or Object-Z. We will, in fact, use the syntax and schema calculus as provided by Z, but crucially we will assume a 'blocking' model of operation preconditions. That is, we assume an operation is not possible outside its precondition, thus the precondition acts as a guard. This is also the interpretation of operations in Object-Z[1].

[1] We make this assumption because it is convenient to fix a single interpretation, and this interpretation is consistent with a process algebraic model, e.g., as in CSP. This

Refinement is a development methodology that allows one to move from abstract specifications towards an eventual implementation. Refinement is based upon the idea that valid developments are those that reduce non-determinism present in an abstract specification. In state-based languages such as Z, Object-Z, VDM etc, the standard approach to making verification tractable is to use *simulations*, see, for example, the use of upward and downward simulations in Z and Object-Z as described in [22, 7].

In an upward or downward simulation, a retrieve relation R links the abstract state ($AState$) and the concrete state ($CState$), and requires that the concrete specification simulates the abstract specification. The advantage of these simulation methods is that they allow refinements to be verified on a step-by-step basis, i.e., there are individual conditions for the operations and the initialisation and, in particular, consideration of the complete program behaviour is not required.

The standard conditions [7] for a downward simulation (using a blocking model) are as follows. (Throughout the paper we assume specifications consist of a state space $State$, an initialisation $Init$ together with a collection of operations.)

Definition 1. *Downward simulation*
A specification C is a downward simulation of the specification A if there is a retrieve relation R such that every abstract operation AOp is recast into a concrete operation COp and the following hold.

$$\begin{array}{l}
\forall\, CInit \bullet (\exists\, AInit \bullet R) \\
\forall\, AState;\ CState;\ CState' \bullet R \land COp \Longrightarrow (\exists\, AState' \bullet R' \land AOp) \\
\forall\, AState;\ CState \bullet R \Longrightarrow (pre\, AOp \Longleftrightarrow pre\, COp)
\end{array}$$

As we have already discussed there is interest in defining a correct notion of non-atomic refinement where an abstract operation is refined by not one, but by a sequence of concrete operations thus allowing a change of granularity when we develop a specification. Such non-atomic refinements are useful since they allow an abstract specification to be described independently of the structure of the eventual implementation.

It is a relatively simple task to adapt Definition 1 to such a situation. Instead of a single concrete operation COp we refine into a sequence $COp_1 \fatsemi COp_2$. Clearly, the equivalent of the above conditions must hold, it is also sensible to require that immediately after executing COp_1 the precondition of COp_2 is true. We are thus led to the following conditions (which are justified and derived formally from a relational semantics in [7]).

Definition 2. *Non-atomic downward simulation without IO transformations*
A specification C is a non-atomic downward simulation of the specification A if there is a retrieve relation R such that every abstract operation AOp is recast

is desirable since we are ultimately interested in combinations of notations across paradigms.

into a sequence of concrete operations $COp_1 \fatsemi COp_2$ *and, in addition to the initialisation, the following hold.*

$$\forall AState;\ CState;\ CState' \bullet R \wedge (COp_1 \fatsemi COp_2) \Rightarrow \exists AState' \bullet R' \wedge AOp$$
$$\forall AState;\ CState \bullet R \Rightarrow (\mathrm{pre}\, AOp \iff \mathrm{pre}\, COp_1)$$
$$\forall AState;\ CState;\ CState' \bullet R \wedge COp_1 \Rightarrow (\mathrm{pre}\, COp_2)'$$

The conditions reflect the intuition described in the introduction. The first condition says that the effect of $COp_1 \fatsemi COp_2$ is consistent with that of AOp (but can of course reduce any non-determinism in AOp). The second says that COp_1 can be invoked whenever AOp can, and the third says that when COp_1 has been completed COp_2 can be invoked (we write $(\mathrm{pre}\, COp_2)'$ so the before-state matches the after-state of COp_1). Informally these are clearly the correct[2] conditions for a refinement of AOp into $COp_1 \fatsemi COp_2$.

As an example, consider the abstract specification containing inverse and translate operations on coordinates x and y given in Figure 1. Then according to

$$\begin{array}{|l} \hline \textit{AState} \\ x_A, y_A : \mathbb{Z} \\ \hline \end{array}$$

$$\begin{array}{|l} \hline \textit{AInit} \\ AState' \\ \hline x'_A = 0 \wedge y'_A = 0 \\ \hline \end{array}$$

$$\begin{array}{|l} \hline \textit{Translate} \\ \Delta AState \\ \hline x'_A = x_A + 3 \wedge y'_A = y_A + 2 \\ \hline \end{array}$$

$$\begin{array}{|l} \hline \textit{AObserve} \\ \Xi AState \\ x!, y! : \mathbb{Z} \\ \hline x! = x_A \wedge y! = y_A \\ \hline \end{array}$$

$$\begin{array}{|l} \hline \textit{Inverse} \\ \Delta AState \\ \hline x'_A = -x_A \wedge y'_A = -y_A \\ \hline \end{array}$$

Fig. 1. *ATranslate*

Definition 2, the specification in Figure 2 is a valid non-atomic refinement, and it is trivial to check that the required conditions hold. However, this specification, viewed as a non-atomic refinement, has some shortcomings. In particular, the concrete components $TransX_0$ and $TransY_0$ can be invoked an arbitrary number of times in any order. In other words, we have failed to capture the requirement that we can't do $TransY_0$ unless $TransX_0$ has already happened at

[2] These conditions generalise to a non-atomic refinement with an arbitrary number of concrete operations in the obvious manner.

$CState_0$

$$x_C, y_C : \mathbb{Z}$$

$CInit_0$

$$CState'_0$$

$$x'_C = 0 \land y'_C = 0$$

$TransX_0$

$$\Delta CState_0$$

$$x'_C = x_C + 3 \land y'_C = y_C$$

$TransY_0$

$$\Delta CState_0$$

$$x'_C = x_C \land y'_C = y_C + 2$$

$CObserve_0$

$$\Xi CState_0$$
$$x!, y! : \mathbb{Z}$$

$$x! = x_C \land y! = y_C$$

Inv_0

$$\Delta CState_0$$

$$x'_C = -x_C \land y'_C = -y_C$$

Fig. 2. $CTranslate_0$

some point in the past. This deficiency we will rectify in the definition we derive below (Definition 4).

There are other potential deficiencies though. For example, suppose we tackle the above problem by inserting Booleans (we assume the existence of a Boolean type) to control applicability. This is depicted in Figure 3, where we have also decomposed the inverse operation. The problem is solved. The non-atomic components can only be invoked in the correct order, although clearly it would still be possible to construct a specification where after $TransX_1$ it wasn't *always* possible to do $TransY_1$ (by adding another operation that disables it).

However, we now have an additional problem: the concrete non-atomic operations can be interleaved in such a way that they do not match any sequence of abstract operations. For example, $TransX_1 \mathbin{⨾} InvX_1 \mathbin{⨾} TransY_1 \mathbin{⨾} InvY_1$ is a valid sequence in the concrete specification, as is $TransX_1 \mathbin{⨾} InvX_1 \mathbin{⨾} InvY_1 \mathbin{⨾} TransY_1$. Whereas the first sequence refines $Translate \mathbin{⨾} Inverse$ the second sequence has no abstract counterpart (using a retrieve relation R which relates x_C with x_A and y_C with y_A).

The problem arises because the concrete specification has allowed interleavings which weren't possible in the abstract specification. *Translate* and *Inverse* are not independent since they modify the same variables. Thus a "concurrent" execution of their refinements, i.e., an interleaving of the concrete operations, may in principle lead to states with no matching abstract counterpart. This we will also seek to remedy in Definition 4 below.

Nevertheless, non-atomic refinement should not completely forbid interleavings on the concrete level. The next example shows a non-atomic refinement with acceptable interleavings. This example specifies a simple protocol between two

CState$_1$

$$x_C, y_C : \mathbb{Z}$$
$$b, c : \mathbb{B}$$

CInit$_1$

$$CState_1'$$
$$x_C' = 0 \wedge y_C' = 0 \wedge \neg b' \wedge \neg c'$$

TransX$_1$

$$\Delta CState_1$$
$$x_C' = x_C + 3 \wedge y_C' = y_C$$
$$\neg b \wedge b' \wedge c' = c$$

TransY$_1$

$$\Delta CState_1$$
$$x_C' = x_C \wedge y_C' = y_C + 2$$
$$b \wedge \neg b' \wedge c' = c$$

InvX$_1$

$$\Delta CState_1$$
$$x_C' = -x_C \wedge y_C' = y_C$$
$$\neg c \wedge c' \wedge b' = b$$

InvY$_1$

$$\Delta CState_1$$
$$x_C' = x_C \wedge y_C' = -y_C$$
$$c \wedge \neg c' \wedge b' = b$$

CObserve$_1$

$$\Xi CState_1$$
$$x!, y! : \mathbb{Z}$$
$$x! = x_C \wedge y! = y_C$$

Fig. 3. *CTranslate*$_1$

users sending and receiving messages. A message consists of the actual text and a time-stamp, where $Time == \mathbb{N}$ and $M == Time \times Text$

AState

$$q : \mathrm{seq}\, M$$

AInit

$$AState'$$
$$q' = \langle\, \rangle$$

Send

$$\Delta AState$$
$$m? : M$$
$$q' = q \frown \langle m? \rangle$$

Receive

$$\Delta AState$$
$$m! : M$$
$$q' = tail\ q$$
$$m! = head\ q$$

This specification can be considered as a description of a protocol on a certain layer in the ISO/OSI reference model. Next we look at the protocol on a lower layer and develop a more concrete specification. In this layer, the send and receive operations are split: *Send* into *PrepareToSend* $\fatsemi$ *Transmit* and *Receive* into *PrepareToRec* $\fatsemi$ *Deliver*. *PrepareToSend* stores the message to be sent in a sequence *inStore* which contains just a single place. The time-stamp is only added when the message is actually transmitted to the link between the users. The transmission adds additional information to the message, in our case a *checksum*. Before delivering a message this additional information has to be removed (and for this the message is first stored in *outStore*).

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CState}} \\
\hline
\textit{inStore} : \operatorname{seq} \textit{Text} \\
\textit{outStore} : \operatorname{seq} M \\
\textit{link} : \operatorname{seq}(M \times \mathbb{N}) \\
\hline
\#\textit{inStore} \leq 1 \wedge \#\textit{outStore} \leq 1 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CInit}} \\
\hline
\textit{CState}' \\
\hline
\textit{inStore}' = \langle\,\rangle \wedge \textit{link}' = \langle\,\rangle \\
\textit{outStore}' = \langle\,\rangle \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{PrepareToSend}} \\
\hline
\Delta \textit{CState} \\
\textit{text}? : \textit{Text} \\
\hline
\textit{inStore} = \langle\,\rangle \\
\textit{inStore}' = \textit{inStore} \frown \langle \textit{text}? \rangle \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\textit{PrepareToRec}} \\
\hline
\Delta \textit{CState} \\
\hline
\textit{outStore} = \langle\,\rangle \\
\textit{outStore}' = \textit{head}\ \textit{link} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Transmit}} \\
\hline
\Delta \textit{CState} \\
t? : \textit{Time} \\
\hline
\textit{inStore} \neq \langle\,\rangle \\
\textit{inStore}' = \textit{tail}\ \textit{inStore} \\
\textit{link}' = \textit{link} \frown \langle ((t?, \textit{head}\ \textit{inStore}), \textit{checksum}(t?, \textit{head}\ \textit{inStore})) \rangle \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Deliver}} \\
\hline
\Delta \textit{CState} \\
\textit{msg}! : M \\
\hline
\textit{outStore} \neq \langle\,\rangle \\
\textit{outStore}' = \textit{tail}\ \textit{outStore} \\
\textit{msg}! = (\textit{first}\ \textit{head}\ \textit{outStore}, \textit{second}\ \textit{head}\ \textit{outStore}) \\
\hline
\end{array}
$$

The concrete specification is a non-atomic refinement of the abstract, where q is related to *link* projected down to the first and second component of the triples, as is documented via the retrieve relation R.

$$
\begin{array}{|l}
\multicolumn{1}{l}{R} \\
\hline
AState \\
CState \\
\hline
\#q = \#link \\
\forall\, i : 1..\#q \bullet q.i = (link.i.1.1, link.i.1.2) \\
inStore = \langle\,\rangle \\
outStore = \langle\,\rangle \\
\hline
\end{array}
$$

Now all possible concrete interleavings are valid. For example, $PrepareToSend \mathbin{;} PrepareToRec \mathbin{;} Transmit \mathbin{;} Deliver$ yields, if started with a non-empty $link$, the same as $PrepareToSend \mathbin{;} PrepareToRec \mathbin{;} Deliver \mathbin{;} Transmit$. This is due to the fact that abstract sends and receives can be interleaved, that is, with non-empty q, $Send \mathbin{;} Receive$ is the same as $Receive \mathbin{;} Send$.

Thus our definition of non-atomic refinement will have to distinguish when such interleavings are permissible, or when they are not.

Notice that this refinement is also general in other respects:

- none of the concrete non-atomic components (e.g., *PrepareToSend*) are a simple stuttering step (i.e., equal to *skip*).
- the ordering between the concrete non-atomic components is achieved by use of a common sequence which the first operations fills and the second empties.
- the input in *Send* is split into two inputs distributed onto *PrepareToSend* and *Transmit*.

To deal with the latter point we need to use the notion of IO transformations. Indeed, Definition 2 is labelled 'without IO transformations' because it requires that any inputs or outputs in the abstract operation are not distributed across the concrete operations. Again this is not always realistic, and often we need to verify a non-atomic refinement where, for example, the inputs or outputs in the abstract operation are split across the individual concrete operations (e.g., as in the above example). To support such a scenario in the schema calculus we need to use *IO transformers*.

2.1 Using IO transformers

IO transformers are a mechanism to alter the input and output in an IO refinement. IO refinement [7, 4, 18] is a generalisation of the standard simulation rules, which require identities between the concrete and abstract operations' inputs and outputs. In order to allow the types of inputs and outputs to change, IO refinement replaces these identities with arbitrary relations IT and OT between the input and output elements respectively. IT and OT act as retrieve relations between the inputs and outputs, hence allowing these to change under a refinement in a similar way to changing the state space.

IT and OT are written as schemas and called *input and output transformers*. An input transformer for a schema is an operation whose outputs exactly match the schema's inputs, and whose signature is made up of input and output components only; similarly for output transformers. These are applied to the abstract and concrete operations using piping ($\gg$). To do so we use (in Definition 3) an overlining operator, which extends componentwise to signatures and schemas: $\overline{x?} = x!, \overline{x!} = x?$. Thus $\overline{IT}$ denotes the schema where all inputs become outputs with the same basename, and all outputs inputs.

IO refinement allows inputs and outputs to be refined in a controlled manner. Controlled because since inputs and outputs are observable we must be able to reconstruct the original behaviour from a concrete refinement. This reconstruction is achieved by using the input and output transformers which act as wrappers to a concrete operation, converting abstract inputs to concrete ones and similarly for the output.

To use IO transformers in a non-atomic setting we use mappings from an abstract input to a sequence of concrete inputs representing the inputs needed in the decomposition.

The following definition expresses the refinement of AOp into a fixed sequence $COp_1 \fatsemi COp_2$. In the definition we assume, without loss of generality, that COp_1 and COp_2 are distinct operations with distinct parameter names in order to simplify the presentation.

Definition 3. *Non-atomic downward simulation with IO transformations*
A specification C is a non-atomic IO downward simulation of the specification A if there is a retrieve relation R such that every abstract operation AOp is recast into a sequence of concrete operations $COp_1 \fatsemi COp_2$, and for every $COp_1 \fatsemi COp_2$ there is an input transformer IT which is total on the abstract inputs, for every AOp there is a total injective output transformer OT, and, in addition to initialisation, the following hold.

$$\begin{array}{l}
\forall\, AState;\ CState;\ CState' \bullet \\
\qquad R\ \wedge (IT \gg COp_1 \fatsemi COp_2) \Rightarrow \exists\, AState' \bullet R' \wedge (AOp \gg OT) \\
\forall\, AState;\ CState \bullet R \Rightarrow (pre(\overline{IT} \gg AOp) \iff pre\, COp_1) \\
\forall\, AState;\ CState \bullet R \wedge COp_1 \Rightarrow (pre\, COp_2)'
\end{array}$$

As an example, consider adding some input to our translate operation. Starting with the same $AState$ and $AInit$ (i.e., as in Figure 1) we specify $Translate$ as follows.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Translate}} \\ \hline
\Delta AState \\
val? : \mathbb{Z} \times \mathbb{Z} \\ \hline
x'_A = x_A + \mathit{first}\ val? \wedge y'_A = y_A + \mathit{second}\ val? \\ \hline
\end{array}$$

Using concrete state space $CState_0$ (c.f., Figure 2) we now wish to decompose $Translate$ into the following:

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ \mathit{TransX}\ }}\\
\Delta \mathit{CState}_0\\
x_C? : \mathbb{Z}\\
\hline
x_C' = x_C + x_C?\\
y_C' = y_C\\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ \mathit{TransY}\ }}\\
\Delta \mathit{CState}_0\\
y_C? : \mathbb{Z}\\
\hline
x_C' = x_C\\
y_C' = y_C + y_C?\\
\hline
\end{array}$$

Now, for $\mathit{TransX} \mathbin{;} \mathit{TransY}$ to be a non-atomic refinement of *Translate*, we need a suitable mapping between the input of the abstract operation and the inputs of the concrete operations. Thus we use an input transformer IT which takes the input of *Translate* and transforms it into inputs for the concrete sequence $\mathit{TransX} \mathbin{;} \mathit{TransY}$. IT will be the following.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\ IT\ }}\\
\mathit{val}? : \mathbb{Z} \times \mathbb{Z}\\
x_C!, y_C! : \mathbb{Z}\\
\hline
\mathit{first}\ \mathit{val}? = x_C! \land \mathit{second}\ \mathit{val}? = y_C!\\
\hline
\end{array}$$

Due to absence of outputs no output transformer is needed here, it is the identity and the retrieve relation R is the obvious. Definition 3 can then be applied, checking, for example,

$$\begin{array}{l}
\forall \mathit{AState};\ \mathit{CState}_0;\ \mathit{CState}_0' \bullet\\
\qquad R\ \land (IT \gg \mathit{TransX} \mathbin{;} \mathit{TransY}) \Rightarrow \exists \mathit{AState}' \bullet R' \land \mathit{Translate}
\end{array}$$

which upon expansion is seen to be true. The purpose of the input transformer is to take in the input $\mathit{val}?$ for *Translate* and turn them into outputs $x_C!, y_C!$ to be used as inputs for *TransX* and *TransY*.

3 Coupled Simulations

In this section we explain how coupled simulations can be used to overcome the problems identified above. Remember that the intuition we want to capture is the following.

1. AOp is simulated by $COp_1 \mathbin{;} COp_2$.
2. After COp_1 it is always possible to do COp_2 (*completion* of refinement).
3. It is not possible to do COp_2 unless the beginning of the non-atomic operation has already started, i.e., COp_1 has already happened at some previous point (no starts "in the middle").
4. At the concrete level refinements of two (or more) abstract operations may only be interleaved if the interleaving matches a sequence of abstract operations.

We first deal with the situation where IO transformations are not needed. The generalisation to IO considerations is surprisingly easy, and we discuss this in Section 4 below.

For the sake of simplicity of presentation we assume a single abstract operation AOp is decomposed into a concrete sequence $COp_1 \fatsemi COp_2$ and that the non-atomic operations are used uniquely (i.e., not in two separate decompositions). We also assume IO names are distinct across operations in a decomposition. For example, it is not allowed to split *Translate* into *TransX* and *TransY* and use the same name for the input in *TransX* and *TransY* (e.g. replace both $x_C?$ and $y_C?$ by $z?$). Furthermore, we do not allow *autoconcurrency*: once a refinement of an abstract operation has been started, we may not start another refinement unless the first is completed. All of these restrictions can, however, be omitted (at the price of a large amount of technical overhead).

For convenience of presentation we furthermore use the notation $s \in S$ (for a sequence S) to stand for $s \in \operatorname{ran} S$ and $S \setminus s$ to stand for the sequence S with the first occurrence of s removed.

Intuition (1) from above is covered by Definition 2, thus the purpose of the remainder of the definition we derive is to add the restrictions 2-4. It is clear from these requirements that one must somehow record whether the first part of a non-atomic sequence has commenced. This the coupled simulation does by being indexed with a sequence of concrete operations - those which have been started but not completed.

For instance, if we have two abstract operations AOp and BOp, being refined into $COp_1 \fatsemi COp_2$ and $DOp_1 \fatsemi DOp_2$, respectively, a simulation relation $R^{\langle DOp_1, COp_1 \rangle}$ records the fact that both refinements have been started (first the one of BOp and afterwards that of AOp) but not yet completed. Upon completion of a sequence the corresponding start operation is removed from the index. The coupling condition then requires that the indexed simulation relation agrees with the standard retrieve relation when the sequence is empty, which is exactly the case when refinements that have been started are completed.

For a sequence of concrete operations S we write the coupled simulations as R^S. Coupledness thus requires that

$$R^{\langle\rangle} = R$$

Coupledness (together with the conditions on R^S) guarantees requirement 4 to hold.

The next condition records which non-atomic sequences have been started but not yet completed:

$$\forall\, AState, CState, CState' \bullet R^S \land COp_1 \Rightarrow \exists\, AState' \bullet \Xi AState \land (R^{S \frown \langle COp_1 \rangle})'$$

This says that whenever an operation COp_1 (which begins a non-atomic decomposition) is executed the resulting concrete state should be related to the same abstract state as before, however, not using R^S but instead $R^{S \frown \langle COp_1 \rangle}$.

The requirement embodied in (2) is then simply

$$\forall\, AState, CState \bullet R^S \wedge COp_1 \in S \Rightarrow \text{pre}\ COp_2$$

The third requirement that one can't do COp_2 unless the beginning of the non-atomic operation has already started, can be checked by inspecting the current index S ($COp_1 \in S$). Thus the third requirement is covered by the following

$$\begin{array}{l} \forall\, AState, CState, CState' \bullet \\ \qquad\qquad R^S \wedge COp_2 \Rightarrow COp_1 \in S \wedge \exists\, AState' \bullet AOp \wedge (R^{S \setminus \langle COp_1 \rangle})' \end{array}$$

This says that to perform COp_2 from a particular point one must have already done a COp_1, that is, $COp_1 \in S$, and that once the system has completed COp_2 it should be able to match this with the abstract AOp and link the states appropriately (i.e., via R^T with T being S with COp_1 removed). Note that COp_1 might not just have occurred, sometimes other concrete operations will be interleaved as we saw in the example above.

Taken together the rules lead to the following definition of non-atomic coupled simulation (without IO transformers).

Definition 4. *Non-atomic coupled downward sim. without IO transformations A specification C is a non-atomic coupled downward simulation of the specification A if there is a retrieve relation R showing that C is a non-atomic downward simulation of A, and there is a family of simulation relations R^S such that the following hold.*

C $R^{\langle\rangle} = R$

S1 $\forall\, AState, CState, CState' \bullet R^S \wedge COp_1 \Rightarrow \exists\, AState' \bullet \Xi AState \wedge (R^{S ^\frown \langle COp_1 \rangle})'$

S2 $\forall\, AState, CState \bullet R^S \wedge COp_1 \in S \Rightarrow \mathit{pre}\ COp_2$

S3 $\forall\, AState, CState, CState' \bullet R^S \wedge COp_2 \Rightarrow COp_1 \in S \wedge \exists\, AState' \bullet AOp \wedge (R^{S \setminus \langle COp_1 \rangle})'$

Note that this in particular requires that for every initial state of C there is an initial state of A related by $R^{\langle\rangle}$.

Next, we take another look at the (problematic) translate and invert example (without inputs). We should now be able to show that the concrete specification $CTranslate_1$ from Figure 3 is not a non-atomic coupled simulation of the abstract specification $ATranslate$ from Figure 1, at least not when we use the same R as before. Here R is:

$$\begin{array}{|l} \hline R \\ AState \\ CState_1 \\ \hline x_A = x_C \\ y_A = y_C \\ \neg b \wedge \neg c \\ \hline \end{array}$$

We set $R^{\langle\rangle}$ to be R thereby fulfilling the coupling condition C. There are now four (nonempty) sequences S which are relevant when checking the remaining conditions: $\langle TransX_1\rangle$, $\langle InvX_1\rangle$, $\langle TransX_1, InvX_1\rangle$ and $\langle InvX_1, TransX_1\rangle$. The corresponding retrieve schemas R^S can, in fact, be calculated from the (normal) retrieve relation R. For example, we find we need the following.

$$\begin{array}{|l}
\multicolumn{1}{l}{R^{\langle TransX_1\rangle}} \\
AState \\
CState_1 \\
\hline
x_A = x_C - 3 \\
y_A = y_C \\
b \land \neg c \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{R^{\langle InvX_1\rangle}} \\
AState \\
CState_1 \\
\hline
x_A = -x_C \\
y_A = y_C \\
\neg b \land c \\
\hline
\end{array}$$

To see this, note that whenever $TransX_1$ or $InvX_1$ are executed in a concrete state which is related to an abstract state via $R = R^{\langle\rangle}$, the resulting concrete state should be related to the *same* abstract state via $R^{\langle TransX_1\rangle}$ or $R^{\langle InvX_1\rangle}$, respectively. This can only be achieved when we require x_A to be equal to $x_C - 3$ or $-x_C$.

In a similar fashion we can derive $R^{\langle TransX_1, InvX_1\rangle}$ and $R^{\langle InvX_1, TransX_1\rangle}$. After execution of $TransX_1$, $InvX_1$ is enabled and after its execution the resulting concrete state still has to be related to the same abstract state. This is achieved by now equating x_A with $-x_C - 3$. In a similar way we can derive the retrieve relation needed after $InvX_1$ followed by $TransX_1$.

$$\begin{array}{|l}
\multicolumn{1}{l}{R^{\langle TransX_1, InvX_1\rangle}} \\
AState \\
CState_1 \\
\hline
x_A = -x_C - 3 \\
y_A = y_C \\
b \land c \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{R^{\langle InvX_1, TransX_1\rangle}} \\
AState \\
CState_1 \\
\hline
x_A = -x_C + 3 \\
y_A = y_C \\
b \land c \\
\hline
\end{array}$$

This seems to be fine so far. Conditions C and S1 hold for all sequences S considered above. Condition S2 is fulfilled as well: when $TransX_1$ is in the index S then b is always true and thus $TransY_1$ is enabled; analogously for $InvX_1$ and $InvY_1$. However, a problem arises when we finish sequences that have already been started. As we have seen before there are some interleavings which do not have a matching abstract counterpart and for these interleavings condition S3 fails. Figure 4 shows such an interleaving.

Starting with a pair of $R = R^{\langle\rangle}$-related states the concrete specification executes $TransX_1$ and then $InvX_1$. The two states reached via this execution still have to be related to the same abstract state. This is required by condition S1 and is indeed fulfilled. However, the completion of the now started refinement of *Inverse* causes problems. States 1 and 2 are related via $R^{\langle TransX_1, InvX_1\rangle}$, thus $x_A = -x_C - 3$. In the abstract state 3 coordinate x is changed: $x'_A = -x_A$,

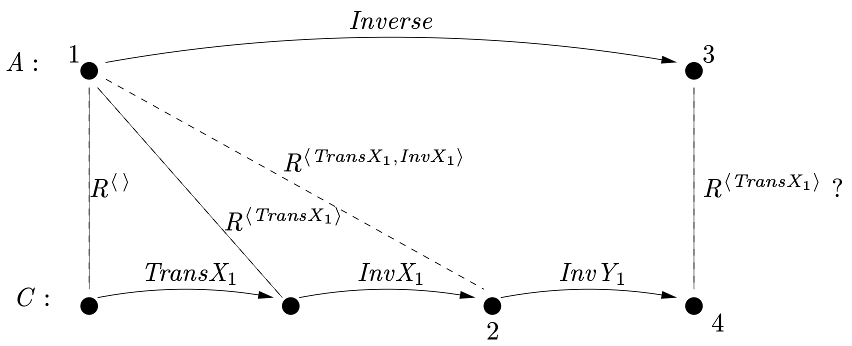

Fig. 4. Matching an interleaved concrete decomposition with its abstract counterpart

whereas in concrete state 4 x remains the same as in 2: $x'_C = x_C$. Thus we get: $x'_A = x'_C + 3$ which is in contrast with relation $(R^{\langle TransX_1\rangle})'$ requiring $x'_A = x'_C - 3$. Thus condition S3 is not met.

This completes the proof that $CTranslate_1$ is not a non-atomic refinement of $ATranslate$ (similar problems arise with other choices of retrieve relations).

The translate example can be adapted into a coupled non-atomic refinement that can be verified. Consider an abstract translate with a count operation (assuming the necessary variables have been declared).

Translate

$\Delta AState$
$x_A?, y_A? : \mathbb{Z}$

$x'_A = x_A + x_A? \wedge y'_A = y_A + y_A?$
$a \wedge \neg a'$

CntTrans

$\Delta AState$

$cnt' = cnt + 1$
$\neg a \wedge a'$

Assume *Translate* is not decomposed in the refinement, but that *CntTrans* is decomposed into

CntTransStart

$\Delta CState$

$cnt' = cnt + 0.5$
$\neg a \wedge \neg d \wedge d'$

CntTransEnd

$\Delta CState$

$cnt' = cnt + 0.5$
$d \wedge \neg d' \wedge a'$

Now, in this decomposition the variables modified as a direct result of the decomposition are independent from those in *Translate*. Hence any amount of interleaving of this with *Translate* (or indeed any non-atomic refinement of it) is a valid coupled simulation (using the obvious coupled retrieve relations).

These two examples illustrate a general point: non-atomic coupled simulations allow either a fully sequential model or a fully concurrent model, but nothing in-between. That is, if some interleavings are allowed then all will be, and conversely, if some interleavings are not allowed, then none will be.

4 Coupled simulations with IO transformation

As a last step we add input and output transformers to our definition. This requires only a small change in the definition. Basically there are two differences: the first concerns the relations R^S which now have to record the inputs and outputs of the operations in S. The second difference is in the formulation of condition S3: the input and output transformers have to be applied when completing a sequence and matching it with the abstract operation AOp.

Definition 5. *Non-atomic coupled downward simulation with IO transformation A specification C is a non-atomic coupled downward simulation of the specification A if there is a retrieve relation R and there are input and output transformers IT and OT showing that C is a non-atomic downward simulation of A, and there is a family of simulation relations R^S such that the following hold.*

C $R^{\langle\rangle} = R$

S1 $\forall AState, CState, CState' \bullet R^S \wedge COp_1 \Rightarrow \exists AState' \bullet \Xi AState \wedge (R^{S \frown \langle COp_1 \rangle})'$
S2 $\forall AState, CState \bullet R^S \wedge COp_1 \in S \Rightarrow pre\ COp_2$
S3' $\forall AState, CState, CState' \bullet IT \gg (R^S \wedge COp_2) \Rightarrow$
$$COp_1 \in S \wedge \exists AState' \bullet (AOp \gg OT) \wedge (R^{S \setminus \langle COp_1 \rangle})'$$

To see this definition in action, consider the translate example with inputs as described in Section 2.1. As before R^S records the effects of part of the concrete operation. With inputs in the concrete operation this necessitates the input being part of the coupled simulation. Thus, for example, $R^{\langle TransX \rangle}$ will be the following

$$\begin{array}{|l}
\multicolumn{1}{l}{R^{\langle TransX \rangle}} \\
\hline
AState \\
CState_0 \\
x_C? : \mathbb{Z} \\
\hline
x_A = x_C - x_C? \\
y_A = y_C \\
\hline
\end{array}$$

We use the same input and output transformers as before (see Section 2.1). Then it is clear by comparison with the example without outputs how this generalises the previous definition. It is also easy to check that the conditions required in Definition 5 hold. For example, calculating $IT \gg (R^{\langle TransX \rangle} \wedge TransY)$ we find it simplifies to

$$
\begin{array}{|l}
\hline
AState \\
\Delta CState_0 \\
val? : \mathbb{Z} \times \mathbb{Z} \\
\hline
y'_C = y_C + second\ val? \\
x_A = x_C - first\ val? \\
y_A = y_C \\
x'_C = x_C \\
\hline
\end{array}
$$

which matches with $Translate \wedge R'$.

A slightly more complicated example is provided by the protocol implementation. We need two transformers: an input transformer for $PrepareToSend \mathbin{;} Transmit$ and an output transformer for $Receive$.

$$
\begin{array}{|l}
\multicolumn{1}{l}{ITPrepTrans} \\
\hline
m? : M \\
text! : Text \\
t! : Time \\
\hline
text! = first\ m? \\
t! = second\ m? \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{OTReceive} \\
\hline
m? : M \\
msg! : M \\
\hline
m? = msg! \\
\hline
\end{array}
$$

Using $R^{\langle\rangle} = R$ (as given in Section 2) as a starting point we get the following simulation relations R^S:

$$
\begin{array}{|l}
\multicolumn{1}{l}{R^{\langle PrepareToSend \rangle}} \\
\hline
AState \\
CState \\
text? : Text \\
\hline
\#q = \#link \\
\forall i : 1..\#q \bullet \\
\quad q.i = (link.i.1.1, link.i.1.2) \\
inStore = \langle text? \rangle \\
outStore = \langle\, \rangle \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{R^{\langle PrepareToReceive \rangle}} \\
\hline
AState \\
CState \\
\hline
\#q = \#(outStore \frown link) \\
q.1 = (outStore.1.1, outStore.1.2) \\
\forall i : 2..\#q \bullet \\
\quad q.i = (link.i.1.1, link.i.1.2) \\
inStore = \langle\, \rangle \\
outStore \neq \langle\, \rangle \\
\hline
\end{array}
$$

Note that the input which is used in operation $PrepareToSend$ is now recorded in the simulation relation. When completing the refinement of $Send$ the input transformer will relate it (together with the input of $Transmit$) to the input of $Send$ (condition S3').

The simulation relations for $S = \langle PrepareToSend, PrepareToReceive \rangle$ and $S = \langle PrepareToReceive, PrepareToSend \rangle$ are equal (since the two operations modify different variables of $CState$). This is already a strong indication that all interleavings will have a matching abstract behaviour.

$$
\begin{array}{|l}
\multicolumn{1}{l}{R^{\langle PrepareToSend, PrepareToReceive\rangle}} \\
AState \\
CState \\
text? : Text \\
\hline
\#q = \#(outStore \frown link) \\
q.1 = (outStore.1.1, outStore.1.2) \\
\forall\, i : 2..\#q \bullet q.i = (link.i.1.1, link.i.1.2) \\
inStore = \langle text? \rangle \\
outStore \neq \langle\,\rangle \\
\hline
\end{array}
$$

We check some of the conditions for non-atomic coupled simulation with IO transformation. Condition C is fulfilled by definition. For condition S1 consider the case that COp_1 is $PrepareToSend$. Since the precondition of $PrepareToSend$ is $inStore = \langle\,\rangle$ it can only be executed in states related by $R^{\langle\,\rangle}$ or $R^{\langle PrepareToReceive\rangle}$. Execution of $PrepareToSend$ leaves variable $link$ unchanged and reads in an input which is stored in $inStore$. Thus the reached state is related to the same abstract state via $R^{\langle PrepareToSend\rangle}$ or $R^{\langle PrepareToReceive, PrepareToSend\rangle}$, respectively.

Condition S2 is easily seen to be fulfilled since the preconditions of $Transmit$ and $Deliver$ require $inStore$ and $outStore$, respectively, to be nonempty. Condition S3' involves the application of input and output transformers. For instance, we have to check that

$$
\begin{aligned}
\forall\, AState, CState, CState' \bullet ITPrepTrans \gg (R^{\langle PrepareToSend\rangle} \land Transmit) \Rightarrow \\
\exists\, AState' \bullet Send \land (R^{\langle\,\rangle})'
\end{aligned}
$$

holds. Here, the input transformer takes in the input $m?$ of $Send$ and relates it to the inputs $text?$ (recorded in the simulation relation) and $t?$ of operation $Transmit$. Thus, in effect $R^{\langle PrepareToSend\rangle}$ records both simulation information (i.e., which states need to be linked up) as well as the effect of performing $PrepareToSend$, condition S3' then checks whether this information is consistent with ending the operation (i.e., $Transmit$) and performing an abstract $Send$.

4.1 Calculating coupled simulations

The conditions required in Definition 5 ask for the existence of an appropriate coupled simulation R^S. However, it should be clear that in the worked examples we have looked at so far, we have in effect calculated R^S using the base retrieve relation R. This, in fact, is a general strategy which can be made precise.

As hinted at before the relations R^S record the effect of the concrete components present in S. We can use this information in the calculation. Consider the translate example given at the start of Section 2.1 (i.e., the one with inputs). R, $R^{\langle TransX\rangle}$ and $TransX$ were

$$\begin{array}{|l}
\hline R \\
AState \\
CState_0 \\
\hline
x_A = x_C \\
y_A = y_C \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline R^{\langle TransX \rangle} \\
AState \\
CState_0 \\
x_C? : \mathbb{Z} \\
\hline
x_A = x_C - x_C? \\
y_A = y_C \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline TransX \\
\Delta CState_0 \\
x_C? : \mathbb{Z} \\
\hline
x_C' = x_C + x_C? \\
y_C' = y_C \\
\hline
\end{array}$$

and we notice that $R^{\langle TransX \rangle}$ is $(R[x_C'/x_C, y_C'/y_C] \mathbin{⨾} TransX)[x_C/x_C', y_C/y_C']$. This holds in general. Thus we can define $R^{\langle\rangle}$ to be R and

$$R^{S \frown \langle COp_1 \rangle} \mathrel{\widehat{=}} (R^S[CState'/CState] \mathbin{⨾} COp_1)[CState/CState'] \tag{1}$$

where $[CState'/CState]$ represent the obvious global substitutions.

This definition allows one to compute R^S for all sequences S with started but unfinished refinements. The conditions in Definition 5 then need verifying. Obviously condition C holds automatically, as does condition S1. To see the latter note that in $R^{S \frown \langle COp_1 \rangle}$ the abstract state remains unchanged (compared to R^S) and is related to the concrete state reached after executing COp_1.

Condition S2 then still requires checking (which is easy) as does S3' (in which lies some complexity). To check S3' we define the effect of finishing a concrete non-atomic operations. We let IT and OT be the input and output transformers, and assume In to be a list of inputs of AOp and Out to be a list of outputs of $COp_1 \mathbin{⨾} COp_2$. We then define:

$$\begin{array}{l}
R^S \blacktriangledown \langle COp_1 \rangle \mathrel{\widehat{=}} \\
\quad \Big(((IT \gg (R^S[CState'/CState] \mathbin{⨾} COp_2)[AState'/AState]) \mathbin{⨾} (AOp \gg OT)) \\
\qquad\qquad\qquad [AState/AState', CState/CState'] \Big) \setminus (In, Out)
\end{array}$$

This is the relation between abstract and concrete state which is reached when finishing a concrete sequence together with executing the corresponding abstract operation. The inputs and outputs are hidden since we do not care about their actual values anymore once both concrete and abstract operations have finished. Condition S3' then requires to check that $R^S \blacktriangledown \langle COp_1 \rangle$ is equal to $R^{S \setminus \langle COp_1 \rangle}$.

To understand the definition and what is required to be verified, we calculate $R^{\langle TransX \rangle} \blacktriangledown \langle TransX \rangle$ in the above example. That is we calculate

$$\begin{array}{l}
(IT \gg ((R^{\langle TransX \rangle})[x_C'/x_C, y_C'/y_C] \mathbin{⨾} TransY)[x_A'/x_A, y_A'/y_A] \mathbin{⨾} Translate) \\
\qquad\qquad\qquad [x_A/x_A', y_A/y_A', x_C/x_C', y_C/y_C'] \setminus (val?)
\end{array}$$

which is

$$
\begin{array}{|l}
\hline
R^{\langle TransX \rangle} \blacktriangledown \langle TransX \rangle \\
AState \\
CState_0 \\
\hline
\exists\, val? : \mathbb{Z} \times \mathbb{Z} \bullet \\
\qquad x_C - \mathit{first}\ val? = x_A - \mathit{first}\ val? \\
\qquad y_C - \mathit{second}\ val? = y_A - \mathit{second}\ val? \\
\hline
\end{array}
$$

which is equivalent to R.

In practice the best approach seems to be to use (1) to calculate the expected values of R^S from a given R. S1 will hold by construction, leaving it necessary to check C, S2 and S3'. If one of the conditions fails then there is no coupled simulation with base relation R. However, the concrete system could still be a coupled non-atomic refinement, namely, under a different relation R.

5 Conclusions

In this paper we have adapted the idea of coupled simulations to a state-based setting. This allowed a derivation of a set of conditions which augment the existing definition of non-atomic downward simulations. The purpose of these additional conditions were to strengthen the downward simulation definition in a number of important ways.

Although the new conditions add complexity, we also showed how the additional coupled simulations that were needed could, in fact, be calculated from the base retrieve relation. Of course, how feasible this will be in practice remains to be seen. This has to be left for further work, as does the integration with process algebraic definitions which we discussed in the introduction.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* CUP, 1996.
2. L. Aceto. *Action Refinement in Process Algebras.* CUP, London, 1992.
3. L. Aceto and M. Hennessy. Towards action-refinement in process algebras. *Information and Computation*, 103:204–269, 1993.
4. E. A. Boiten and J. Derrick. IO-refinement in Z. In A. Evans, D. J. Duke, and T. Clark, editors, *3rd BCS-FACS Northern Formal Methods Workshop.* Springer-Verlag, September 1998. http://www.ewic.org.uk/.
5. W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison.* CUP, 1998.
6. J. Derrick and E. Boiten. Non-atomic refinement in Z. In J. Woodcock and J. Wing, editors, *FM'99, World Congress on Formal Methods*, number 1709 in LNCS, pages 1477–1496. Springer, 1999.
7. J. Derrick and E. A. Boiten. *Refinement in Z and Object-Z.* Springer-Verlag, 2001.

8. C. Fischer. CSP-OZ - a combination of CSP and Object-Z. In H. Bowman and J. Derrick, editors, *Second IFIP International conference on Formal Methods for Open Object-based Distributed Systems*, pages 423–438. Chapman & Hall, July 1997.
9. C. Fischer. How to combine Z with a process algebra. In *ZUM'98: The Z Formal Specification Notation*, volume 1493 of *Lecture Notes in Computer Science*, pages 5–23. Springer-Verlag, September 1998.
10. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
11. C. B. Jones. *Systematic Software Development using VDM.* Prentice Hall, 1989.
12. J. Parrow and P. Sjödin. Multiway Synchronisation Verified with Coupled Simulation. In R. Cleaveland, editor, *CONCUR '92, Concurrency Theory*, number 630 in LNCS, pages 518–533. Springer, 1992.
13. A. Rensink. Action Contraction. In C. Palamidessi, editor, *CONCUR 2000 - Concurrency Theory*, number 1877 in LNCS, pages 290–304. Springer, 2000.
14. A. Rensink and R. Gorrieri. Action refinement as an implementation relation. In M. Bidoit and M. Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 772–786, 1997.
15. G. Smith. A semantic integration of Object-Z and CSP for the specification of concurrent systems. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *FME'97: Industrial Application and Strengthened Foundations of Formal Methods*, volume 1313 of *Lecture Notes in Computer Science*, pages 62–81. Springer-Verlag, September 1997.
16. G. Smith and J. Derrick. Specification, refinement and verification of concurrent systems - an integration of Object-Z and CSP. *Formal Methods in Systems Design*, 18:249–284, May 2001.
17. J. M. Spivey. *The Z Notation: A Reference Manual.* International Series in Computer Science. Prentice Hall, 2nd edition, 1992.
18. S. Stepney, D. Cooper, and J. C. P. Woodcock. More powerful data refinement in Z. In J. P. Bowen, A. Fett, and M. G. Hinchey, editors, *ZUM'98: The Z Formal Specification Notation*, volume 1493 of *Lecture Notes in Computer Science*, pages 284–307. Springer-Verlag, September 1998.
19. H. Treharne and S. Schneider. Using a process algebra to control B operations. In K. Araki, A. Galloway, and K. Taguchi, editors, *International Conference on Integrated Formal Methods 1999 (IFM'99)*, pages 437–456, York, July 1999. Springer.
20. R. van Glabbeek and U. Goltz. Equivalence notions for concurrent systems and refinement of actions. In A. Kreczmar and G. Mirkowska, editors, *Mathematical Foundations of Computer Science 1989*, volume 379 of *LNCS*, pages 237–248. Springer, 1989.
21. Walter Vogler. Failure semantics based on interval semiwords is a congruence for refinement. *Distributed Computing*, 4:139–162, 1991.
22. J. C. P. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof.* Prentice Hall, 1996.