# Changing System Interfaces Consistently: a New Refinement Strategy for CSP$\|$B

Steve Schneider and Helen Treharne

Department of Computing, University of Surrey

**Abstract.** This paper introduces action refinement in the context of CSP$\|$B. Our motivation to include this notion of refinement within the CSP$\|$B framework is the desire to increase flexibility in the refinement process. We introduce the ability to change the events of a CSP process and the B machines when refining a system. Notions of refinement based on traces and on traces/divergences are introduced in which abstract events are refined by sequences of concrete events. A complementary notion of refinement between B machines is also introduced, yielding compositionality results for refinement of CSP$\|$B controlled components. The paper also introduces a notion of I/O refinement into our action refinement framework.

## 1 Introduction

This paper introduces an approach to event refinement in the context of CSP$\|$B. Event refinement (or action refinement) is concerned with developing a finer level of granularity in specifications, by expanding atomic events within the description into more detailed structures. One motivation for our attention to this issue within the CSP$\|$B framework [14] is the desire to increase our range of options when refining processes and operations. We have recently found it useful in the setting of an industrial CSP$\|$B case study [13] to change the level of granularity of the description during the refinement process.

The challenge of how best to do this has been an issue within process algebra since at least the late 1980's, and a broad survey of the work can be found in [4, Chapter 16]. However, the integration of data refinement with action refinement has received limited attention to date. An early paper in this area is [7], which takes a state-based (Z) approach to refining atomic operations by sequences of operations. In this paper we aim to provide a framework for this notion of refinement in the context of the CSP$\|$B combined formal method, using the CSP aspect to capture the action refinements in a more natural way. We introduce the ability to change the events of a CSP process and hence the B machines during a refinement of a system. An important feature of the new refinement framework is that it does not compromise the existing CSP$\|$B theory and does not change the notations of CSP or classical B.

The CSP||B approach favours separation between behavioural patterns and state descriptions. However, behavioural patterns and state may need to be changed during a refinement. For example, a communication protocol may receive a message and subsequently perform some computation. At an abstract level it would be appropriate to denote the type of the message as a deferred set but in a refinement the message may be split into several smaller, more detailed, messages of a concrete type. Furthermore, the subsequent computation could also be segmented. The paper explores what it means to split events in a refinement, and whether the inputs and outputs of operations (and their types) can be changed in a refinement, or distributed across several operations.

The main contribution of the paper is a framework for event refinement: a collection of definitions of how such a notion of refinement may be naturally expressed, together with some theorems that establish that these definitions are collectively consistent. These culminate in Theorem 4, where we see the conditions given in the various definitions support a compositionality result: that refinement of components separately ensures refinement of their parallel combination. Conjecture 1 gives the corresponding result for operations with input and output.

## 2 CSP||B overview

A CSP *controlled component* consists of a CSP process $P$ in parallel with a B machine $M$.

**CSP controllers** Controllers will be written in a subset of the CSP process algebraic language [9, 11]. We begin with the following simple controller language:

**Definition 1 (Controller Syntax).**

$$P ::= a \rightarrow P \mid P_1 \mathrel{\Box} P_2 \mid STOP \mid S$$

The event $a$ is drawn from the set of events, and $S$ is a CSP process variable. Events can either be pure CSP events, or correspond to operations in the controlled B machine. Notationally we will use $e$ for simple atomic CSP events not corresponding to operations, whereas $a$ will be used for operation names. $S$ is a process variable. Recursive definitions are then given as $S \mathrel{\widehat{=}} P$. In a controller definition, all process variables used are bound by some recursive definition.

More generally, events can consist of channels communicating values. An event will then have the structure $c.v$, where $c$ is the channel name and $v$ is the value being passed on the channel. In general, channels can carry multiple values. The process $c!v?x \rightarrow P(x)$ denotes a process ready to output $v$ on channel $c$, and to input a value $x$ at the same time. Its subsequent behaviour is described by $P(x)$.

**B machines** The B-method [1] is structured around B-machines, which provide an encapsulation of state (which can be abstract mathematical structures) and operations on that state, in an object-style structure. A machine is introduced with a name, state variables, an invariant (including type information) on those variables, an initialisation, and a collection of operations on the state.

Operations are declared as $out \longleftarrow op(in) \mathrel{\widehat=} \mathit{PRE}\ P\ \mathit{THEN}\ S\ \mathit{END}$, where $P$ is the precondition of the operation, and $S$ is its body. $in$ and $out$ can in general be sequences of formal parameters. $S$ is an abstract assignment describing how the state can be updated. This can include single and concurrent updates, and nondeterministic choice. Initialisation is also given as an abstract assignment. The abstract assignment constructions we use in this paper are assignment: $x := E$; precondition: $\mathit{PRE}\ P\ \mathit{THEN}\ S\ \mathit{END}$ which executes $S$ if $P$ is true, but otherwise its behaviour is undetermined; parallel assignment: $S \parallel T$; and sequential composition $S; T$.

A machine is *consistent* if its invariant $I$ is initially true, and is preserved by all of the machine's operations when called within their preconditions. The B-Method uses weakest precondition semantics to establish that machines are consistent, and we will assume machine consistency for the purposes of this paper (i.e. the results apply only for consistent machines). The notation $[S]I$ denotes the weakest precondition required for statement $S$ to guarantee achieving postcondition $I$. Invoking a preconditioned operation cannot guarantee anything (not even termination) if the precondition is false, thus $[\mathit{PRE}\ P\ \mathit{THEN}\ S\ \mathit{END}]I = P \wedge [S]I$: to guarantee establishing $I$, $P$ must initially be true, and furthermore $S$ must establish $I$.

Refinement may be considered between two machines $M$ and $M'$. A linking invariant $J$ is a predicate on the states of both $M$ and $M'$ that is used to capture the relationship between their states, to identify when an abstract state is matched by a concrete state. The proof obligation $I \wedge J \Rightarrow [T](\neg[S]\neg J)$ is used to establish that the concrete statement $T$ is a refinement of the abstract statement $S$ in such a context. Further explanation can be found in [1, 12].

**Controlled components** A component is a controller definition $P$ and an associated B machine $M$. The operations $a$ in the machine correspond to events of the same name $a$ in the controller. Operations $out_a \longleftarrow a(in_a)$ are matched by complementary channel communications $a?out_a!in_a$ in the controller: input $in_a$ to the machine is provided by (i.e. an output from) the controller; and output $out_a$ is read by (i.e. input to) the controller. The alphabet $\alpha M$ of the machine is given by its set of operations. We require that $\alpha M \subseteq \alpha P$, that every operation also occurs in the controller. However, controllers may also use CSP events not included in the machine, for interacting with other parts of a larger system, or with its environment.

Morgan's CSP semantics for action systems [10] allows traces, failures, and divergences to be defined for B machines in terms of the sequences of operations that they can and cannot engage in. This gives a way of considering B machines as CSP processes, and treating them within the CSP framework. This enables us to give $P \parallel M$ a CSP semantics.

The traces of a machine $M$ are those sequences of operations $tr = \langle a_1, \ldots, a_n \rangle$ which are possible for the machine. In weakest precondition semantics, an *impossible* trace $tr$ is miraculous: it establishes *false*, i.e. $[T;tr]\mathit{false}$ (where $T$ is the initialisation of the machine). Hence the negation characterises the traces of the machine: $\neg[T;tr]\mathit{false}$. Thus $\mathit{traces}(M) = \{tr \mid \neg[T;tr]\mathit{false}\}$.

A sequence of operations $tr$ is a *divergence* if the sequence of operations is not guaranteed to terminate, i.e. $\neg[T;\ tr]\mathit{true}$. Thus $\mathit{divergences}(M) = \{tr \mid \neg[T;tr]\mathit{true}$

These two definitions provide the link between the weakest precondition semantics of the operations, and the CSP semantics of the B machine. This definition means that calling an operation outside its precondition yields a divergence: termination cannot be guaranteed.

## 3 The basic refinement framework without i/o

To develop the basic framework we will begin by considering pure operations and events, without any input or output communication on them. This will enable us to focus on the sequences of events that we wish to consider. Input/Output considerations will be introduced later, in Section 4.

### 3.1 Sequence Notation

We use the following notation in the paper. If $A$ is a set, then $A^*$ is the set of finite sequences of elements of $A$, and $A^+$ denotes the non-empty finite sequences of elements of $A$. The empty sequence is denoted $\langle\rangle$, and the concatenation of sequences $s$ and $t$ is denoted $s \frown t$. We write $s \leqslant t$ to denote that $s$ is a prefix of $t$. If $A$ is a set, then $s \upharpoonright A$ is the maximal subsequence of $s$ all of whose elements are in $A$: projection of $s$ to $A$. We also define the downwards and upwards closure on a set of sequences $S$ respectively as follows:

$$\downarrow S \mathrel{\widehat=} \{tr \mid \exists\, tr' \in S.tr \leqslant tr'\} \qquad\qquad \uparrow_A S \mathrel{\widehat=} \{tr \in A^* \mid \exists\, tr' \in S.tr' \leqslant tr\}$$

If the set $A$ is implicit from the context then we may write $\uparrow S$.

### 3.2 Implementation mappings

We can now give a definition of consistent refinement between two consistent components $P \parallel M$ and $P' \parallel M'$. The key underlying idea is that whenever an event in an abstract controller $P$ is substituted by a sequence of concrete events in a concrete controller's execution $P'$, and the new concrete events correspond to B operations in a machine $M'$, then we can guarantee that the concrete controlled component is a consistent refinement of the abstract one. We shall see that care will need to be taken when we re-use operations from $M$ in the concrete component.

To do this, we must first introduce an implementation mapping *imp* as follows, which will need to be instantiated for each proposed component refinement.

**Definition 2 (implementation mapping for events).**
*An* implementation mapping *is a function* $imp \in A \rightarrow C^+$*, from abstract events to a sequence of concrete events.*

Here $A = \alpha P$ and $C = \alpha P'$. Note that $A$ and $C$ do not have to be disjoint, therefore we must take into account what happens when events do not change in a refinement. We require a healthiness condition $\mathbf{IMP}_1$ on *imp* as follows:

$$\forall\, b \in A \cap C \;.\; imp(b) = \langle b \rangle$$

We also require that $M$ and $M'$ have the same intersection with $A \cap C$, and that their definitions for those operations in the intersection are identical. Since we are aiming for refinement, this requirement states that these elements do not change in the refinement step.

Observe that implementation mappings are different to CSP *alphabet renamings*, which map events to single events rather than to sequences.

We now define a mapping from sequences of abstract events to sequences of concrete traces.

**Definition 3 (implementation mapping).**
*Given an implementation mapping imp, the function* $\phi_{imp} : A^* \rightarrow C^*$ *is defined as follows:*

$$\phi_{imp}(\langle\rangle) = \langle\rangle \qquad\qquad \phi_{imp}(\langle a \rangle \frown tr) = imp(a) \frown \phi(tr)$$

*If the mapping imp is clear from the context, then it may be elided and we write* $\phi(tr)$*. Note that in functional terms,* $\phi_{imp} = flatten \circ (map\ imp)$*.*

When we come to consider divergences we will need one further construction: the set of non-empty prefixes of concrete traces:

**Definition 4.** *Given an implementation mapping* $imp : A \rightarrow C^+$ *and* $a \in A$*, we define* $imp^+(a) \mathrel{\widehat{=}} \{tr' \mid tr' \leq imp(a) \wedge tr' \neq \langle\rangle\}$*.*

This is used in the following definition: the mapping $\psi$ identifies all subsequences related to an abstract sequence of events including the complete subsequences. This definition will be used in Theorem 4 to track down the point at which a concrete trace diverges.

**Definition 5 (subsequence implementation mapping).**
*The function $\psi : A^* \rightarrow C^*$ is defined as follows:*

$$\psi(\langle\rangle) = \langle\rangle \qquad\qquad \psi(tr \frown \langle a\rangle) = \{\phi(tr)\} \frown imp^+(a)$$

### 3.3 Refinement

Having identified correspondences between abstract and concrete traces, through the function $\phi_{imp}$, we are now in a position to define a corresponding notion of refinement:

**Definition 6 (trace refinement relative to *imp*).**
*$P \sqsubseteq^T_{imp} P'$ iff $traces(P') \subseteq \downarrow \{\phi_{imp}(tr) \mid tr \in traces(P)\}$.*

Note that refinement with respect to an implementation mapping is not preserved by parallel composition, as the following example illustrates:

*Example 1.* Consider $imp(a) = imp(b) = \langle c\rangle$, and

$$P = a \rightarrow STOP \qquad Q = b \rightarrow STOP \qquad P' = Q' = c \rightarrow STOP$$

Observe that $P \sqsubseteq^T_{imp} P'$ and $Q \sqsubseteq^T_{imp} Q'$ but $P \parallel Q = STOP, P' \parallel Q' = c \rightarrow STOP$, and so the refinement relation does not hold between $P \parallel Q$ and $P' \parallel Q'$.

We now obtain the following result which allows refinement of a controlled component to be deduced from the appropriate refinement relation between controllers.

**Theorem 1.** *If $P \sqsubseteq^T_{imp} P'$ then $P \parallel M \sqsubseteq^T_{imp} P' \parallel M'$*

*Proof.* $traces(P \parallel M) = traces(P)$ and $traces(P' \parallel M') = traces(P')$ in this case (since there is no i/o).

Observe that the machines $M$ and $M'$ can be independent: the result follows purely from the relationship between $P$ and $P'$.

The mapping *imp* can also be used to transform a CSP process description to another CSP process which is a refinement.

**Definition 7 (mapping abstract to concrete processes).**
*If $imp : A \rightarrow C^+$ is an implementation mapping, then we define the mapping $\Theta_{imp}$ on CSP process descriptions as follows:*

$$\begin{aligned}
\Theta_{imp}(STOP) &= STOP \\
\Theta_{imp}(a \rightarrow P) &= Pref(imp(a), \Theta_{imp}(P)) \\
\Theta_{imp}(P_1 \Box P_2) &= \Theta_{imp}(P_1) \Box \Theta_{imp}(P_2) \\
\Theta_{imp}(S) &= S
\end{aligned}$$

$$\begin{aligned}
\textit{where} \quad Pref(\langle\rangle, Q) &= Q \\
Pref(((\langle b\rangle \frown tr, Q) &= b \rightarrow Pref(tr, Q)
\end{aligned}$$

The mapping has been constructed to yield the following theorem, proven by structural induction on $P$: that the result of the transformation is a refinement of the original process.

**Theorem 2.** $\forall\, imp, P \;.\; P \sqsubseteq^{T}_{imp} \Theta_{imp}(P)$

### 3.4 Refining B machines

Now we consider what it means to refine a B machine in the context of an implementation mapping *imp*. This will enable the introduction of new operations during the refinement process.

**Definition 8.** *For a machine $M$ and a sequence of events $tr$, we define $op_M(tr) = tr \upharpoonright \alpha M$. In other words, $op_M(tr)$ is the sequence of operations in $tr$ that the machine $M$ participates in. If the machine $M$ is clear from the context then we may write $op(tr)$.*

**Definition 9 (refinement of B machines).**
*If $M$ and $M'$ have linking invariant $J$, then*

$$M \sqsubseteq^{B}_{imp} M' \text{ iff } \forall\, a \in \alpha(M) \;.\; a \sqsubseteq imp(a) \;\; i.e., I \wedge J \wedge P \Rightarrow [op(imp(a))]\neg[a]\neg J$$

This states that any *imp* trace refinement is respected in the B machine: any sequence of operations corresponding to $a$ matches the operation $a$. It is complementary to the trace notion of refinement $\sqsubseteq^{T}_{imp}$, which requires that only those concrete sequences of operations that correspond to abstract ones should be possible.

### 3.5 Traces/Divergences

Now we wish to generalise the notion of refinement so that it works for refinement in the traces/divergences model.

**Definition 10 (Traces/divergences refinement with respect to imp).**
*If imp is an implementation mapping, then*

$$\begin{aligned} P \sqsubseteq^{TD}_{imp} P' \text{ iff } & P \sqsubseteq^{T}_{imp} P' && (1)\\ & \wedge\; divergences(P') \subseteq \uparrow_{\alpha P'} \Big( \bigcup_{tr \in divergences(P)} \psi(tr) \Big) && (2) \end{aligned}$$

This states that any divergence of $P'$ must correspond to a divergence of $P$: given a divergent trace $tr$ of $P$, $\psi(tr)$ gives the corresponding divergences of $P'$. Thus if event $a$ introduces divergence, then divergence can be introduced anywhere along $imp(a)$ from the first event onwards. These are exactly the sequences in $\psi(tr)$.

*Example 2.* Consider $imp(a) = \langle c, d\rangle$ and $imp(b) = \langle e, f\rangle$. Consider $P$ and $P'$ as follows, where $P$ diverges after $\langle a, b\rangle$, and $P'$ diverges after $\langle c, d, e\rangle$:

$$\begin{aligned} traces(P) &= \{\langle\rangle, \langle a\rangle, \langle a, b\rangle\} \cup \{\langle a, b\rangle \frown s \mid s \in \{a, b\}^*\} \\ divergences(P) &= \{\langle a, b\rangle \frown s \mid s \in \{a, b\}^*\} \end{aligned}$$

$$\begin{aligned} traces(P') &= \{\langle\rangle, \langle c\rangle, \langle c, d\rangle, \langle c, d, e\rangle\} \cup \{\langle c, d, e\rangle \frown s \mid s \in \{c, d, e, f\}^*\} \\ divergences(P') &= \{\langle c, d, e\rangle \frown s \mid s \in \{c, d, e, f\}^*\} \end{aligned}$$

Observe that $\psi(\langle a, b\rangle) = \{\langle c, d, e\rangle, \langle c, d, e, f\rangle\}$ and so the condition in Line 2 is satisfied, and $P \sqsubseteq^{TD}_{imp} P'$.

We obtain the same result, again proved by structural induction over $P$, for trace divergence refinement as we did in Theorem 2 for trace refinement.

**Theorem 3.** $\forall\, imp, P \;.\; P \sqsubseteq^{TD}_{imp} \Theta_{imp}(P)$

The previous definitions have laid the groundwork for the following result, which is the key compositionality property we have been working towards:

**Theorem 4 (Trace divergence refinement in controlled components).** *If* $P \sqsubseteq^{TD}_{imp} P'$ *and* $M \sqsubseteq^{B}_{imp} M'$ *then* $P \parallel M \sqsubseteq^{TD}_{imp} P' \parallel M'$.

*Proof.* We know

$$traces(P') \subseteq \downarrow\Big(\bigcup_{tr \in traces(P)} \phi(tr)\Big) \tag{3}$$

$$divergences(P') = \emptyset = divergences(P) \tag{4}$$

$$traces(M') = (\alpha M')^* \quad where \quad \alpha M' \subseteq \alpha P' \tag{5}$$

$$traces(M) = (\alpha M)^* \quad where \quad \alpha M \subseteq \alpha P \tag{6}$$

Then consider $tr \in divergences(P' \parallel M')$. Then let $tr_0$ be the minimal divergent prefix of $tr$.

Then $tr_0 \in traces(P')$ and $tr_0 \upharpoonright \alpha M' \in divergences(M')$

$\exists\, tr'' \in traces(P).tr_0 \leq \phi(tr'')$ from (3)

Also $tr_0 \upharpoonright \alpha M' \in divergences(M')$ so $\phi(tr'') \upharpoonright \alpha M' \in divergences(M')$. Therefore $tr'' \upharpoonright \alpha M \in divergences(M)$ from Lemma 2 below.

Therefore $tr'' \in divergences(M) \cap traces(P)$ so $tr'' \in divergences(P \parallel M)$.

Lemmas 1 and 2 below are used in the proof of Theorem 4 above.

**Lemma 1.** *If* $M \sqsubseteq^B_{imp} M'$ *then* $I \wedge J \wedge [op(tr)]true \Rightarrow [op(\phi_{imp}(tr))]true$

**Lemma 2.** *If* $M \sqsubseteq^B_{imp} M'$ *and* $(\phi_{imp}(tr)) \upharpoonright \alpha M'$ *is a divergence of* $M'$ *then* $tr \upharpoonright \alpha M$ *is a divergence of* $M$*.*

Theorem 4, unlike Theorem 1, requires the refinement relationship between machines. When only traces are considered, internal states of the machines do not affect the semantics of the parallel combination, so refinement relies purely on the CSP controllers. However, when divergences are also considered, then divergent behaviour (corresponding to an operation being called outside its precondition) is reflected in the semantics. Hence refinement of a controlled component requires that the states of the machines match up, so the concrete machine can diverge only where the abstract machine description allows it.

*Example 3.* Consider $M \sqsubseteq^B_{imp} M'$ where

- $imp(a) = \langle b, c \rangle$; $imp(w) = \langle v \rangle$; where $a$, $b$, and $c$ are machine operations and $w$ and $v$ are not;
- Machine $M$ has operation $a \mathrel{\widehat{=}} BEGIN\ nn := nn + 4\ END$;
- Machine $M'$ has $b \mathrel{\widehat{=}} PRE\ \ even(mm)\ \ THEN\ \ mm := mm + 1\ \ END$ and $c \mathrel{\widehat{=}} PRE\ \neg even(mm)\ THEN\ mm := mm + 3\ END$.

The example shows that an event can be refined to a sequence of events. $M'$ does contain divergences (e.g. $\langle b, b \rangle$ or $\langle b, c, c \rangle$), but the refinement of $M$ and $M'$ is in the context of *imp* so only sequences which are the image of some abstract sequence need to be considered. Therefore, we need only show that refining $a$ by the sequence of operations $(b;c)$ is an appropriate B refinement, achieved in practice by discharging the proof obligation identified in Definition 9. An appropriate $J$ would be $nn = mm$. We could equally have reused $nn$ in $M'$. Divergent sequences of operations such as $(b;b)$ and $(b;c;c)$ are ruled out since they cannot arise from an application of *imp* to an abstract trace.

Consider an abstract trace $tr = \langle a, w, a \rangle$. Then $\phi(tr) = \langle b, c, v, b, c \rangle$. If $tr \upharpoonright \alpha M = \langle a, a \rangle$ is not a divergence of $M$, then $\phi(tr) \upharpoonright \alpha M' = \langle b, c, b, c \rangle$ is not a divergence of $M'$ by the contrapositive of Lemma 2.

Define $P = a \rightarrow w \rightarrow P$ and $\Theta_{imp}(P) = P' = b \rightarrow c \rightarrow v \rightarrow P'$. We have $P \sqsubseteq^{TD}_{imp} P'$ from Theorem 3. Theorem 4 then yields that $P \parallel M \sqsubseteq^{TD}_{imp} P' \sqsubseteq M'$.

## 4 The refinement framework with i/o

We begin by focusing on the B framework. Our form of interface refinement in the context of operation input and output means that the input and output values across the operations need to be related.

### 4.1 Refining B operations

For a given event $a$ with $imp(a) = cs = \langle c_1, \ldots, c_n \rangle$, let $in_a$ be the sequence of input variables to $a$, and $out_a$ be the sequence of output variables for $a$, i.e. the declaration of $a$ is $out_a \longleftarrow a(in_a)$. Let $in_{cs}$ be the sequence of input variables to the collection of the $c$ operations for $c \in cs$, and $out_{cs}$ be the sequence of output variables for the $c$ operations. In other words, if the $c_i$ operations' declarations are $out_{c_i} \longleftarrow c_i(in_{c_i})$, then $out_{cs} = out_{c_1} \frown \ldots \frown out_{c_n}$, and $in_{c_i} = in_{c_1} \frown \ldots \frown in_{c_n}$. We assume that all operations have disjoint input and output variable names.

An interface refinement for $a$ will relate the abstract and concrete input variables, and similarly with the output variables. The relationships can be formalised with a relation $r_{in,a}$ relating the abstract and concrete input variables, and a relationship $r_{out,a}$ relating the abstract and concrete output variables. These relations may be thought of as linking invariants for the inputs and for the outputs. We will use $r$ to abbreviate the collection of all the $r_{in,a}$ and $r_{out,a}$.

We generalise Definition 9. The refinement relation is with respect both to the mapping $imp$ and the collection of relations $r$:

**Definition 11 (Refinement of operations within B machines).** *If $M$ and $M'$ have linking invariant $J$ then*

$$\begin{array}{l} M \sqsubseteq^B_{imp,r} M' \textit{iff} \\ \forall\, a \in \alpha(M)\ ,\ r_{in,a} \wedge I \wedge J \wedge P_a \Rightarrow [op(imp(a))]\neg[a]\neg(J \wedge r_{out,a}) \end{array}$$

### 4.2 Examples illustrating aspects of Definition 11

*Example 4 (Implementation modulo* 5*).* The example in Figure 1 considers a change in data representation, resulting in a loss of information but in a way that allows refinement. Our single operation multiplies an input by 3 and returns the result. If we wish to refine this so that all values are modulo 5, then the refined operation may be used. This only inputs and outputs values modulo 5. The relations on inputs and on outputs capture this relationship: input of an abstract value is implemented by the input of that value modulo 5, and the resulting output will be the abstract output, modulo 5. The resulting proof obligation can be discharged to establish the refinement relationship.

$Times3R$ is a refinement of $Times$ with $imp(triple) = \langle tripleR \rangle$ and the following definitions, which together satisfy the proof obligation of Definition 11:

$$J = true \qquad r_{in,triple} : ww = xx \bmod 5 \qquad r_{out,triple} : zz = yy \bmod 5$$

*Example 5 (Change of offset).* In the example of Figure 2, we change the offset of the readings, so that concrete inputs are the abstract inputs offset by +1.

```
MACHINE     Times3                   MACHINE     Times3R
OPERATIONS                           OPERATIONS
  yy <-- triple(xx) =                  zz <-- tripleR(ww) =
      PRE xx : NAT                         PRE ww : 0..4
      THEN yy := 3 * xx                    THEN zz := (ww * 3) mod 5
      END                                  END
END                                  END
```

**Fig. 1.** Tripling, modulo 5

```
MACHINE         Increase          MACHINE         IncreaseR
VARIABLES       total             VARIABLES       totalR
INVARIANT       total : NAT       INVARIANT       totalR : NAT
INITIALISATION  total := 0                           & totalR = total
OPERATIONS                        INITIALISATION  totalR := 0
  add(xx) =                       OPERATIONS
      PRE xx : NAT                  addR(ww) =
      THEN total := total + xx          PRE ww : NAT
      END                               THEN totalR := totalR + (ww - 1)
END                                     END
                                  END
```

**Fig. 2.** Change of offset I

An abstract input value $xx$ is implemented by the concrete value $xx + 1$. This is captured in the relation $r_{in,add}$.

*IncreaseR* is a refinement of *Increase*, under the following definitions:

$$imp(add) = \langle addR \rangle \qquad J : totalR = total \qquad r_{in,add} : xx = ww - 1$$

The proof obligation of Definition 11 is met by these definitions. The steps are as follows:

$$\begin{aligned}
&[addR]\neg[total := total + xx]\neg(J \wedge r_{out,add}) \\
&= [addR](total + xx = totalRR) \\
&= ww \in NAT \wedge (total + xx = totalR + ww - 1) \\
&\Leftarrow xx \in NAT \wedge xx = ww - 1 \wedge total = totalR \qquad (7) \\
&= P_{add} \wedge r_{in,add} \wedge J
\end{aligned}$$

*Example 6 (Change of offset).* The example in Figure 3 is similar to the previous example, except that the concrete inputs are the abstract inputs offset by $-1$.

The change of offset is captured in the relation $r_{in,add}$. One might hope that the following definitions would show that *IncreaseR* is a refinement of *Increase*:

$$imp(add) = \langle addR \rangle \qquad J : totalR = total \qquad r_{in,add} : xx = ww + 1$$

However, the proof obligation of Definition 11 is not met by these definitions, and in particular the implication in Line 7 does not carry through, since an abstract

```
MACHINE         Increase          MACHINE         IncreaseR
VARIABLES       total             VARIABLES       totalR
INVARIANT       total : NAT       INVARIANT       totalR : NAT
INITIALISATION  total := 0                          & totalR = total
OPERATIONS                        INITIALISATION  totalR := 0
  add(xx) =                       OPERATIONS
      PRE xx : NAT                  addR(ww) =
      THEN total := total + xx          PRE ww : NAT
      END                               THEN totalR := totalR + (ww + 1)
END                                     END
                                  END
```

**Fig. 3.** Change of offset II

input $xx = 0$ cannot be matched by any natural number $ww$. Note that if the precondition on the concrete operation allowed $ww$ also to range over negative integers, then the proof obligation would be met: clearly the abstract value 0 is represented by $-1$.

Examples 5 and 6 together illustrate the delicate relationship between what is required by the refinement and what is allowed by the abstract machine. We see that whenever the abstract operation is enabled with a particular input, then the refinement must also be enabled with a related input value. However, we see from Example 5 that the converse is not the case: the concrete input 0 there does not correspond to any abstract input. The abstract machine imposes no requirements on the refinement behaviour for that input value: it corresponds to a value that is outside the abstract precondition.

*Example 7 (distributing inputs).*

In Figure 4, *SensorR* is a refinement of *Sensor* with the following definitions:

$$imp(update) = \langle updatet, updatep \rangle \qquad r_{in,update} : dt = dt1 \wedge dp = dp1$$

Observe that the proof obligation requires only that the abstract and refined machine states match at the end of the sequence of concrete operations. The refinement machine will pass through states that need not match the abstract state.

## 5 Trace refinement with i/o

Given an implementation mapping $imp$ and relations $r_{in,a}$, $r_{out,a}$, we can define a refinement relation on processes that incorporates the input and output values.

Given a particular $r_{in,a}$ (as used in the machine refinement), and where $imp(a) = \langle c_1, \ldots, c_n \rangle$ we will define the sequences of concrete events with their inputs and

```
MACHINE         Sensor            MACHINE       SensorR
VARIABLES       tt, pp            VARIABLES     rrR, ppR
INVARIANT       tt : NAT          INVARIANT     ttR : NAT & ppR : NAT
                 & pp : NAT                      & ttR = tt & ppR = pp
INITIALISATION  tt :: NAT         INITIALISATION  ttR :: NAT
                 || pp :: NAT                      || ppR :: NAT
OPERATIONS                        OPERATIONS
  update(dt,dp) =                   updatet(dt1) =
      PRE dt : NAT & dp : NAT           PRE dt1 : NAT
      THEN tt := tt + dt                THEN ttR := ttR + dt1
            || pp := pp + dp            END;
      END                           updatep(dp1) =
END                                     PRE dp1 : NAT
                                        THEN ppR := ppR + dp1
                                        END
                                  END
```

**Fig. 4.** Distributing inputs

outputs, associated with an abstract i/o event $a.v.w$, where $v$ is the inputs to $a$, and $w$ is the outputs. The mapping *imp* lifts to a mapping *imp*′ which gives the set of all sequences corresponding to a particular i/o event:

**Definition 12.**

$$\begin{aligned}&\mathit{imp}'(a.v.w) \;=\\ &\quad \{\langle c_1.v_1.w_1, \ldots, c_n.v_n.w_n\rangle \mid r_{in,a}(v, v_1, \ldots, v_n) \wedge r_{out,a}(w, w_1, \ldots, w_n)\}\end{aligned}$$

The function $\phi$ then generalises as follows:

$$\phi_{\mathit{imp},r}(\langle\rangle) = \{\langle\rangle\} \qquad\qquad \phi_{\mathit{imp},r}(\langle a.v.w\rangle \frown \mathit{tr}) = \mathit{imp}'(a.v.w) \frown \phi_{\mathit{imp},r}(\mathit{tr})$$

This supports the natural definition of trace refinement: that every trace of $P'$ should arise from some trace of $P$.

**Definition 13 (trace refinement relative to *imp* and $r$).**
$P \sqsubseteq^T_{\mathit{imp},r} P'$ *iff* $\mathit{traces}(P') \subseteq\, \downarrow (\bigcup_{\mathit{tr} \in \mathit{traces}(P)} \phi_{\mathit{imp},r}(\mathit{tr}))$

We have already identified a notion of refinement for processes, and one for machines in terms of relationships between their operations. We are aiming for the following compositionality result, which is an extension of Theorem 4 in the context of the relation $r$ on i/o:

*Conjecture 1.* If $P \sqsubseteq^T_{\mathit{imp},r} P'$ and $M \sqsubseteq^B_{\mathit{imp},r} M'$ then $P \parallel M \sqsubseteq^T_{\mathit{imp},r} P' \parallel M'$.

Note that the traces of machines $M$ are no longer all possible traces (as they are without i/o), since they constrain the possible outputs. Hence the conjecture takes the machine traces into account, since they restrict the overall behaviour.

This notion of refinement is not reflexive: $P \sqsubseteq_{imp,r} P$ does not hold in general, because the inputs and outputs may change (even where *imp* is the identity function). Hence a combination $P \parallel M$ will not be refined simply by refining $M$; the controller will need to be refined as well.

## 6 Discussion and Related Work

In this paper we presented the theoretical framework to support the refinement of an abstract event with a sequence of concrete events within the CSP$\|B$ framework. From this point of view the important result is Theorem 4. We also described what it means to distribute inputs and outputs across the concrete sequence of operations, and showed how the type of the inputs and outputs can also be refined. Natural extensions to the work are consideration of failures information, and refinement of events by processes, allowing nondeterminism. We were not able to consider them in this paper for reasons of space.

In [7], Derrick and Boiten present a theory for non-atomic refinement using Z. They also support the refinement of an abstract operation with a sequence of concrete operations. Our motivation is the same as theirs: the precise structure of an implementation may not be known at the abstract level and we need to provide a way of being able to introduce more detail at the concrete level. We can also split a collection of inputs and/or outputs across a number of operations. The difference with our work is that the sequences of operations we need to consider are defined within a CSP controller and the implementation mapping between abstract and concrete operations is explicitly described.

Derrick and Boiten also consider a notion of I/O refinement in [5, Chapter 10]. They establish conditions for changing the I/O within single operations to provide a refinement, using input and output transformers, which play a similar role to our relations $r_{in,a}$ and $r_{out,a}$. In [8] Derrick and Wehrheim bring together the ideas from [7] and [5] and refine atomic operations by sequences of operations together with I/O refinement. Their approach is entirely state-based, which makes the handling of sequences of operations more difficult, and the authors state in their conclusions that the combination with process algebra remains to be investigated. This paper does combine the state-based view with a process algebra, giving explicit and natural descriptions of control in specifications, and so handling the refining sequences of operations more easily.

In Event-B [2], a refinement of an event, e.g., $a$ can be achieved using several events (at least one), one of which must be the refinement of the original $a$ event. Any new events must be a refinement of *Skip*. Event-B refinement proof obligations ensure that new events do not cause infinite internal behaviour. Furthermore, new events can occur non-deterministically, provided their guards are true, i.e., Event-B does not require an explicit scheduler. We have shown how to refine an event (which may have a corresponding B operation) with a single sequence of events (again with underlying B operations) and thus an explicit schedule must be provided in the refinement. This may be restrictive when there are several scheduling possibilities. However, if the scheduler is known in advance then we provide an explicit way of describing it in a refinement. Also, we do not require that one event is a refinement of the original event. What we require is that a sequence of events is an appropriate refinement of an abstract event. Our refinement also allows I/O refinement and type refinement of the inputs and the outputs; recent research in Event-B is also examining how to include I/O parameters in events [6].

Our approach to traces and trace divergences event refinement bears some resemblance to the approaches to action refinement in process algebras developed in the 1980's and early 1990's, see e.g. [3], where single events are refined by more complex behaviour. However, the focus then was within pure process algebra, and with more intricate semantics. In contrast, our emphasis is on developing an approach which integrates with state-based components, in our case B-machines, and it is this emphasis that has driven the development of the approach presented in this paper.

In our recent work we have continued to generalise the results in order to support refining events to sets of sequences of events and to processes.

**Acknowledgments** Thanks to Thai Son Hoang for discussions related to Event-B. Thanks also to the IFM reviewers for their insightful and helpful comments.

## References

1. J.-R. Abrial. *The B Book: Assigning programs to meanings.* Cambridge University Press, New York, NY, USA, 1996.
2. J-R. Abrial. *Modelling in Event-B: System and Software Engineering.* Cambridge University Press, In preparation.
3. L. Aceto. *Action Refinement in Process Algebras.* Cambridge University Press, 1992.
4. J.A. Bergstra, A. Ponse, and S.A. Smolka, editors. *Handbook of Process Algebra.* North-Holland, 2001.
5. E. Boiten and J. Derrick. *Refinement in Z and Object-Z: Foundations and Advanced Applications.* Springer-Verlag, 2001.
6. M. Butler. personal communication, September, 2008.
7. J. Derrick and E. Boiten. Non-atomic refinement in Z. In *FM '99 Volume II*, pages 1477–1496, London, UK, 1999. Springer-Verlag.
8. J. Derrick and H. Wehrheim. Using coupled simulations in non-atomic refinement. In *ZBB*, 2003.
9. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
10. C. Morgan. Of wp and CSP. *Beauty is our business: a birthday salute to E. W. Dijkstra*, pages 319–326, 1990.
11. S. Schneider. *Concurrent and Real-Time Systems: the CSP Approach.* Wiley, 1999.
12. S. Schneider. *The B-Method: an introduction.* Palgrave, 2001.
13. S. Schneider, D. Pizarro, and H. Treharne. The futuretech demonstrator, 2008. Future Technologies for System Design Technical Report, University of Surrey.
14. S. Schneider and H. Treharne. CSP theorems for communicating B machines. *Formal Asp. Comput.*, 17(4):390–422, 2005.