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 [19] 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 [18] 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 [5, Chapter 16]. However, the integration of data refinement with action refinement has received limited attention to date. An early paper in this area is [11], which takes a state-based (Z) approach to refining atomic operations by sequences of operations. In the paper presented here 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, both 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 this paper is a framework for event refinement with state. We obtain a compositionality result, that refinement of components separately ensures refinement of their parallel combination. The paper is an extension to the initial work reported in [20], where events are implemented by single sequences of events. This paper generalises that approach so that events are implemented by sets of sequences, supporting choice in refinement and laying the foundation for ultimately refining events by processes, as in classical action refinement. This paper also extends [20] by including the proofs of key theorems, as well as additional examples and discussion.

2 CSP‖B overview

A CSP *controlled component* consists of a CSP process P in parallel with a B machine M. This is written as $P \parallel M$.

CSP controllers Controllers will be written in a subset of the CSP process algebraic language [13, 16]. We begin with the following simple controller language for process terms:

Definition 1 (Controller Syntax).

$$P ::= a \rightarrow P \mid P_1 \,\Box\, P_2 \mid STOP \mid X \mid \mu\, X.P$$

The event a is drawn from the set of events. 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 B operations, whereas a will be used for operation names. $P_1 \,\Box\, P_2$ denotes a choice between processes P_1 and P_2. $STOP$ represents termination. X is a process variable, and recursive definitions are given as $\mu\, X.P$. We normally aim for recursive definitions to be guarded, i.e. a recursive call should be preceded by an event. In a controller definition, all process variables must be bound within some recursive definition: a CSP *process* is a term with no free variables.

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 $P(x)$.

Processes P are associated with alphabets, denoted αP, which are understood as their interface — the set of events that they can engage in. The CSP language also contains other constructs, most notably parallel composition. The process $P_1 \parallel P_2$ executes P_1 and P_2 concurrently: they synchronise on events in the intersection of their alphabets, but can perform other events independently.

The CSP approach to semantics is to associate each process with a set of possible observations. In the *traces model* for CSP, observations are *traces*, which are finite sequences of events. Thus $traces(P)$ is the set of possible traces that may be observed of some execution of P. Another kind of observation is a *divergence*, which is a sequence of events during or after which the process might not terminate, for example if it enters an internal loop. No guarantees can be made about the behaviour of a process after divergence. $divergences(P)$ is the set of divergences that process P can exhibit. The *traces/divergences model* for CSP associates each process with a set of traces and a set of divergences. Full descriptions of the semantic models can be found in [15, 16].

The semantic models support notions of *refinement*. In the traces model, P is refined by P' if $traces(P') \subseteq traces(P)$. This is written $P \sqsubseteq_T P'$. This states that any trace of P' must be a trace of P. Thus if all of P's behaviours are appropriate in some context, then all of P''s behaviours must also be. In the traces/divergences model, P is refined by P' means that $traces(P') \subseteq traces(P)$, and $divergences(P') \subseteq divergences(P)$. This is written $P \sqsubseteq_{TD} P'$. We may view these definitions as requiring that any behaviour of P' is allowed by P, in the sense that P can also perform them. This notion of refinement will be generalised in this paper.

B machines The B-method [1] is structured around B-machines, which provide an encapsulation of abstract state and operations on that state, in an object-based style. 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 $out \longleftarrow op(in) \mathrel{\widehat{=}} PRE\ P\ THEN\ S\ END$, 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: *PRE P THEN S 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 $[PRE\ P\ THEN\ S\ END]I = P \wedge [S]I$: 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 them, 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, 17].

Controlled components A component consists of 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 αM 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, a controller 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 [14] allows traces 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. $[T;tr]false$ (where T is the initialisation of the machine). Hence the negation characterises the traces of the machine: $\neg[T;tr]false$. Thus $traces(M) = \{tr \mid \neg[T;tr]false\}$.

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

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 in the CSP sense: 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 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 of events, then $s \upharpoonright A$ is the maximal subsequence of s all of whose elements are in A: the projection of s to A.

These operators also lift to sets. If $S, T \subseteq A^*$ then we define

$$S \frown T \mathrel{\widehat{=}} \{s \frown t \mid s \in S \wedge t \in T\}$$
$$S \leqslant T \mathrel{\widehat{=}} \forall s \in S.\, \exists t \in T.s \leqslant t$$
$$s \leqslant T \mathrel{\widehat{=}} \exists t \in T.s \leqslant t$$

If $S \leqslant T$ then we say that S *is dominated by* T.

We also define the downwards and upwards closure on a set of sequences S respectively as follows:

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

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 sequences 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 needs to be taken when we re-use operations from M in the concrete component.

In this paper we generalise the initial approach of [20] which mapped each abstract events to a single concrete sequence. By generalising here to sets of concrete sequences we allow events to be refined by choices as well as individual sequences.

We introduce the notion of an *implementation mapping* in Definition 2 below. Such a mapping will be needed for each proposed component refinement.

Definition 2 (implementation mapping for events).
An implementation mapping *is a function* $imp \in A \rightarrow \mathbb{P}(C^*)$*, from abstract events to sets of concrete sequences.*

We will assume without loss of generality that A and C are disjoint, Where we wish to reuse an event name at the abstract and concrete level we can consider there to be two copies of the event, one in A and one in C.

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

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

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

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

If the mapping imp is clear from the context, then it may be elided and we write $\phi(tr)$*.*

3.3 Refinement

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

Definition 4 (trace refinement relative to *imp*).

$$P \sqsubseteq^T_{imp} P' \text{ iff } traces(P') \leq \bigcup_{tr \in traces(P)} \phi_{imp}(tr)$$

Observe that if *imp* is the identity function (strictly, that $imp(a) = \{\langle a \rangle\}$), then $P \sqsubseteq^T_{imp} P'$ is simply trace refinement.

Example 1. Consider $imp(a) = \{\langle c, d \rangle, \langle e, f \rangle\}$, and

$$P = a \rightarrow STOP \qquad P' = (c \rightarrow d \rightarrow STOP) \,\Box\, (e \rightarrow f \rightarrow STOP)$$

Then $P \sqsubseteq^T_{imp} P'$. Every trace of P' is an implementation of a trace of P.

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

Example 2. Consider $imp(a) = imp(b) = \{\langle c \rangle\}$, and

$$P_1 = a \rightarrow STOP \qquad P_2 = b \rightarrow STOP \qquad P'_1 = P'_2 = c \rightarrow STOP$$

Observe that $P_1 \sqsubseteq^T_{imp} P'_1$ and $P_2 \sqsubseteq^T_{imp} P'_2$. However $P_1 \parallel P_2 = STOP$, and also $P'_1 \parallel P'_2 = c \rightarrow STOP$, so the refinement relation does not hold between $P_1 \parallel P_2$ and $P'_1 \parallel P'_2$: we have $\langle c \rangle \in traces(P'_1 \parallel P'_2)$ but $\langle c \rangle \notin \pi_{imp}(traces(P_1 \parallel P_2))$.

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 completely unconnected, and there are no restrictions on them. In the traces model, B machines without i/o all have the same semantics: all possible traces. Thus this theorem requires consideration only of the relationship between P and P'.

Although this theorem is true for trivial reasons, it has been included because it is the kind of compositionality result we aim for, and will aim to establish similar (but non-trivial) results in other semantic models.

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

Definition 5 (mapping abstract to concrete processes).
If $imp : A \rightarrow C^*$ *is an implementation mapping, then we define the mapping*

Θ_{imp} *on CSP process descriptions as follows:*

$$\begin{aligned}
\Theta_{imp}(STOP) &= STOP \\
\Theta_{imp}(a \rightarrow P) &= \Box_{tr \in imp(a)}\, Pref(tr, \Theta_{imp}(P)) \\
\Theta_{imp}(P_1 \,\Box\, P_2) &= \Theta_{imp}(P_1) \,\Box\, \Theta_{imp}(P_2) \\
\Theta_{imp}(X) &= X \\
\Theta_{imp}(\mu\, X.P) &= \mu\, X.\Theta_{imp}(P)
\end{aligned}$$

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

Example 3. Consider $imp(a) = \{\langle c, d\rangle, \langle e, f\rangle\}$, and $imp(b) = \{\langle c, e\rangle\}$. Consider the process $\mu\, X.a \rightarrow b \rightarrow X$. Then

$$\begin{aligned}
&\Theta(\mu\, X.a \rightarrow b \rightarrow X) \\
&= \mu\, X.\Theta(a \rightarrow b \rightarrow X) \\
&= \mu\, X.(Pref(\langle c, d\rangle, \Theta(b \rightarrow X)) \,\Box\, Pref(\langle e, f\rangle, \Theta(b \rightarrow X)) \\
&= \mu\, X.((c \rightarrow d \rightarrow \Theta(b \rightarrow X) \,\Box\, (e \rightarrow f \rightarrow \Theta(b \rightarrow X)) \\
&= \mu\, X.((c \rightarrow d \rightarrow c \rightarrow e \rightarrow \Theta(X)) \,\Box\, (e \rightarrow f \rightarrow c \rightarrow e \rightarrow \Theta(X))) \\
&= \mu\, X.((c \rightarrow d \rightarrow c \rightarrow e \rightarrow X) \,\Box\, (e \rightarrow f \rightarrow c \rightarrow e \rightarrow X))
\end{aligned}$$

The full expansion of $\Theta(b \rightarrow X)$ is $\Box_{tr \in \{\langle c, e\rangle\}}\, Pref(tr, \Theta(X))$. This is a singleton choice (i.e. a general choice with only one branch), so is equivalent in CSP to the single process that may be chosen: $Pref(\langle c, e\rangle, \Theta(X))$, so we have made this simplification in the relevant steps above.

The mapping Θ has been constructed to yield the following theorem: that the result of the transformation is a refinement of the original process.

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

This is proved by structural induction on P. The proof is in Appendix A

Example 4. This example shows how our notion of refinement, with respect to an implementation mapping, can be used to introduce design detail into a specification. The example is inspired by the mechanical press controller introduced by Abrial in [3]. We first identify basic events that provide the minimum information required to represent the behaviour of starting and stopping a car.

Consider the following events: *treat_start_motor* (with concrete version *tsm*) that turns on the starter motor, *motor_start* that turns on the engine, and conversely *motor_stop* that turns off the engine. We then define the process *Car* to represent a possible pattern of behaviour:

$$Car = treat\_start\_motor \rightarrow motor\_start \rightarrow motor\_stop \rightarrow Car$$

In our refinement we wish to model the user interaction of turning a key which eventually results in starting the engine, and in older cars this sometimes required the additional manual intervention of turning a crank. In our model we will introduce the event *turn_start_key* (*tsk*) to represent the notion of the user wanting to start the car and the event *crank* to represent manual intervention by the user. We will allow up to two failures of the starter motor before manual intervention may be required. A further event *treat_start_motor_false* (*tsmf*) is introduced to represent failure of the starter motor. This notion of a specific action resulting in a reaction is described in [3] as an action/reaction pattern.

We define an appropriate implementation mapping as follows:

$$\begin{aligned} imp(motor\_start) &\mathrel{\widehat{=}} \{\langle motor\_start\rangle\} \\ imp(motor\_stop) &\mathrel{\widehat{=}} \{\langle motor\_stop\rangle\} \\ imp(treat\_start\_motor) &\mathrel{\widehat{=}} \{\langle tsk, tsm\rangle, \\ &\qquad \langle tsk, tsmf, tsk, tsm\rangle, \\ &\qquad \langle tsk, tsmf, tsk, tsmf, crank, tsm\rangle\} \end{aligned}$$

Then $Car' = \Theta_{imp}(Car)$ can be defined as follows (with some rewriting of the external choice permitted by CSP laws). Theorem 2 above ensures that Car' is a refinement of Car.

$$\begin{aligned} Car' = tsk \rightarrow (&tsm \rightarrow motor\_start \rightarrow motor\_stop \rightarrow Car' \\ \Box\ &tsmf \rightarrow tsk \rightarrow (tsm \rightarrow motor\_start \rightarrow motor\_stop \rightarrow Car' \\ \Box\ &tsmf \rightarrow crank \rightarrow tsm \rightarrow \\ &motor\_start \rightarrow motor\_stop \rightarrow Car')) \end{aligned}$$

Notice that we have retained performing the *treat_start_motor* event following the *crank* event. We could also have provided a refinement where, after performing *crank*, the motor started automatically. We do not have to retain the event being refined in the implementation mapping in all its corresponding concrete sequences (or even at all).

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. We first define weakest preconditions on sequences of operations:

Definition 6. *The weakest precondition for a sequence to establish a postcondition I is defined inductively as follows:*

$$[\langle\rangle]I = I$$
$$[\langle a\rangle \frown tr]I = [a]([tr]I)$$

This definition applies to traces which include events e not in the alphabet of a machine M. Such events have no effect on the machine state, and so in such cases we define $[e]I = I$. It follows that $[tr]I = [tr \upharpoonright \alpha M]I$.

Definition 7 (refinement of B machines).
If M and M' have linking invariant J, then

$$M \sqsubseteq^B_{imp} M' \textit{ iff } \forall\, a \in dom(imp), tr \in imp(a)\ .\ I \wedge J \wedge P \Rightarrow [tr]\neg[a]\neg J$$

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 concrete sequences of operations corresponding 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. This requires one further construction: the set of non-empty prefixes of concrete traces.

Definition 8. *Given an implementation mapping $imp : A \to \mathbb{P}(C^*)$ and $a \in A$, we define $imp^+(a) \mathrel{\widehat{=}}\downarrow imp(a) - \{\langle\rangle\}$.*

This is used in the definition of the mapping ψ to follow. This mapping identifies all concrete sequences relating precisely to an abstract sequence of events, rather than a prefix of it. This definition will be used in Theorem 4 with regard to the point at which a concrete trace diverges.

Definition 9 (subsequence implementation mapping).
The function $\psi_{imp} : A^* \to \mathbb{P}(C^*)$ is defined as follows:

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

Example 5. Consider

$$imp(a) = \{\langle c, d\rangle, \langle e, f\rangle\}$$
$$imp(b) = \{\langle c, e\rangle\}$$

Then

$$\psi_{imp}(\langle a, b\rangle) = \{\langle c, d, c\rangle, \langle c, d, c, e\rangle, \langle e, f, c\rangle, \langle e, f, c, e\rangle\}$$

We see that $\psi(\langle a, b\rangle)$ is those traces that correspond to the full sequence $\langle a, b\rangle$, and not just prefixes of it. Thus we have the sequences associated with a followed by non-empty parts of the sequences associated with b.

ψ_{imp} will be used to capture a definition of traces/divergences refimement.

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

$$P \sqsubseteq^{TD}_{imp} P' \textit{ iff } P \sqsubseteq^{T}_{imp} P' \tag{1}$$

$$\wedge\ \textit{divergences}(P') \subseteq \uparrow_{\alpha P'} \Big(\bigcup_{tr \in \textit{divergences}(P)} \psi(tr) \Big) \tag{2}$$

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 6. Continuing Example 5 above, if $\langle a, b\rangle$ is an abstract divergence, then $\psi_{imp}(\langle a, b\rangle)$ is the set of concrete divergences 'allowed' by $\langle a, b\rangle$. In other words, if the abstract system diverges on performance of the event b, having previously performed a, then the concrete system can diverge at some point after the start of a sequence associated with b.

Example 7. Consider $imp(a) = imp(b) = \{\langle c, d\rangle, \langle e, f\rangle\}$. Let

$$P = a \rightarrow (a \rightarrow P \,\Box\, b \rightarrow \bot)$$
$$P' = (c \rightarrow d \rightarrow e \rightarrow f \rightarrow P') \,\Box\, (e \rightarrow f \rightarrow c \rightarrow \bot)$$

[$\bot$ is shorthand for $\mu\, X.X$, a process that immediately diverges.] Then we obtain $P \sqsubseteq^{TD}_{imp} P'$. Every trace of P' is reflected in some trace of P, and every divergence of P' also is allowed by a divergence of P.

We obtain a similar result to Theorem 2, this time for traces/divergences, again proved by structural induction over P.

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. This theorem states that in a controlled component, the CSP controller, and the B machine can each be refined independently to yield an overall refinement.

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

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

$$traces(M') = (\alpha M')^* \quad where \quad \alpha M' \subseteq \alpha P' \tag{4}$$

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

Then consider $tr' \in divergences(P' \parallel M')$. Then let tr'_0 be the minimal divergent prefix of tr'. From the definition of $\sqsubseteq^{TD}_{imp}$ refinement it is sufficient to establish that $tr'_0 \in\uparrow \psi(tr_0)$ for some $tr_0 \in divergences(P \parallel M)$.

By the divergence semantics of the parallel operator, there are two possibilities for how the divergence tr'_0 has arisen: from a divergence of P' or from a divergence of M':

- **Case** $tr'_0 \in divergences(P')$ and $tr'_0 \upharpoonright \alpha M' \in traces(M')$. Then from the fact that $P \sqsubseteq^{TD}_{imp} P'$, there is some $tr_0 \in divergences(P)$ such that $tr'_0 \in\uparrow (\psi(tr_0))$. Then $tr_0 \upharpoonright \alpha M \in traces(M)$. Thus $tr_0 \in divergences(P \parallel M)$, which establishes the case.
- **Case** $tr'_0 \in traces(P')$ and $tr'_0 \upharpoonright \alpha M' \in divergences(M')$. Then let tr_0 be a *minimal* trace of P such that $tr'_0 \leqslant \phi(tr_0)$.
  We have that $tr_0 \upharpoonright \alpha M \in divergences(M)$ by Lemma 2. Hence $tr_0 \in divergences(P \parallel M)$.
  Also, $tr'_0 \in \psi(tr_0)$, since tr_0 is minimal. Therefore $tr'_0 \in\uparrow \psi(tr_0)$, which establishes the case.

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

Lemma 1 states that if a sequence tr for M is guaranteed to terminate, then any implementation of M is guaranteed to terminate on any sequence tr' which is an implementation of tr.

Lemma 1. *If* $M \sqsubseteq^{B}_{imp} M'$ *and* $tr' \in \phi(tr)$ *then* $I \wedge J \wedge [tr]true \Rightarrow [tr']true$

Proof. We prove the result by induction over tr.

Case $\langle\rangle$: the result holds trivially, since $\phi(\langle\rangle) = \{\langle\rangle\}$.

Case $\langle a\rangle \frown tr$: consider $tr' \in \phi(\langle a\rangle \frown tr)$. Then $I \wedge J \wedge P_a \Rightarrow [tr']\neg[\langle a\rangle \frown tr]\neg J$ from the repeated application of the definition $M \sqsubseteq^{B}_{imp} M'$ to sequences:

following the concrete sequence tr', the linking invariant J can be re-established by some execution of the abstract sequence $\langle a \rangle \frown tr$.

$$\begin{aligned}
[tr']true &\Leftarrow [tr'](\neg[\langle a \rangle \frown tr]\neg J) \\
&\Leftarrow I \wedge J \wedge P_a \\
&\Leftarrow I \wedge J \wedge (P_a \wedge [S_a]true) \\
&\Leftarrow I \wedge J \wedge [a]true \\
&\Leftarrow I \wedge J \wedge [a]([tr]true) \\
&\Leftarrow I \wedge J \wedge [\langle a \rangle \frown tr]true
\end{aligned}$$

as required. Note that the operation a expands to $PRE\ P_a\ THEN\ S_a\ END$.

Lemma 2 is the converse: it states that for related machines M and M', any divergence of a concrete machine M' related to an abstract sequence must have that abstract sequence as a divergence of M. Thus concrete divergences relate to abstract divergences.

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

Proof. If $tr' \leqslant \phi(tr)$ then $\exists\, tr''.tr' \leqslant tr'' \wedge tr'' \in \phi(tr)$. We prove the contrapositive: that if $tr \restriction \alpha M$ is not a divergence of M then $tr' \restriction \alpha M'$ is not a divergence of M'.

$$\begin{aligned}
[tr \restriction \alpha M]true &= [tr]true \\
&\Rightarrow [tr'']true \qquad \text{by Lemma 1} \\
&\Rightarrow [tr']true \qquad \text{since } tr' \leqslant tr'' \\
&= [tr' \restriction \alpha M']true
\end{aligned}$$

Theorem 4, unlike Theorem 1, requires the refinement relationship to hold between the component 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, so the concrete machine can diverge only where the abstract machine description allows it.

Example 8. 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
$$b \mathrel{\widehat{=}} PRE\ even(mm)\ THEN\ mm := mm + 1\ END, \text{ and}$$
$$c \mathrel{\widehat{=}} PRE\ \neg even(mm)\ THEN\ mm := mm + 3\ END.$$

In this example 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 between M and M' is in the context of *imp* so only those sequences of M' 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 7. 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 not relevant 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.

Now 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 $P \parallel M \sqsubseteq^{TD}_{imp} P' \sqsubseteq M'$.

Example 9. Recall Example 4 which illustrated the behaviour of a car starter motor. In this example, we revise the example to include explicit state. We restrict our focus to the starting of the engine, and will not refine the stopping of the engine. Tracking state explicitly allows us to model the fact that the starter motor is not always running once the engine is switched on. We associate the *treat_start_motor* and *motor_start* events with B operations, as shown in the `CarM` machine in Figure 1. Observe that the *motor_start* operation turns off the starter motor when it starts the engine.

In the refinement we change our notion of crank and also associate the crank event with a corresponding B operation in the `CarM'` machine. When manual intervention occurs the engine is turned on without having to go through the intermediate step of turning on the starter motor. Therefore, we identify an alternative implementation mapping as follows:

$$\begin{aligned} imp(motor\_start) &\mathrel{\widehat{=}} \{\langle\rangle\} \\ imp(motor\_stop) &\mathrel{\widehat{=}} \{\langle motor\_stop\rangle\} \\ imp(tsm) &\mathrel{\widehat{=}} \{\langle tsk, tsm, motor\_start\rangle, \\ &\qquad \langle tsk, tsmf, tsk, tsm, motor\_start\rangle, \\ &\qquad \langle tsk, tsmf, tsk, tsmf, crank\rangle\} \end{aligned}$$

The refinement step itself expands the interface of the system in order to incorporate the notion of manual intervention. In order to show that $Car' \parallel$ `CarM'` is

an appropriate refinement of $Car \parallel \texttt{CarM}$, we apply Theorem 4. We must show that the machines are refinements of each other by applying Definition 7. The linking invariant J required to meet the condition of Definition 7 is the following:

$$\begin{aligned} J = (&startMotor = startMotorR = off \\ &\wedge\ engine = engineR) \\ &\vee \\ &(startMotor = on \wedge startMotorR = off \\ &\wedge\ engine = off \wedge engineR = on) \end{aligned}$$

Thus, for each operation related to an event in the domain of *imp* it is straightforward to show that the corresponding sequences of concrete operations are appropriate refinements of the operations. We need not relate all the CSP events with B operations in a refinement step, and this is illustrated by the introduction of the *turn_start_key* event. In the weakest precondition proof *turn_start_key* corresponds to the AMN statement `skip`. We must also show that the *Car* process:

$$Car' = tsm \rightarrow motor\_start \rightarrow motor\_stop \rightarrow Car'$$

is refined by the following revised *Car'* process:

$$\begin{aligned} Car' = tsk \rightarrow (\ &tsm \rightarrow motor\_start \rightarrow motor\_stop \rightarrow Car') \\ \Box\ (\ &tsmf \rightarrow tsk \rightarrow (tsm \rightarrow motor\_start \rightarrow motor\_stop \rightarrow Car') \\ &\Box\ (tsmf \rightarrow crank \rightarrow motor\_stop \rightarrow Car')) \end{aligned}$$

Since $Car' = \Theta_{imp}(Car)$ the refinement holds by application of Theorem 3.

Ths example has shown that new events can be introduced in a refinement and do not have any corresponding abstract operations, and also that new events can correspond to new operations.

4 The machine 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. In this context the implementation mapping $imp : A \rightarrow \mathbb{P}(C^*)$ has C as CSP channel names, corresponding to B operation names. C does not include input and output values explicitly.

```
MACHINE     CarM
VARIABLES startMotor, engine
INVARIANT
    startMotor : {on, off} &
    engine : {on,off}
OPERATIONS
  treat_start_motor =
      PRE engine = off &
          startMotor = off
      THEN startMotor := on
      END;
  motor_start =
      PRE engine = off &
          startMotor = on
      THEN startMotor := off ||
           engine := on
      END
END
```

```
MACHINE     CarM'
VARIABLES startMotorR, engineR
INVARIANT
    startMotorR : {on, off} &
    engineR : {on,off}
OPERATIONS
  treat_start_motor (as previous);
  motor_start (as prevous);
  treat_start_motor_false =
      PRE engineR = off &
          startMotorR = off
      THEN skip
      END;
  crank =
      PRE engineR = off &
          startMotorR = off
      THEN engineR := on
      END
END
```

Fig. 1. Car Machine examples

4.1 Refining B operations

For a given event a with $cs = \langle c_1, \ldots, c_n \rangle \in imp(a)$, 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 also 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 7. 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

$$M \sqsubseteq^{B}_{imp,r} M' \textit{iff}$$
$$\forall\, a \in \alpha M, cs \in imp(a) \;.\; r_{in,a} \wedge I \wedge J \wedge P_a \Rightarrow [cs]\neg[a]\neg(J \wedge r_{out,a})$$

4.2 Examples illustrating aspects of Definition 11

Example 10 (Implementation modulo 5*).* The example in Figure 2 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.

```
MACHINE     Times3
OPERATIONS
  yy <-- triple(xx) =
      PRE xx : NAT
      THEN yy := 3 * xx
      END
END
```

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

Fig. 2. Tripling, modulo 5

*Times*3*R* is a refinement of *Times* with $imp(triple) = \{\langle tripleR \rangle\}$ and the following definitions, which together satisfy the condition of Definition 11:

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

Example 11 (Change of unit). In the example of Figure 3, we change the units (i.e. the degree of sensitivity) being read by the system from centimetres to millimetres, but retain the use of centimetres in outputs.

CounterR is a refinement of *Counter* under the following conditions:

The *imp* function giving implementations of the abstract operations is as follows:

$$imp(add) = \{\langle addR, countR \rangle, \langle countR, addR \rangle\}$$
$$imp(mean) = \{\langle meanR \rangle\}$$

For the *add* operation, we capture the relationship between the inputs and outputs at abstract and concrete levels:

$$r_{in,add} = 10nn - 5 \leq nnR \leq 10nn + 5$$

For *mean*, the relationship is on outputs:

$$r_{out,mean} = mm - 1 \leqslant mmR \leqslant mm + 1$$

```
MACHINE         Counter
VARIABLES       totcm, num
INVARIANT       totcm : NAT
                  & num : NAT

INITIALISATION  totcm := 0
                  || num := 0
OPERATIONS
  add(nn) =
      PRE nn : NAT
      THEN totcm := totcm + nn
           || num := num + 1
      END;

mm <-- mean =
      PRE num > 0
      THEN mm := totcm / num
      END
END
```

```
MACHINE         CounterR
VARIABLES       totmmR, numR
INVARIANT       totmmR : NAT
                  & numR : NAT
                  & num = numR
                  & 10*tot - 5*num <= totR
                  & totR <= 10*tot + 5*num

INITIALISATION  totmmR := 0
                  || numR := 0
OPERATIONS
  addR(nnR) =
      PRE nnR : NAT
      THEN totmmR := totmmR + nnR
      END;

  countR =
      BEGIN
        numR := numR + 1
      END;

mmR <-- meanR =
      BEGIN
        mmR := totmmR / (10 * numR)
      END
END
```

Fig. 3. Change of unit

The linking invariant is

$$J = (num = numR) \wedge (10 * totcm - 5 * num \leqslant totmmR \leqslant 10 * totcm + 5 * num)$$

The condition of Definition 11 is met by these definitions, for both *add*, and *mean*.

To establish this for $\langle addR, countR \rangle \in imp(add)$ the steps are as follows:

$$\begin{aligned}
&[addR;\ countR]\neg[totcm := totcm + nn \parallel num := num + 1]\neg J \\
&= [addR;\ countR](\,(num + 1 = numR) \wedge \\
&\qquad\qquad (10 * (totcm + nn) - 5 * (num + 1) \leqslant totmmR \wedge \\
&\qquad\qquad totmmR \leqslant 10 * (totcm + nn) + 5 * (num + 1))) \\
&= ((num + 1 = numR + 1) \wedge \\
&\quad (10 * (totcm + nn) - 5 * (num + 1) \leqslant totmmR + nnR \wedge \\
&\quad totmmR + nnR \leqslant 10 * (totcm + nn) + 5 * (num + 1))) \\
&\Leftarrow J \wedge (10 * nn - 5 \leqslant nnR \leqslant 10 * nn + 5) \\
&= J \wedge r_{in,add}
\end{aligned}$$

The steps are identical for $\langle countR, addR \rangle \in imp(add)$, thus covering all sequences in $imp(add)$.

For $\langle meanR \rangle \in imp(mean)$ the reasoning is as follows:

$$\begin{aligned}
&[meanR]\neg[mm := totcm/num]\neg(J \wedge r_{out,mean}) \\
&= [mmR := totmmR/mmR](J \wedge num \neq 0 \wedge totcm/num - 1 \leqslant mmR \leqslant totcm/num + 1) \\
&= (J \wedge num \neq 0 \wedge (totcm/num) - 1 \leqslant totmmR/(10 * numR) \leqslant (totcm/num) + 1) \\
&\Leftarrow J \wedge P_{mean} \wedge (10 * totcm - 5 * num \leqslant totmmR \leqslant 10 * totcm + 5 * num) \\
&= J \wedge P_{mean}
\end{aligned}$$

Example 12. Change of offset I In this example we change the offset of the readings, so that concrete inputs are the abstract inputs offset by +1.

```
MACHINE         Increase
VARIABLES       total
INVARIANT       total : NAT
INITIALISATION  total := 0
OPERATIONS
  add(xx) = PRE xx : NAT
            THEN total := total + xx
            END
END

MACHINE         IncreaseR
```

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

In this example, we have a change of offset: 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* with the following definitions:

$$\begin{aligned} imp(add) &= \langle addR \rangle \\ J &= totalR = total \\ r_{in,add} &= ww = xx - 1 \end{aligned}$$

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

$$\begin{aligned} & [addR]\neg[total := total + xx]\neg(J \wedge r_{out,add}) \\ & = [addR](total + xx = totalRR) \\ & = ww \in NAT \wedge (total + xx = totalR + ww - 1) \\ & \Leftarrow xx \in NAT \wedge xx = ww - 1 \wedge total = totalR \\ & = P_{add} \wedge r_{in,add} \wedge J \end{aligned} \tag{6}$$

Example 13 (Change of offset II). The example in Figure 4 is similar to the previous example, except that the concrete inputs are the abstract inputs offset by -1.

```
MACHINE         Increase
VARIABLES       total
INVARIANT       total : NAT
INITIALISATION  total := 0
OPERATIONS
  add(xx) =
      PRE xx : NAT
      THEN total := total + xx
      END
END
```

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

Fig. 4. Change of offset II

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 6 does not carry through, since an abstract 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: the abstract value 0 would be represented by -1.

Examples 12 and 13 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 12 that the converse is not the case: the concrete input 0 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 14 (distributing inputs). This example illustrates an abstract operation with two input parameters being implemented by to operations each accepting one of the inputs. We describe a sensor machine that is taking temperature and pressure readings at particular points in a single operation. This can be refined by a machine that takes temperature and pressure readings in separate operations. Our implementation mapping *imp* ensures that the readings match.

```
MACHINE         Sensor
VARIABLES       tt, pp
INVARIANT       tt : NAT
                 & pp : NAT
INITIALISATION  tt :: NAT
                 || pp :: NAT
OPERATIONS
  update(dt,dp) =
      PRE dt : NAT & dp : NAT
      THEN tt := tt + dt
            || pp := pp + dp
      END
END
```

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

Fig. 5. Distributing inputs

In Figure 5, *SensorR* is a refinement of *Sensor* with the following definitions:

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

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.

4.3 Trace refinement for processes

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

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

The function ϕ then generalises as follows:

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

This supports the natural extension to the 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).
$P \sqsubseteq^T_{imp,r} P'$ *iff* $traces(P') \leq \ (\bigcup_{tr \in traces(P)} \phi_{imp,r}(tr))$

Example 15. Recall Example 11 concerning a change of unit from centimetres to millimetres. In that case we had

$$r_{in,add} = 10nn - 5 \leq nnR \leq 10nn + 5$$

In this case, if $P \sqsubseteq^T_{imp,r} P'$ then $add?nn : \{1..10\} \rightarrow P$ will be refined by $addR?nnR : \{5..105\} \rightarrow P'$. Every concrete input corresponds to some abstract input.

We have now 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:

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

Note that the traces of machines M are no longer all possible traces, since they constrain the possible outputs. Hence the conjecture takes the machine traces into account, since they restrict the overall behaviour.

5 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. The implementation mapping permitted is more general than that presented in [21]. Previously, any event that was present in both the abstract and the concrete process required the event to be represented as the identity within *imp* but this is no longer necessary. Example 9 illustrated the benefit of the weakening of the definition so that *motor_start* could be represented appropriately in the refinement (i.e., it need not always be invoked).

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. In the case of refining events by processes, the aim is to replace the abstract performance of a single action with the execution of a process. Termination (via *SKIP*) of the process will correspond to completion of the action, and it will be necessary to establish correspondence of machine states at that point. However, in general we may wish to make use of non-terminating processes. For example, Example 4 allows two retries to start the engine before a manual *crank* becomes necessary. If we wish to do away with the crank and allow arbitrary retries, then we would use the process

$$imp(treat\_start\_motor) = \mu\, X \bullet tsk \rightarrow (tsm \rightarrow SKIP \,\Box\, tsmf \rightarrow X)$$

This process does not necessarily terminate. The issue of whether we wish to consider a process such as this as a refinement of *treat_start_motor* will depend on whether it is acceptible in the context concerned for *treat_start_motor* to possibly not terminate. For example, if this is not acceptible then an additional proof obligation regarding termination will also be necessary. For example, not starting successfully might be acceptible from a safety point of view, but not stopping successfully might not be. Thus different abstract events might be associated with different requirements.

In this paper we have restricted ourselves to refining single processes. As a continuation of the work we will also be investigating refining events which occur within parallel compositions of processes and machines. Considerations such as refining disjoint events to shared events would need to be made. Decomposition using shared events is discussed in [8] and the ideas presented in that paper are particularly relevant to our ongoing work.

In [11], 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 [6, 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 [12] Derrick and Wehrheim bring together the ideas from [11] and [6] 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 [7].

The aim of developing interface refinement techniques in CSP‖B is so that we can use the refinement process to introduce more detail into a model of a system.

This refinement philosophy is the strength of Event-B and it is clear that it allows designs to evolve to include more detail during refinement steps. Other work that has also been investigating increasing the level of detail during a refinement of a CSP‖B specification is [9] and [10] . The authors use the notion of component refinement; they introduce new controlled process/machine components and by using renaming and hiding demonstrate that a specification which contains new components is a refinement of a more abstract one.

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. [4], 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.

Acknowledgments Thanks to Thai Son Hoang for discussions related to Event-B. Thanks to James Sharp and Edward Turner for discussions on the examples in the paper. 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. J-R. Abrial, T. S. Hoang, and S. Matthias. A mechanical press controller, 2008. Deliverable D8 D10.1 Teaching Materials http://deploy-eprints.ecs.soton.ac.uk/54/.
4. L. Aceto. *Action Refinement in Process Algebras*. Cambridge University Press, 1992.
5. J.A. Bergstra, A. Ponse, and S.A. Smolka, editors. *Handbook of Process Algebra*. North-Holland, 2001.
6. E. Boiten and J. Derrick. *Refinement in Z and Object-Z: Foundations and Advanced Applications*. Springer-Verlag, 2001.
7. M. Butler. personal communication, September, 2008.
8. Michael Butler. Decomposition structures for Event-B. In *IFM*, pages 20–38, 2009.
9. S. Colin, A. Lanoix, O. Kouchnarenko, and J. Souquières. Using CSP‖B Components: Application to a Platoon of Vehicles. In *13th International ERCIM Wokshop on Formal Methods for Industrial Critical Systems (FMICS 2008)*, LNCS. Springer-Verlag, September 2008.
10. Samuel Colin, Arnaud Lanoix, Olga Kouchnarenko, and Jeanine Souquières. Towards validating a platoon of cristal vehicles using CSP‖B. In J. Meseguer and G. Rosu, editors, *12th International Conference on Algebraic Methodology and SoftwareTechnology (AMAST 2008)*, number 5140 in LNCS, pages 139–144. Springer-Verlag, July 2008.

11. J. Derrick and E. Boiten. Non-atomic refinement in Z. In *FM '99 Volume II*, pages 1477–1496, London, UK, 1999. Springer-Verlag.
12. J. Derrick and H. Wehrheim. Using coupled simulations in non-atomic refinement. In *ZB*, 2003.
13. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.
14. C. Morgan. Of wp and CSP. *Beauty is our business: a birthday salute to E. W. Dijkstra*, pages 319–326, 1990.
15. A. W. Roscoe. *The theory and practice of concurrency.* Prentice Hall, 1998.
16. S. Schneider. *Concurrent and Real-Time Systems: the CSP Approach.* Wiley, 1999.
17. S. Schneider. *The B-Method: an introduction.* Palgrave, 2001.
18. S. Schneider, D. Pizarro, and H. Treharne. The Futuretech demonstrator, 2008. Future Technologies for System Design Technical Report, University of Surrey.
19. S. Schneider and H. Treharne. CSP theorems for communicating B machines. *Formal Asp. Comput.*, 17(4):390–422, 2005.
20. Steve Schneider and Helen Treharne. Changing system interfaces consistently: A new refinement strategy for CSP||B. In *IFM*, pages 103–117, 2009.
21. Steve Schneider and Helen Treharne. Changing system interfaces consistently: A new refinement strategy for CSP‖B. In *IFM*, pages 103–117, 2009.

A Proof of Theorem 2

We first require a technical lemma regarding the traces of $Pref(tr, P)$. $Pref(tr, P)$ is concerned with CSP processes at the syntactic level. The intention of the definition is that $Pref(tr, P)$ prefixes P with the sequence of events listed in tr. This lemma shows that the semantics of the resulting process is dominated by the traces of P prefixed with tr.

Lemma 3.

$$traces(Pref(tr, P)) \leq \{tr\} \frown traces(P)$$

Proof. By induction on tr.

Case $\langle\rangle$. Then

$$\begin{aligned} traces(Pref(\langle\rangle, P)) &= traces(P) \\ &= \{\langle\rangle\} \frown traces(P) \end{aligned}$$

which establishes ths case.

Case $\langle a\rangle \frown tr$. Then

$$\begin{aligned} traces(Pref(\langle a\rangle \frown tr), P) &= traces(a \rightarrow Pref(tr, P)) \\ &= \{\langle\rangle\} \cup \{\langle a\rangle\} \frown traces(Pref(tr, P)) \\ &\leq \{\langle\rangle\} \cup \{\langle a\rangle\} \frown \{tr\} \frown traces(P) \\ &= \{\langle\rangle\} \cup \{\langle a\rangle \frown tr\} \frown traces(P) \\ &\leq \{\langle a\rangle \frown tr\} \frown traces(P) \end{aligned}$$

which establishes the case.

We are now in a position to prove the theorem:

Theorem 2 $\forall\, imp, P\ .\ P \sqsubseteq^T_{imp} \Theta_{imp}(P)$

Proof. The proof is by structural induction over process terms. We must first introduce *environmental mappings*, which map process variables to processes in the traces model. A process term P in the context of an environmental mapping ρ is written $[P]\rho$ associates a set of traces with each free process variable in the process term, enabling us to associate trace sets with any process term.

Formally, an environmental mapping ρ is a function mapping process variables X to sets of traces with the appropriate alphabet. A refinement relation $\sqsubseteq^T_{imp}$ can be defined on environmental mappings as follows:

Definition 14. *If* $\rho_A : VAR \to \mathbb{P}(A^*)$ *and* $\rho_C : VAR \to \mathbb{P}(C^*)$ *then*

$$\rho_A \sqsubseteq^T_{imp} \rho_C \textit{ iff } \forall\, X \in VAR.\rho_A(X) \sqsubseteq^T_{imp} \rho_C(X)$$

We will then prove the following by structural induction over process terms: for any environmental mappings ρ_A and ρ_C, if $\rho_A \sqsubseteq^T_{imp} \rho_C$ then $[P]\rho_A \sqsubseteq^T_{imp} [\Theta(P)]\rho_C$.

Case $STOP$: In this case $\Theta(STOP) = STOP$, and so $traces[\Theta(STOP)]\rho_C = \{\langle\rangle\}$. Therefore

$$[STOP]\rho_A \sqsubseteq^T_{imp} [\Theta(STOP)]\rho_C$$

Case X: In this case $\Theta(X) = X$, and we have $\rho_A(X) \sqsubseteq^T_{imp} \rho_C(X)$ by the original condition on the relationship between ρ_A and ρ_C. i.e.

$$[X]\rho_A \sqsubseteq^T_{imp} [\Theta(X)]\rho_C$$

Case $a \to P$: Assume the inductive hypothesis for P.

$$
\begin{aligned}
&traces([\Theta(a \to P)]\rho_C) \\
&= traces(\Box_{tr \in imp(a)}\, Pref(tr, [\Theta(P)]\rho_C) \\
&= \bigcup_{tr \in imp(a)} traces(Pref(tr, [\Theta(P)]\rho_C)) \\
&\leq \bigcup_{tr \in imp(a)} (\{tr\} \frown traces([\Theta(P)]\rho_C)) \qquad \text{by Lemma 3} \\
&= (\bigcup_{tr \in imp(a)} \{tr\}) \frown traces([\Theta(P)]\rho_C)) \\
&= imp(a) \frown traces([\Theta(P)]\rho_C)) \\
&\leq imp(a) \frown \bigcup_{tr \in traces([P]\rho_A)} \phi(tr) \qquad \text{by Inductive Hypothesis} \\
&= \bigcup_{tr \in traces([P]\rho_A)} imp(a) \frown \phi(tr) \\
&= \bigcup_{tr \in traces([P]\rho_A)} \phi(\langle a \rangle \frown tr) \\
&\leq \bigcup_{tr \in traces([a \to P]\rho_A)} \phi(tr)
\end{aligned}
$$

which establishes the case.

Case $P_1 \,\Box\, P_2$: Assume the inductive hypothesis for P_1 and P_2:

$$
\begin{aligned}
[P_1]\rho_A &\sqsubseteq [\Theta(P_1)]\rho_C \\
[P_2]\rho_A &\sqsubseteq [\Theta(P_2)]\rho_C
\end{aligned}
$$

Then

$$
\begin{aligned}
&traces([\Theta(P_1 \,\Box\, P_2)]\rho_C) \\
&= traces([\Theta(P_1) \,\Box\, \Theta(P_2)]\rho_C) \\
&= traces([\Theta(P_1)]\rho_C) \cup traces([\Theta(P_2)]\rho_C) \\
&\leq \bigcup_{tr \in traces([P_1]\rho_A)} \phi(tr) \quad \cup \quad \bigcup_{tr \in traces([P_2]\rho_A)} \phi(tr) \\
&= \bigcup_{tr \in traces([P_1]\rho_A) \cup traces([P_2]\rho_A)} \phi(tr) \\
&= \bigcup_{tr \in traces([P_1 \Box P_2]\rho_A)} \phi(tr)
\end{aligned}
$$

Therefore

$$[P_1 \,\Box\, P_2]\rho_A \sqsubseteq^T_{imp} [\Theta(P_1 \,\Box\, P_2)]\rho_C$$

Case $\mu\, X.P$: Assume the inductive hypothesis for P, and consider $\rho_A \sqsubseteq^T_{imp} \rho_C$.

Define F to be the following function: F maps a process Y to $\Theta(P)$ in the environment ρ_C with Y substituted for X:

$$F(Y) = [\Theta(P)](\rho_C[Y/X])$$

We prove inductively that $[\mu\, X.P]\rho_A \sqsubseteq^T_{imp} F^n(STOP)$.

Case 0. This follows immediately, from

$$traces(STOP) = \{\langle\rangle\} \leq \bigcup_{tr \in traces([\mu\, X.P]\rho_A)} \phi(tr)$$

Case $n+1$. Assume the result holds for n. Then

$$\begin{array}{ll} & [\mu\, X.P]\rho_A \\ = & \{[\mu\, X.P]\rho_A \text{ is a fixed point of } \lambda\, Y.[P]\rho_A[Y/X]\} \\ & [P](\rho_A[([\mu\, X.P]\rho_A)/X]) \\ \sqsubseteq^T_{imp} & \{\text{by inductive hypothesis, since } \rho_A[([\mu\, X.P]\rho_A)/X] \sqsubseteq^T_{imp} \rho_C[F^n(STOP)/X]\} \\ & [\Theta(P)](\rho_C[F^n(STOP)/X]) \\ = & \{\text{definition of } F\} \\ & F(F^n(STOP)) \\ = & F^{n+1}(STOP) \end{array}$$

which establishes the case.

We are now in a position to complete the proof.

$$\begin{array}{ll} & traces([\Theta(\mu\, X.P)]\rho_C) \\ = & \{\Theta(\mu\, X.P) = \mu\, X.\Theta(P)\} \\ & \bigcup_n traces(F^n(STOP)) \\ \leq & \{\text{true for each } F^n(STOP), \text{ so true for their union}\} \\ & \bigcup_{tr \in traces([\mu\, X.P]\rho_A)} \phi(tr) \end{array}$$

Therefore, $\Theta(\mu\, X.P)]\rho_C) \sqsubseteq^T_{imp} [\mu\, X.P]\rho_A$ as required.