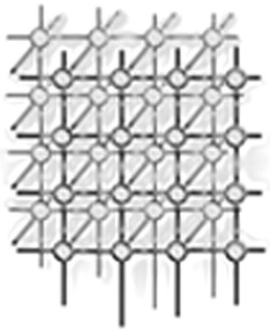
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2008; **1**:1–26 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*



A step towards refining and translating B control annotations to Handel-C

W Ifill[1], S Schneider[2,*]

[1] *AWE Aldermaston, D2, Reading, Berks, RG7 4PR.*
[2] *Department of Computing, University of Surrey, Guildford, England.*

SUMMARY

The design and implementation of critical controllers benefits from development in a formal method such as the B-Method. However, B does not support direct specification of executions, but this is a requirement in controller design. The aim here is to develop a set of annotations so that they can be used by a B design engineer to capture execution requirements while creating the B model. The annotations, once shown to be consistent with the B machine, can be used independently to assess the correctness of proposed CSP controllers. CSP||B is an alternative formal method integration that can be used to develop critical controller with both state and event behaviour. The advantage of using annotations is that the execution requirements can be captured and shown to be consistent with the state during operation development, and that a control loop invariant to establish correctness does not have to be independently developed. Handel-C is used on route to hardware synthesis as it supports the implementation of concurrency and the manipulation of state. Annotations are again used to guide the translation of the B and control annotations into Handel-C.

This work has three main aims. Firstly, to introduce a set of annotations to describe control directives to permit controller development in B. The annotations capture execution requirements. They give rise to proof obligations that when discharged prove that the annotations are consistent with the machine they are written in, and therefore will not cause the machine to diverge. Secondly, we have proven that CSP controllers that are consistent with the annotations will preserve the non-divergence property established between the machine and the annotations. Thirdly, we show how annotation refinement is possible, and show a range of mappings from annotated B and consistent controllers to Handel-C. The development of mappings demonstrates the feasibility of automatic translation of annotated B to Handel-C.

KEY WORDS: *B Method, CSP, annotations, Handel-C.*

1. Introduction

Annotating B-Method specifications with control flow directives enables engineers to describe many aspects of design within a single notation. This paper is concerned with the development of annotations which can be used to capture control flow requirements within B specifications, and then to relate those annotations to controller descriptions written in CSP. Previous work on combining CSP and B [20] verified the CSP controller against the B description directly, but this

*Correspondence to: AWE Aldermaston, D2, Reading, Berks, RG7 4PR.

Received 1 January 2008
Revised 1 January 2008
Accepted 1 January 2008

forces the entire model to be considered at once. The annotations approach presented in this paper allows a separation of the state aspects of the model from the control aspects, while ensuring they are consistent.

We generate proof obligations (pobs) to demonstrate that the set of executions allowable by the annotations of a B [1, 17] machine do not cause operations to diverge. The benefit of this approach is that only the semantics of the machine operations are required in checking the annotations, and these checks are similar in size and difficulty to standard B machine consistency checks. Controllers written in a subset of CSP (definition 2.7), which describe the flow of control explicitly, can be checked against the annotations. There is therefore no need to check the CSP [7, 15, 16] directly against the full B description. Once the annotations are shown to be correct with respect to the B machine we can evaluate CSP controllers against the annotations without further reference to the machine (definitions 2.13 and 2.14). Machines can be refined and implemented directly in accordance with the B-methodology [1] in the normal way while remaining consistent with the controller. In previous work the authors [10] presented the NEXT and FROM annotations, which permitted simple annotated B specifications and controllers to be written. Prior to that the authors [11] presented a route to VHDL [5], a hardware description language. In this paper we present three more annotations: NEXT_SEQ, NEXT_PAR and NEXT_COND and add input and output to the operations. We also present an informal refinement theory for annotations and a route to implementation via Handel-C, which was chosen because of its links to CSP. The refinement theory outlined in this paper allows annotations to be independently refined and remain consistent with a B machine. A detailed description of all the annotations is given in [9].

Previous work obtaining hardware implementations from B approached the problem by using B as a Hardware Description Language (HDL) that translates to VHDL [8, 3]. The approach presented here achieves the goal of obtaining hardware via Handel-C as an intermediate stepping stone, which means that the B that is translated does not require the same degree of HDL structural conformance as does the B for VHDL translation. Approaches that translate HDLs to B for analysis [4] do not directly support development by formal refinement. Event B [12] has been used to support the development of hardware circuits [2] in an approach that includes refinement but not the code generation process. Not only are we working towards code generation, but we wish to work with specifications that model state and control equally strongly. CSP‖B [20, 19] has the capability to model state and event behaviour, but the CSP controller must be instantiated with B components to verify the combination. We break the verification of controllers down into manageable stages, and offer an approach to refinement and translation. Integrations of CSP and Z (CSP-Z) by Mota and Sampaio [13] and CSP and Object Z (CSP-OZ) Fischer [6] require a CSP semantics to be given to Z in order for integration to be analysable as a whole. Our approach differs from other formal language integrations in two ways. Firstly, the control flow behaviour is captured during the development of the state operation in the form of annotations. The annotations are control specifications or directives. Only later is a complete controller developed that satisfies the annotations. In this way the developer of the state operations in B can constrain controller behaviour, but full controller development can be postponed and possibly performed by a different engineer. Secondly, there is no notion of executing the models together and analysing this integration for deadlocks. In this approach the different formal notations provide different views of the system, and both views are required to obtain a executable model.

This paper describes extensions to the work presented in B2007 [10], in particular through the introduction of additional annotations, the incorporation of I/O, and an informal treatment of refinement and translation. We introduce annotations for use within B machine specifications, together with their associated proof obligations. We then introduce a simple controller language, a subset of the CSP process algebraic language [7, 16], and show how controllers written in this language relate to annotations. Finally we introduce an illustrative example, of a controller for the door to a safe, and discuss a translation route to hardware via Handel-C. The Handel-C translations are basic and in a few cases performed in accordance with existing translation work [14, 18].

We restrict our attention in this paper to correct B machines: those for which all B Machine consistency proof obligations have already been discharged. Controllers are considered as *processes* performing *events*, which correspond to operations in the controlled B machine. Hence, operation names will appear in the controller descriptions as well as the B machine definitions. The one-to-one alignment of CSP events and B operations is a constraint imposed by this modelling approach to simplify the linkage between the B and CSP. In practice it is straightforward to introduce additional CSP modelling events, in effect by matching them with a dummy null B operation.

2. The General Framework

The approach proposed in this paper introduces *annotations* on B operations as a mechanism for bridging the gap between B machines and CSP controllers, while maintaining the separation of concerns between the B modelling of state and the CSP modelling of behaviour. The approach consists of the following components:

- **Machine Definition**: the controlled component must first be defined and shown to be machine consistent. Consistency means that its initialisation establishes the invariant of the machine, and each operation preserves it.
- **Annotate Machine**: the initialisation and the operations in the machine definition are annotated with fragments of the requirements for control flow.
- **Establish Machine-Annotation Consistency** Discharge POs between definitions and establish consistency of the annotations with the controlled machine. This means that the fragments of control flow captured by the annotations really are appropriate for the machine.
- **Define controller**: A controller is a CSP process that describes the overall flow of control for the B machine.
- **Establish Annotation-Controller Consistency**: establishing that the controller is consistent with the annotations by showing that every part of the controller is supported by some annotation. Annotation-controller consistency is developed in two phases: firstly initial-consistency and secondly step-consistency
- **Refine and translate**: refinement to a concrete B specification may be required to enable translation to Handel-C. Concrete specifications are essentially those consisting of programming or directly implementable constructs. The translation is the final step and requires additional annotation directives to set type sizes and I/O ports characteristics.

Checking a CSP controller against a machine is thus reduced to checking it against the annotations and verifying that the annotations are annotation consistent with the machine.

The framework presented here is quite general, in that it may be applied to both Event-B and classical B, and new annotations maybe introduced provide the framework is followed. New annotations maybe added along with supporting control operations as required provided that a consistency argument is developed also. In defining the framework the first step to be taken is therefore to fix on the control language and the associated annotations to be incorporated into the B machine descriptions. We will demonstrate the approach to developing the framework with a simple model to illustrate aspects of the approach. The annotation we consider first is the NEXT annotation. An extremely simple controller language consisting only of prefixing, choice, interleaving, if-then-else, and recursion is used to develop the example.

B Machines

The B-Method [1] has evolved two major approaches: classical B and Event-B. Annotations can be used in either classical B machines, or Event-B systems. Classical B approaches focus on the services that a system might provide, whereas Event-B focuses on the events that occur within the system. Classical B is used in this paper.

A generic classical B *MACHINE M*, given below, has variables, invariant, initialisation, and a set of operations $OP1$ through to OPn that have inputs and outputs. v describes a set of inputs and y describes a set of outputs to and from an operation, respectively. The types in the B Machine may be maximal, but during the development to implementation maximal types are replaced with finite implementable subsets.

$$
\begin{array}{l}
MACHINE\ M \\
VARIABLES\ v \\
INVARIANT\ I \\
INITIALISATION\ v :\in u \\
OPERATIONS \\
\qquad y_1 \longleftarrow OP1(z_1) \mathrel{\widehat{=}} P_1 \mid B_1; \\
\qquad y_2 \longleftarrow OP2(z_2) \mathrel{\widehat{=}} G_2 \Longrightarrow B_2; \\
\qquad\qquad \ldots \\
\qquad y_n \longleftarrow OPn(z_n) \mathrel{\widehat{=}} P_n \mid B_n \\
END
\end{array}
$$

The operations are defined in the Guarded Substitution Language (GSL). The B machine is machine consistent (internally consistent) when each operation preserves the machine invariant, I. Every operation must be either guarded, G, or have a precondition, P, but all must have a next annotation (not shown). In Event-B, unlike classical B, new operations can be added during refinement. In the examples we anticipate the need for operations in the later stages of refinement by introducing the signature of the operation with a body defined by the *skip* operation. We adopt pobs for classical B refinement. Proof obligations (pobs) are written in terms of weakest preconditions, and the notation $[S]P$ denotes the weakest precondition for statement S to achieve predicate P.

A key notion in B is refinement. The refinement process adds detail to the specification in a consistent way to realise an implementation. Refinement involves removing non-determinism and adopting concrete types. We add to the concept of B refinement with the annotations, by adding the notion of annotation control flow refinement.

The Annotation with I/O

We annotate operations of B machines with a NEXT annotation that supports operations with I/O. If the conjunction of pobs for all the annotations are discharged then we say that the annotations are consistent with the machine. A consistent controller that evolves in accordance with the *NEXT* annotations steps will not diverge or deadlock. A NEXT annotation on the current operation OP_i (where OP_i represents $y_i \longleftarrow Op_i(z_i)$ and y_i is the output vector, $y_1 \ldots y_n$, and z_i is the input parameter vector, $z_1 \ldots z_m$) introduces another operation OP_j, or set of operations $OP_J = \{\ OP_{j_1}, \ldots, OP_{j_n}\}$, which will all be enabled after OP_i has executed (where an operation in the annotation such as OP_j represents $Op_j(e_j)$ and e_j is the input expression vector, $e_1 \ldots e_m$). In the NEXT annotation e_j is a list of expressions which serves as inputs on which OP_j can be called next. In this paper we will restrict the expressions to variables v defined in the B machines. In

summary, operations depicted in the annotations will be of the form OP_j ($OP_j(v_j)$), and operation definitions in the machine will be of the form $y \longleftarrow Op(z)$. The B input and output variables become ports in the hardware implementation. The value of these variables is not considered when calculating the pobs. Only the type of the variables is checked.

We use I to refer to the invariant of the B machine, T to refer to the machine's initialisation, P_i to refer to the precondition of operation Op_i, and B_i to refer to the body of operation Op_i.

The Basic NEXT Annotation

$OP_i \quad \widehat{=} \quad \textbf{PRE} \quad P_i \quad \textbf{THEN} \quad B_i \ \textbf{END} \quad /*\ OP_J \text{ NEXT } */ \ ;$

Definition 2.1 (Proof Obligations of the Basic NEXT on INITIALISATION) *Given the following B initialisation:*

$$\textbf{INITIALISATION} \quad \text{T } /*\ \{Op_j(v_j)\ \}\ NEXT\ */ \ ;$$
$$\textit{The following pobs arise:} \qquad [T]((v_j \in T_j) \ \wedge \ P_j[v_j/z_j])$$

The NEXT annotation following the initialisation indicates the set of operations that are enabled initially. T is the B initialisation substitution. $[T](P)$ is a predicate in which the substitutions of T are made in P (in the above pob P is $((v_j \in T_j) \ \wedge \ P_j[v_j/z_j])$ where $P_j[v_j/z_j]$ is the predicate in which the free variables z_j are replaced by v_j). The pob states that the initialisation establishes the preconditions of the operations stated in the NEXT annotation, which means that any of the operations introduced by the NEXT annotation may follow the initialisation in execution without causing the machine to diverge.

Although there can be more than one operation in the annotation, the example illustrates only one NEXT operation. The variables used as input parameters in the annotation (v_{j_1} ,..., v_{j_m}) must be of the type required in the operation definition, and T_j cannot be empty by definition [1]

Definition 2.2 (Proof Obligations of the Basic NEXT on Operations) *Given the following B operation:*

$$y_i \longleftarrow Op_i(z_i) \quad \widehat{=} \quad \textbf{PRE} \quad P_i \quad \textbf{THEN} \quad B_i \quad \textbf{END}$$
$$/*\ \{\ Op_j(v_{j_1}), \ldots, Op_{j_n}(v_{j_n})\ \}\ NEXT\ */ \ ;$$

the related pobs follow:

$$(P_i \ \wedge \ I \ \Rightarrow \ [B_i]((v_{j_1} \ \in \ T_{j_1}) \ \wedge \ P_{j_1}[v_{j_1}/z_{j_1}])) \ \wedge$$
$$\ldots$$
$$(P_i \ \wedge \ I \ \Rightarrow \ [B_i]((v_{j_n} \ \in \ T_{j_n}) \ \wedge \ P_{j_n}[v_{j_n}/z_{j_n}]))$$

where the elements of v_i and v_j are free in B_i, P_i, and I

The pob of the NEXT annotation in the operation, Op_i, states that the body, B_i, of the operation establishes the preconditions, $P_{j\,1}$ to $P_{j\,n}$, of the operations in the annotations $Op_{j\,1}$ to $Op_{j\,n}$. This means that to discharge the pob the annotated operation must enable the operations listed in the annotation. In an execution the operations in the annotation may follow the annotated operation.

The NEXT_PAR Annotation

Operations can be annotated to indicate that a parallel execution (NEXT_PAR) is required after the current operation. Two or more sets are introduced (only two illustrated below). Any operation of a respective set can run in parallel with any other operation from any of the other sets.

Definition 2.3 (Proof Obligations of NEXT_PAR) *Given the following B operation:*

$$y_i \longleftarrow Op_i(z_i) \quad \widehat{=} \quad \textbf{PRE} \quad P_i \quad \textbf{THEN} \quad B_i \quad \textbf{END} \\ /*\ OP_{J_1} \ldots OP_{J_n}\ NEXT\_PAR\ */ ;$$

The related pobs follow:

$$(P_i \ \wedge \ I \ \Rightarrow \ [B_i]((v_{j_1} \ \in \ T_{j_1}) \ \wedge \ P_{j_1}[v_{j_1}/z_{j_1}])) \ \wedge$$
$$\ldots$$
$$(P_i \ \wedge \ I \ \Rightarrow \ [B_i]((v_{j_n} \ \in \ T_{j_n}) \ \wedge \ P_{j_n}[v_{j_n}/z_{j_n}])) \ \wedge$$

$$\forall \ i, j \neq i, op_1 \in OP_{J_i}, op_2 \in OP_{J_j} \ \bullet \ variable\_used(op_1) \ \cap \ variable\_used(op_j) = \{\}$$

The parallel annotation offers the option to execute two or more operations in parallel after the current operation, provided they do not set or read any variables in common. The proof obligation ensures that all the operations in the annotations are enabled after the current operation. Only one from each set will be executed in parallel in practice. The $variable\_used(op_i)$ function lists in a set all the variables either read or written in a particular operation op_i. Definition 2.3 requires that the alphabets be disjoint, which is the main difference between NEXT and NEXT_PAR.

The FROM_PAR Annotation

The FROM_PAR annotations is used to indicate that the current operation is part of an interleaving. The operation that precedes the interleaving is given in the annotation. To be annotation consistent (definition 2.13) the NEXT_PAR and the FROM_PAR annotation must agree. In this section the FROM_PAR may not refer back to the initialisation: it may not follow the INITIALISATION.

Definition 2.4 (The FROM_PAR Annotation and Proof Obligation)

$$y_j \longleftarrow Op_j(v_j) \quad \widehat{=} \quad \textbf{PRE} \quad P_j \quad \textbf{THEN} \quad B_j \quad \textbf{END} \\ /*FROM\_PAR\{OP_i\}(V_j)*/;$$

The following pob arises:

$$(P_i \wedge \ I \Rightarrow ([B_i]((v_{j_1} \ \in \ T_{j_1}) \ \wedge \ P_{j_1}[V_j/v_{j_1}])))$$

The FROM_PAR annotation can be used in the same operation as the NEXT annotation. Apart from identifying which operation the current operation can be invoked from as part of an interleaving, it also identifies the input variable list (V_j) used with the invocation.

The NEXT_SEQ Annotation

Operations can be annotated to indicate a requirement for a particular sequential execution: NEXT_SEQ.

Definition 2.5 (Proof Obligations of NEXT_SEQ) *Given the following B operation:*

$$y_i \longleftarrow Op_i(z_i) \quad \widehat{=} \quad \textbf{PRE} \quad P_i \quad \textbf{THEN} \quad B_i \quad \textbf{END} \\ /*\,\{Op_{j_1}(v_{j_1})\}\ ,\ldots,\{\ Op_{j_n}(v_{j_n})\}\ \textit{NEXT\_SEQ}\ */\ ;$$

The related pobs follow:

$$(P_i\ \wedge\ I\ \Rightarrow\ [B_i]((v_{j_1}\ \in\ T_{j_1})\ \wedge\ P_{j_1}[v_{j_1}/z_{j_1}]))\ \wedge \\ \ldots \\ (P_{n-1}\ \wedge\ I\ \Rightarrow\ [B_{n-1}]((v_{j_n}\ \in\ T_{j_n})\ \wedge\ P_{j_n}[v_{j_n}/z_{j_n}]))$$

where the elements of z_i and v_j are free in B_i, P_i, and I

The NEXT_SEQ annotation is conceptually different from the NEXT annotation, because it captures specific paths of executions that must exist in a controller. The current operation Op_i enables $Op_{j_1}(v_{j_1})$ which in turn enables $Op_{j_2}(v_{j_2})$, etc.

The NEXT_COND Annotation

To enable the current operation to conditionally select one set of operations next as opposed to some other set the NEXT_COND annotation is used. The condition NEXT_COND annotation is an extension to the NEXT annotation that supports conditional next path selection.

In definition 2.6, proving the pobs establishes that if the output of the current operation is *true* then all the operations OP_{j_1} through to OP_{j_n} are guaranteed to be available to execute, and if the current operation returns false then the operations OP_{p_1} through to OP_{p_n} are guaranteed to be available to execute.

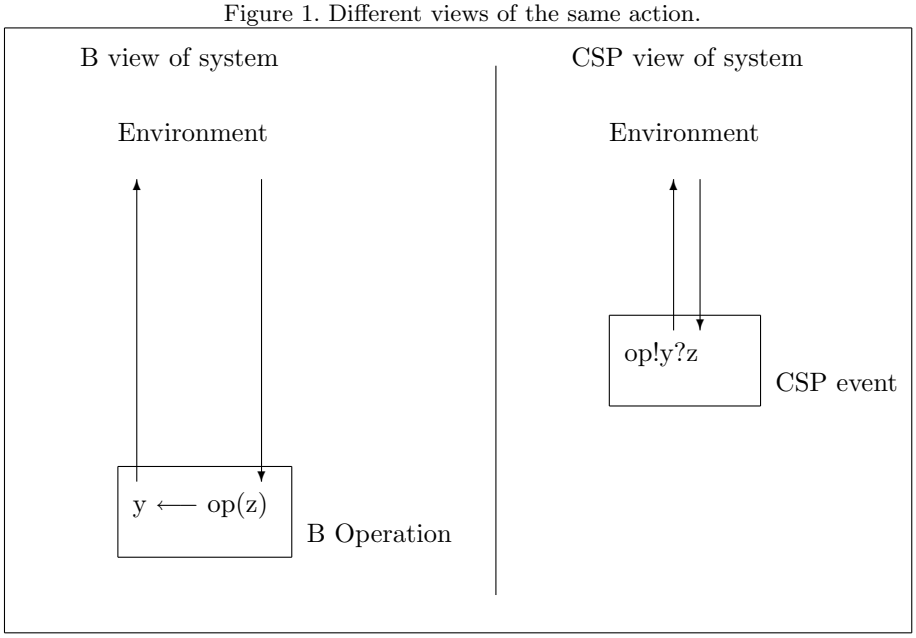
Definition 2.6 (Proof Obligation of NEXT_COND) *Given the following B operation:*

$$y_i \longleftarrow Op_i(z_i) \quad \widehat{=} \quad \textbf{PRE} \quad P_i \quad \textbf{THEN} \quad B_i \quad \textbf{END} \\ /*\,\{\ Op_{j_1}(v_{j_1}),\ldots,Op_{j_n}(v_{j_n})\ \} \\ \{\ Op_{p_1}(v_{p_1}),\ldots,Op_{p_m}(v_{p_m})\ \}\ \textit{NEXT\_COND}\ */\ ;$$

The related pobs follow:

$$(I \wedge P_i \Rightarrow [B_i]((y_i = \ \mathit{TRUE}\ \Rightarrow v_{j_1} \in T_{j_1}) \wedge P_{j_1})) \\ \ldots \\ \wedge\ (I \wedge P_i \Rightarrow [B_i]((y_i = \ \mathit{TRUE}\ \Rightarrow v_{j_n} \in T_{j_n}) \wedge P_{j_n})) \\ \wedge\ (I \wedge P_i \Rightarrow [B_i]((y_i = \ \mathit{FALSE}\ \Rightarrow v_{p_1} \in T_{p_1}) \wedge P_{p_1})) \\ \ldots \\ \wedge\ (I \wedge P_{1\,i} \Rightarrow [B_i]((y_i = \ \mathit{FALSE}\ \Rightarrow v_{p_n} \in T_{p_n}) \wedge P_{p_n}))$$

Figure 1. Different views of the same action.



The lists of the NEXT_COND annotation do not have to be the same size. The operation that carries this annotation must have a single boolean output. The B-Method has the given set *BOOL*, which has two members *TRUE* and *FALSE*. They are used to represent true and false in the logic sense.

A Simple Controller Language

The annotations within a B machine together form a controller requirement for the system. The CSP controller represents a CSP trace refined view of the annotated B system. The annotated B machine does not have the fidelity to clearly portray the necessary control detail that the CSP can: the annotations are not clearly laid out as a set of recursive definitions. The annotations are a set of control requirements, whereas the CSP controller is a specific controller realisation. On translation from B and CSP to Handel-C both the annotated B and the CSP are used to build the Handel-C implementation, hence the need to develop a controller.

A distinction is drawn between operations that respond to external commands and those that are driven internally. A development will begin with a description of a number of operations: things that the system must do when commanded. During the development refinements will introduce internal operations. We distinguish between external and internal operations by marking the external operations with $/*ext*/$ annotations, which are discussed in more detail in the refinement and translation Section 3.

Definition 2.7 gives the control language used in this paper, as a BNF for the CSP subset used in this paper: event prefix, choice, interleaving, if-then-else, and recursion control.

Definition 2.7 (Controller Syntax with I/O)

$$
\begin{array}{rcl}
R & ::= & \Box\limits_{y} a!y?z \rightarrow R \mid \\
 & & R1 \,\Box\, R2 \mid \\
 & & (\,|||\limits_{i=1}^{n} \Box\limits_{y} a_i!y_{a_i}?v_{a_i} \rightarrow \ skip); R \mid \\
 & & \mathit{if}\ y\ \mathit{then}\ R1\ \mathit{else}\ R2 \ \mid \\
 & & S(p)
\end{array}
$$

In this paper the CSP controller is a different view of the annotated B specification. A more complex arrangement arises if the CSP controller is permitted to carry around local state. The simplified view is pictorially represented in Figure 1. The Safe Control System example in Section 3 gives a more concrete illustration of the relationship between events and operations. An annotated B machine output is the same as a CSP controller output. In definition 2.7 S is a process name, p is a process variable, R_n are process terms defined by the BNF category as a whole, the channel a (in the controller fragment $\Box_y\, a!y?z \rightarrow R$) is an operation name with a choice over all possible outputs y: from the controller's point of view, if a is called then any output y should be allowed. The outputs are fresh and modelled as a distributed external choice ranging over the type given in the B (the type is not always given in the controller definition). The channel has an input vector z. To accommodate analysis, finite types are used in the CSP. The same restriction is not required in the B. Hence the CSP representation of the B operation may not be a true representation in terms of input and output: the CSP types may be subsets of the B types. $S(p)$ is a parameterised process variable. The external choice operator chooses between two process $R1 \,\Box\, R2$ and relates to the /* OP_J NEXT*/ annotation that has one set. The interleave operator executes the two or more processes concurrently which will not synchronise on any events. The $\mathit{if} - \mathit{then} - \mathit{else}$ operator makes the decision on y; an output of the e operation. Recursive definitions are given as $S \mathrel{\widehat{=}} R$. In a controller definition, all process variables used are bound by some recursive definition.

A major constraint is enforced on the way controllers can be written. It facilitates translations, but in practice the restriction is not significant. Controllers must start with an initialisation ($R1$), then enter a main loop ($S_i = R_i$). This is summarised in definition 2.8. A controller $CTRL$ has a definition, $R1$, given in definition 2.7, in which all the parameterised process variables are the same, S_i. The definition of S_i is R_i and is also given in definition 2.7. The only recursive calls allowed are to S_i. $S_i = R_i$ is a family of recursive calls.

Definition 2.8 (Controller form)

$$
\begin{array}{rcl}
CTRL & = & R1 \\
S_i & = & R_i
\end{array}
$$

where

- $R1$ *and* R_i *are category terms from the BNF definition 2.7;*
- S_i *are process name and are the only recursive calls allowed;*
- R_i *is guarded as defined in definition 2.11.*

The results presented in this paper require that all recursive definitions are *guarded*, which means that at least one event must occur before a recursive call. The meaning of consistency between the controller and the annotations is given in terms of the *init* functions. The *init* (the *init* function here is semantically equivalent to the CSP *init*) function returns a set of operations available next and is developed in definition 2.9. The function is required to extract the event of an action prefix so that it can be compared with the annotation.

Definition 2.9 (init on CSP controller process with I/O extensions)

$$\begin{aligned} init(\underset{y}{\Box}\, a!y?z \rightarrow R1) &= \{a\} \\ init(R1 \Box\ R2) &= init(R1)\ \cup\ init(R2) \\ init((\ \underset{i=1}{\overset{n}{|||}}\ \underset{y}{\Box}\, a_i!y_{a_i}?v_{a_i} \rightarrow\ skip); R) &= \{a_i \mid i = 1...n\} \\ init(if\ y\ then\ R1\ else\ R2) &= init(R1) \cup init(R2) \\ init(S(p)) &= init(R(p)) \end{aligned}$$

There is a requirement for a specialised *init* projecting function, $init - par$, which arise as a result of the need to ensure that an action prefix to an interleaving is correctly annotated.

Definition 2.10 (*init* for CSP Controller Process Fragments with Interleaving)

$$\begin{aligned} init - par(\underset{y}{\Box}\, a!y?v \rightarrow R1) &= \{\} \\ init - par(R1 \ \Box\ R2) &= init - par(R1)\ \cup\ init - par(R2) \\ init - par((\ \underset{i=1}{\overset{n}{|||}}\ \underset{y}{\Box}\, a_i!y_{a_i}?v_{a_i} \rightarrow\ skip); R) &= \{a_i \mid i = 1...n\} \\ init - par(if\ y\ then\ R1\ else\ R2) &= init - par(R1) \cup init - par(R2) \\ init - par(S(p)) &= init - par(R(p)) \end{aligned}$$

An action prefix must appear with output on the left. In the first case of the *init* definition the head of the control fragment is extracted. The outputs and inputs of the action are the same as the outputs and inputs of the B operation. The *init* of a prefixed action is the action (event). The *init* of a choice between two processes is the union of the *init* of the individual processes. The *init* of the interleaving is the set of first actions of each interleaving. Annotations clearly show an ordering of operations: an initial operation and a set of next operations. Every operation has a prefix, and is therefore *guarded*. Every control fragment must have a prefix and hence be guarded. The *guard* function is defined in definition 2.11. Prefixed operations are *guarded*. A fragment with an external choice separating the two processes is prefixed if the individual processes are *guarded*. Similarly with the if-then-else. The parameterised process variable is not *guarded*, whereas the recursive definition is guarded if the body is *guarded*.

Definition 2.11 (guarded on CSP controller process with I/O)

$$\begin{aligned} guarded(\underset{y}{\Box}\, a!y?z \rightarrow R1) &= true \\ guarded(R1 \ \Box\ R2) &= guarded(R1) \wedge\ guarded(R2) \\ guarded((\ \underset{i=1}{\overset{n}{|||}}\ \underset{y}{\Box}\, a_i!y_{a_i}?v_{a_i} \rightarrow\ skip); R) &= true \\ guarded(if\ TRUE\ then\ R1\ else\ R2) &= guarded(R1) \wedge\ guarded(R2) \\ guarded(if\ FALSE\ then\ R1\ else\ R2) &= guarded(R1) \wedge\ guarded(R2) \\ guarded(S(p)) &= false \end{aligned}$$

We introduce a function to extract the set of operations associated with a NEXT annotations in an operation comments to help define consistency.

Definition 2.12 (NEXT Annotation Listing)

$$next(OP_i) \quad = \{OP_j, ..., OP_k\} \quad if\ \ OP_i \triangleq (P_i \mid B_i) /*\ \{OP_j, ..., OP_k\}\ NEXT\ */;$$

$$undefined\ otherwise$$

The *next* listing function defined in 2.12 lists the operations that are enabled by the current operation. The *init* and *guard* functions are applied to fragments of controllers or operations. The dual nature of the *next* function arises because operations and actions correspond in the different models. Hence, $next(a)$ and $next(a \ \widehat{=} \ (P_a \mid B_a)/*X \ NEXT*/)$ produce the same result. If the *next* function is applied to an action then the action is unconditionally converted into its equivalent B operation. This holds true of all the listing functions defined in future chapters.

I/O NEXT Consistency

Consistency between a *guarded* controller and the annotated B machine is broken down into initial (definition 2.13) and step-consistency (definition 2.14).

Definition 2.13 (Initial-Consistency of M with respect to M_CTRL) *The initial-consistency of the controller fragment R is defined as follows:*

1. $\Box_y \ a!y?z \rightarrow R$

   is initially-consistent with M if $a \in$ *next(INITIALISATION) and*
   R is step-consistent with M

2. $R1 \ \Box \ R2$

   is initially-consistent with M if $R1$ *and* $R2$ *are initially-consistent with M.*

3. $S(p)$

   is initially-consistent with M

   where:

   (a) *A family of recursive definitions* $S \ \widehat{=} \ R$ *is initially-consistent with M's annotations if each R is initially-consistent with M's annotations.*

A controller that starts with an interleaving or a conditional control fragment is not initially-consistent and should therefore be avoided. An initialisation cannot have an output which rules out the use of an $if - then - else$ annotation on the initialisation. Ruling out the *interleaving* annotation simplifies initial-consistency checking.

Definition 2.14 (Step-Consistency of M with respect to M_CTRL) *The step-consistency of the controller fragment R is defined as follows:*

1. $\Box_y \ a!y?v \ \rightarrow \ R$

   is step-consistent with Machine M if $(\forall \, b \in \ init(R) \bullet \ b \in next(y \longleftarrow \ a(v)))$ *or*
   $(\forall \, b \ \in \ init - par(R) \bullet b \in next - par(y \longleftarrow \ a(v)))$, *and R is step-consistent with M.*

2. $R1 \ \Box \ R2$

   is step-consistent with Machine M *if* $R1$ *and* $R2$ *are step-consistent with* M*.*

3. *if* y *then* $R1$ *else* $R2$

   is $step-consistent$ *with Machine* M *if* $y \in BOOL$ *and* $R1$ *and* $R2$ *are* $step-consistent$ *with* M *and*
   $\forall \ b \ \in \ init \ (R1) \ \bullet \ b \ \in \ condition\_true(a!y?v)$ *and*
   $\forall \ c \ \in \ init \ (R2) \ \bullet \ c \ \in \ condition\_false(a!y?v)$

4. $(\ |||_{i=1}^{n} \Box_y \ a_i!y_{a_i}?v_{a_i} \rightarrow \ skip); R$

   is step-consistent with Machine M *if*
   $\exists \, v \bullet \forall \ x \bullet v = from - par(a_x)$ *and*
   $\forall \ op \ \in \ \{a_1, ..., a_n\} \ \bullet \ init(R) \subseteq \ next(op)$ *and* R *is step-consistent with* M

5. $S(p)$ *is step-consistent with Machine* M

   where:

   (a) *A family of recursive definitions* $S \mathrel{\widehat{=}} R$ *is step-consistent with* M*'s annotations if each* R *is step-consistent with* M*'s annotations.*

   (b) *We define* $condition\_true(a)$ *as the true set of operations in the* NEXT-CONDITION *annotation of* a*.*

   (c) *We define* $condition\_false(a)$ *as the false set of operations in the* NEXT-CONDITION *annotation of* a*.*

   (d) *We define* $next - par(a)$ *as the set of operations in the* NEXT-PAR *annotation of* a*.*

   (e) *We define* $from - par(a)$ *as the set of operations in the* FROM-PAR *annotation of* a*.*

The interleaving operator can only be shown to be consistent in a very limited sense. Two actions are allowed to occur in parallel provided they do not attempt to change the variables used by the other action.

Definition 2.15 (Consistency) *A controller* R *is* consistent *with the annotations of machine* M *if it is step-consistent with* M*'s annotations and initially-consistent with* M*'s annotations.*

The main result of this section is that if R is consistent with the annotations of a machine M, and the annotations of M are consistent with machine M, then operations of M called in accordance with the control flow of R will never be called outside their preconditions. The proof of this is given in [9].

The annotations are expressive enough to permit a large set of possible consistent controllers. As such the controller is viewed as a trace refinement of the annotations. The controllers do not refine the annotations in a failures divergence sense, so liveness behaviour may not be preserved in refinement.

Table I. NEXT Refinements - Reduction of Non-determinism.

	Annotation	Refinement	type
1	$OP_i \mathrel{\widehat{=}} ...OP_J\ NEXT$	$OP_i \mathrel{\widehat{=}} ...OP'_J\ NEXT$	next external choice refinement
2	$OP_i \mathrel{\widehat{=}} ...OP_J\ OP_K\ NEXT\_PAR$	$OP_i \mathrel{\widehat{=}} ...OP'_J\ OP'_K\ NEXT\_PAR$	next interleave refinement
	$OP_{j\,1} \mathrel{\widehat{=}} ...OP_X\ NEXT$ ... $OP_{j\,n} \mathrel{\widehat{=}} ...OP_X\ NEXT$	$OP_{j\,1} \mathrel{\widehat{=}} ...OP_X\ NEXT$ ... $OP_{j\,n} \mathrel{\widehat{=}} ...OP_X\ NEXT$	
	$OP_{k\,1} \mathrel{\widehat{=}} ...OP_X\ NEXT$ ... $OP_{k\,n} \mathrel{\widehat{=}} ...OP_X\ NEXT$	$OP_{k\,1} \mathrel{\widehat{=}} ...OP_X\ NEXT$ ... $OP_{k\,n} \mathrel{\widehat{=}} ...OP_X\ NEXT$	
		where $OP'_J \subseteq OP_J$ *and* $OP'_K \subseteq OP_K$	

The key feature of the proof of this main result is an argument that no trace of R leads to an operation of M called outside its precondition or guard. This is established by building up the traces of R and showing that at each step an operation called outside its precondition cannot be introduced, by appealing to the relevant annotation and applying its proof obligation.

The benefit of this main result is that the details of the operations of M are required only for checking the consistency of the annotations, and are not considered directly in conjunction with the controller. The annotations are then checked against the controller using the definition of consistency above. This enables a separation of concerns, treating the annotations as an abstraction of the B machine.

3. Refinement and Translation to Handel-C

Refining should be considered where an otherwise cumbersome translation would result. Refining away unwanted behaviour before translation makes the translation easier and less cumbersome. Narrowing down the choice of the next operation reduces the size of the implementation, and avoids the translation process making an arbitrary choice to resolve the choice in the annotations. The first set of refinements, given in Table I replaces annotated sets with their subsets: non-determinism is reduced. The operation references, like OP_J, quoted in the tables are all sets.

NEXT external choice refinement reduces the non-determinism in the choices that are offered in the next step. The NEXT interleave refinement reduces the non-determinism in one or more branches of the interleave execution. Although not shown, the NEXT sequential refinement reduces the non-determinism in one or more sections of the sequence. The NEXT conditional refinement reduces choice in a similar way. The selection criteria concerning which parts of behaviour to remove with annotation refinement are for the design engineer to make. We are only offering a framework for the engineer to make such design decisions.

Table II. NEXT Refinements - Structural Refinements.

	Annotation	Refinement	type
1	$OP_i \mathrel{\widehat{=}} ...OP_X\ NEXT$	$OP_i \mathrel{\widehat{=}} ...OP_J\ OP_X\ NEXT\_SEQ$ $OP_{j1} \mathrel{\widehat{=}} ...OP_X\ NEXT$ $OP_{jn} \mathrel{\widehat{=}} ...OP_X\ NEXT$	introduction of new operation
2	$OP_i \mathrel{\widehat{=}} ...OP_J\ OP_P\ NEXT\_SEQ$ $OP_{j1} \mathrel{\widehat{=}} ...OP_P\ NEXT$ ... $OP_{jn} \mathrel{\widehat{=}} ...OP_P\ NEXT$ $variable\_used(\{OP_j, \ldots, OP_k\})$ $\cap$ $variable\_used(\{Op_p, \ldots, OP_q\})$ $= \{\}$	$OP_i \mathrel{\widehat{=}} ...OP_J OP_P NEXT\_PAR$ $OP_{j1} \mathrel{\widehat{=}} ...OP_P\ NEXT$ ... $OP_{jn} \mathrel{\widehat{=}} ...OP_P\ NEXT$	next sequence to interleave refinement

The second refinement table II outlines some structural refinements, which are more involved than refinements that just remove nondeterminism. Structural refinement allow annotation types to be changed, and they can be made to take advantage of particular target machine implementation architectures. For example, if the execution speed is important and the target implementation system is able to take advantage of parallel operations, then there may be an advantage to replacing sequential annotations with parallel annotations. In case 1 a new set of operations are introduced OP_J. New operations can be introduced into Event-B in subsequent refinements. In classical B *new* operations must be introduced beforehand as operators that implement skip. Case 1 refines a simple NEXT operation into a sequence of detailed operations. The refinement sequence must end in the original next operation, which signifies the end of the refinement chain. In case 2 a next sequence NEXT_SEQ to next interleave refinement NEXT_PAR is depicted. It is possible if the operations that would make up the sequence are independent: they neither read nor write to similar variables.

A translations guide for annotations is given in table III . This is a guide because without the knowledge of the control structure, in particular the points of recursion, a translation can not be automated. However, the annotations do differentiate between internal and external B operations, which has an impact on the final structure of the code. The CSP controller is required to get a full picture for translation and Table VI and to some extent Table IV illustrates how translation of the control can proceed. As mentioned, the translation of a particular annotated operator is dependent on whether the operation is an internal or external operation. Internal operations can execute immediately after invocation. The execution of an external operation must wait for external stimulus: a change in the command input bus. A wait loop is introduced to poll the appropriate input bus until an external operation invocation is detected: *wait_on_*.... Some annotated operators have restrictions on their I/O mode. External operators are marked with $/*\ ext\ */$. The NEXT_PAR can only be associated with internal operations next. The NEXT_SEQ must have an external operator at the head of the sequence and internal operations following. This restriction relates to the way this annotation is used in refinement. The CSP controller does not

Table III. NEXT Annotation Translation Guide.

	Annotation	Handel-C Translation Fragment	Comment
1	$OP_i \mathrel{\widehat{=}} ...\{OP_{j1}\}NEXT$ $op_i!y_i?z_i \rightarrow (op_{j1}!y_{j1}?z_{j1} \rightarrow \ldots$	$y_i = OP_i(v_i)$; $y_{j1} = OP_{j1}(v_{j1})$	internal single next translation
2	$OP_i \mathrel{\widehat{=}} ...\{OP_{j1}\}NEXT$ $/*\ ext\ */ OP_{j1} \mathrel{\widehat{=}} ...$ $op_i!y_i?z_i \rightarrow (op_{j1}!y_{j1}?z_{j1} \rightarrow \ldots$	$y_i = OP_i(v_i)$; $wait\_on\_OP_{j1}$; $if\ in = OP_{j1}$ $then\ y_{j1} = OP_{j1}(v_{j1})\}$ $else\ delay$;	external single next translation
3	$/*\ ext\ */ OP_i \mathrel{\widehat{=}} \ldots$ $\{OP_{j1}, \ldots, OP_{jn}\}NEXT$ $op_i!y_i?z_i \rightarrow (op_{j1}!y_{j1}?z_{j1} \rightarrow \ldots \Box \ldots$ $\Box$ $op_{jn}!y_{jn}?z_{jn} \rightarrow \ldots)$	$y_i = OP_i(z_i)$; $wait\_on\_OP_{j1}\_\ldots\_OP_{jn}$; $if\ in = OP_{j1}$ $then\ y_{j1} = OP_{j1}(v_{j1})$ $else\ \ldots$ $\ldots$ $if\ in = OP_{jn}$ $then\ y_{jn} = OP_{jn}(v_{jn})$ $else\ skip$	external multiple next choice translation
4	$OP_i \mathrel{\widehat{=}} ...OP_j\ OP_k\ NEXT\_PAR$ $OP_j \mathrel{\widehat{=}} ...OP_X\ NEXT$ $OP_k \mathrel{\widehat{=}} ...OP_X\ NEXT$ $op_i!y_i?z_i \rightarrow (op_j!y_j?z_j \rightarrow \ldots) \parallel$ $(op_k!y_k?z_k \rightarrow \ldots)$	$seq\{y_i = OP_i(v_i),$ $par\{y_j = OP_j(v_j),$ $y_k = OP_k(v_k)$ $\}$ $\}$	internal next interleave translation

differentiate between internal and external operations. Hence Tables III and IV are required to obtain a translation.

In Table III a NEXT annotation with one next operation translates to a sequence of two operations. If the second operation is an internal operation then it is case 1: all its inputs are not ported. If the second operation is an external operation (all inputs are ported) then case 2 is the translation template. The controller will wait until a new command arrives then execute the external operation if it was requested. Case 3, sequential arrangement of external operations, is restricted to external operations only. A translation of a sequence that starts with one operation then has a choice of several external operations will test each input set and execute the first operation for which the input has change since its last execution. (The new input values must be latched in.) Interleave action is only permitted between internal operations (case 4): those that take their input from internal variables. The Handel-C *par* statement ensures that all the branches when complete wait until the longest (in terms of clock cycles) has completed. The conditional operator can be used for internal or external action. Details of the NEXT_SEQ and NEXT_COND are given in [9].

Table IV. Existing CSP to Handel-C Translation Guide.

Feature	CSPM	Handel-C	PS	SS
Channel Declarations (from use)	channel	chan, chanin, chanout	✓	
Channel Declarations	channel c	chan SYNC c;		✓
Typed Structured Channel Declarations	channel d : T.T	chan struct d_DATA d		✓
Input Channel Operations	in?x	in?x;	✓	✓
Output Channel Operations	out!x	out!x;	✓	✓
Integer Declarations		int 8 x;	✓	✓
Parametrisable functions	p(n) = ...	void(n)...	✓	✓
External Choice	[ ]	prialt ...	✓	✓
Synchronous Parallel	[| { | ... | } |]	par ...	✓	✓
Replicated Sharing Parallel	[| *Event* |] n: { i..j }•P(n)	par (n=i; n¡=j; ++n)P(n);		✓
Recursion	P = ... → P	while(1) ...	✓	✓
Conditional Choice	if b then P else Q	if (B) then P(); else Q();		✓
Macros	{- ... -}	...	✓	

The translations of Stepney [18], and Phillips and Stilles [14] are given in table IV. Only the translation of parametrisable integer declaration, functions, and recursion are used. This is because our source is not CSP (it is annotated B and CSP)and as such channels are not being used to synchronise events, but to call operations. In the table the CSP language construct and translation are mapped. A tick is inserted if they are supported by Stepney [18] or Phillips and Stilles [14]. When an operation is invoked it takes its input from a port in the environment. Internal synchronisation of operations within machines is not dealt with in this paper. To guide the B translation Table V has been developed. A discussion of the example is given in the next section.

Example: Safe Control System

We use the example of a safe locking system to illustrate the ideas introduced in the previous sections. The abstract specification outlines the operations of the environment. The operations that are invoked by the environment are indicated with $/*ext*/$ annotations. Both the operation output and the operation can be marked with $/*ext*/$ annotations. All $/*ext*/$ annotation outputs are ported and become part of the Handel-C interface output. All $/*ext*/$ operations are associated with a bus port that has a state of the same name as the operation. Variables intended as input are marked with $/*IN*/$. It is possible to mark the variables as $/*IN*/$ or $/*OUT*/$. Along with the mode the width of the type is given in bits. Operations are invoked in two ways. The first way has already been introduced; an $/*ext*/$ operation will have a input bus associated with it, which when set to the operator name will invoke the operation when it is enabled by the control flow. Operations not labelled with $/*ext*/$ are internal and are invoked immediately when enabled by the control flow.

Figure 2. Safe Machine

MACHINE *Safe*

SETS *COMMAND* = { *Locked* , *Unlocked* , *BrokenOpen* }/*2*/

VARIABLES *Door*

INVARIANT *Door* ∈ COMMAND /*OUT2*/

INITIALISATION *Door* := *Unlocked* /* { Lock } NEXT */

OPERATIONS

/*ext*/*Status* ⟵ /*ext*/ **Lock** ≙

PRE *Door* = *Unlocked* **THEN** *Door* := *Locked* ‖ *Status* := *Locked* **END**
/* { Unlock, BreakOpen } NEXT */ ;

UnlockR1 (Comb1a,Comb1b) ≙
PRE *Comb1a* ∈ NAT ∧ *Comb1b* ∈ NAT ∧ *Door* = *Locked* **THEN** skip **END** ;

UnlockR2(Comb2a,Comb2b) ≙
PRE *Comb2a* ∈ NAT ∧ *Comb2b* ∈ NAT ∧ *Door* = *Locked* **THEN** skip **END** ;

/*ext*/ *Status* ⟵ /*ext*/ **Unlock** ≙
PRE *Door* = *Locked*
THEN
ANY *dd* **WHERE** *dd* : COMMAND - { *BrokenOpen* }
THEN
IF (*Unlocked* = *dd*) **THEN** *Status* := *1* **ELSE** *Status* := 0 **END** ‖
Door := *dd*
END
END /* { Lock } { UnLock,BreakOpen } NEXT_COND */ ;

/*ext*/ *Alarm* ⟵ /*ext*/ **BreakOpen** ≙
BEGIN *Door* := *BrokenOpen* ‖ *Alarm* := *1* **END**
/* { BreakOpen } NEXT */ ;
END

Figure 3. Safe Machine Controller.

$$
\begin{aligned}
CTRL &= Initialisation \rightarrow \Box_{y}\, Lock!y \rightarrow S \\
S &= (\Box_{y}\, Unlock!y \rightarrow (if\ y\ then\ \Box_{y}\, Lock!y \rightarrow CTRL\ else\ S))\Box \\
&\quad (\Box_{y}\, BreakOpen!y \rightarrow B\_CTRL) \\
B\_CTRL &= \Box_{y}\, BreakOpen!y \rightarrow B\_CTRL
\end{aligned}
$$

The Example's State and Control Flow

In Figure 2 the B Abstract Machine for the safe is given. There are three command states: *Locked*, *Unlocked*, and *BrokenOpen* which are represented in two bits (the /*2*/ annotation allocates 2 bits for the storage of the enumerated type). The variable *Door* is drawn from the *COMMAND* type and initialised to *Unlocked*. The **Lock** operation is enabled after initialisation. It is an external operation with externally ported output. After setting the *Door* state variable to *Locked*, **Unlocked** and **BreakOpen** are enabled. For completeness we introduce two operations that will be used later to develop the detailed functionality of the machine during refinement. These operations are **UnlockR1** and **UnlockR2**. Their bodies are not expanded. New operations can not be introduced during refinement in classical B hence the reason for their introduction now. The B method requires explicit typing of the inputs, though the type of the output is deduced so is not given explicitly. The **Unlock** is an external operation and has externally ported output. It non-deterministically decides to set the *Door* variable to *Unlocked* or *Locked*. The next operator to be enabled depends on the outcome of the **Unlock** operation. If *Unlocked* is chosen then the next enabled operation is **Lock**, otherwise **Unlocked** or **BreakOpen** will be offered. The **BreakOpen** operation sets the *Door* state to *BrokenOpen* and offers itself as the next operation available.

The controller *CTRL*, given in Figure 3, first performs a *Initialisation* then a *Lock* and then jumps to the *S* process where it can perform either an *Unlock* or *BreakOpen*. The *Unlock* event has a single output that is used as the conditional test in the if-then-else following the *Unlock* event. If the output of the *Unlock* operation is true then the flow of control returns to *CTRL*, otherwise control returns to *S*.

A Refined Example

A refinement of the *Safe* machine, called SafeR, is given in Figure 4 and Figure 5. It is a classical B refinement mimicking a refinement in Event-B, though it is not in itself an Event B refinement since the new events are not guarded to control their occurrence. The operation *UnlockR*1 and *UnlockR*2 are introduced to refine *Unlock*. The *Safe* REFINEMENT, *SafeR*, breaks down the Unlocking process into two stages. Firstly, two new operation are slotted into the control in parallel: $UnlockR1(Comb1a, Comb1b)$ and $UnlockR2(Comb2a, Comb2b)$. Both have a combination parameter which is compared against a stored master code and a secondly parameter that is used to create a new master key. The *UnlockR* commands update the master combination if a successful comparison occurs. New input variables are added: $Cx1a$, $Cx2a$, $Cx1b$ and $Cx2b$. These are used to input the combination values and are not used by the B Operations. $Checked1$, $Checked2$, $Master1$ and $Master2$ are new variables used by the operations. The annotations of the *Lock* operation are refined. Two operation are added before the *Unlock*. The extra proof

Figure 4. Safe Refinement Part 1.

REFINEMENT *SafeR*
REFINES *Safe*
VARIABLES *Door*, *Cx1a*, *Cx2a*, *Cx1b*, *Cx2b*,
Master1, *Checked1* *Master2*, *Checked2*
INVARIANT
Cx1a ∈ NAT/*IN16*/ ∧ *Cx2a* ∈ NAT/*IN16*/ ∧
Cx1b ∈ NAT/*IN16*/ ∧ *Cx2b* ∈ NAT/*IN16*/ ∧
Master1 ∈ NAT/*16*/ ∧ *Checked1* ∈ NAT/*1*/ ∧
Master2 ∈ NAT/*16*/ ∧ *Checked2* ∈ NAT/*1*/
INITIALISATION
Door:=Unlocked ‖ *Cx1a*:=0 ‖ *Cx2a*:=0 ‖ *Cx1b*:=0 ‖ *Cx2b*:=0 ‖
Master1:=67 ‖ *Checked1*:=0 ‖ *Master2*:=76 ‖ *Checked2*:=0 /* { Lock } NEXT */
OPERATIONS

/*ext2*/ Status ⟵ /*ext1*/ **Lock** ≙
PRE
Door = *Unlocked*
THEN
Door := *Locked* ‖ *Status* := *Locked* ‖ *Checked1* := 0 ‖ *Checked2* := 0
END
/* { UnlockR1(Cx1a,Cx1b), UnlockR2(Cx2a,Cx2b) } { Unlock } NEXT_SEQ */
/* { UnlockR1(Cx1a,Cx1b) } { UnlockR2(Cx2a,Cx2b) } NEXT_PAR */ ;

/*ext1*/**UnlockR1(/*16*/Comb1a,/*16*/Comb1b)** ≙
PRE
Comb1a ∈ NAT ∧ *Comb1b* ∈ NAT ∧ *Door* = Locked
THEN
IF
(Comb1a = Master1)
THEN
Checked1 := 1 ‖ *Master1* := Comb1b
ELSE
Checked1 := 0
END
END /* { Unlock } NEXT */ /* FROM-PAR {Lock} */;

Figure 5. Safe Refinement Part 2

/*ext1*/**UnlockR2(/*16*/Comb2a,/*16*/Comb2b)** $\widehat{=}$
PRE
$Comb2a \in \text{NAT} \wedge Comb2b \in \text{NAT} \wedge Door = \text{Locked}$
THEN
IF
$(Comb2a = Master2)$
THEN
$Checked2 := 1 \parallel Master2 := \text{Comb2b}$
ELSE
$Checked2 := 0$
END
END /* { Unlock } NEXT */ /* FROM-PAR {Lock} */**;**

/*ext2*/**Status** $\longleftarrow$ **Unlock** $\widehat{=}$
PRE
$Door = Locked$
THEN
IF $(Checked1 = 1) \wedge (Checked2 = 1)$
THEN
$Door := Unlocked \parallel Status := 1$
ELSE
$Door := Locked \parallel Status := 0$
END
END
END /* { Lock } { UnlockR } COND_NEXT */ **;**

/*ext*/ **Alarm** $\longleftarrow$ /*ext*/ **BreakOpen** $\widehat{=}$
PRE $Door \in \text{COMMAND}$ **THEN** $Door := BrokenOpen \parallel Alarm := 1$ **END**
/* { BreakOpen } NEXT */
END

Figure 6. Refined Safe Controller.

$$
\begin{aligned}
CTRL &= Initialisation \rightarrow \Box_y\, Lock!y \rightarrow S \\
S &= (UnlockR1?Cx1a?Cx1b \rightarrow skip \;|||\; UnlockR2?Cx2a?Cx2b \rightarrow skip) \rightarrow \\
&\quad \Box_y\, Unlock!y \rightarrow (if\ y\ then\ \Box_y\, Lock!y \rightarrow S\ else\ S)
\end{aligned}
$$

Figure 7. SafeR Translation Part 1a.

```
//set clock = external "Clock";
#define PAL_TARGET_CLOCK_RATE 25175000
#include "pal_master.hch"       ///////////////
// BreakOPen removed in translation as
// not used and no command default added
typedef enum {Not_Commanded =
  (unsigned 2) 0, Locked, Unlocked} COMMAND; //
typedef enum {No_Command =
  (unsigned 2) 0, Lock, UnlockR1, UnlockR2} SafeR;
unsigned 2 Door;      // B variables
unsigned 1 Checked1;   //
unsigned 16 Master1;   //
unsigned 1 Checked2;   //
unsigned 16 Master2;   //
SafeR SafeR_Bus_var;   // latch input bus values to
                       // request operation execution
unsigned 1 Status_Unlock;   // operation output values
unsigned 2 Status_Lock;     //
interface bus_in(unsigned 16 inp) Cx1a();   // IN annotations
interface bus_in(unsigned 16 inp) Cx2a();   //
interface bus_in(unsigned 16 inp) Cx1b();   // IN annotations
interface bus_in(unsigned 16 inp) Cx2b();   //
interface bus_in(SafeR inp) SafeR_Bus();      // ext operations
interface bus_out() Door1 (unsigned 2 OutPort=Door);  // OUT an.
interface bus_out()
  Status_Unlock1 (unsigned 1 OutPort=Status_Unlock); //
```

obligations can be discharged. The bodies of the *UnlockR* and *Rekey*(*Comb*2) are completed at this level. The body of the *Unlock* operation is refined. The annotations of the *Unlock* are refined: the *BreakOpen* operation is removed as an option. What was one unlock operation has been expanded into three (two in parallel). Before refinement the *Unlock* operation has both input and output. The refined version has the input occurring on the first operations in the refined sequence of operations (*UnlockR*1 and *UnlockR*2), and the output occurring on the final operation of the sequence (the original *Unlock* operation).

The controller given in Figure 6 starts off like the abstract process with an *Initialisation* and a *Lock* then a jump to S. There is in this refined process no choice to *breakOpen*, only *UnlockR*1and *UnlockR*2 are offered with $Cx1a$, $Cx1b$, $Cx2a$ and $Cx2b$respectively offered as input. The *UnlockR* operation is the first in a sequence of operations that refines the original *UnLock* operation. The refined sequence starts with a parallel combination of the *UnlockR*1 and the *UnlockR*2 events then the original *Unlock* event, at which point the output is given. Both legs of the interleaving must terminate before control is passed to the *Unlock*. As before the outcome of *Unlock* determines what happens next. If the *Unlock* was successful the process will be restarted from the beginning. If the current attempt at locking failed then another go at *Unlock* will occur. It is noted that $Lock \rightarrow S$ could have been replaced by *CTRL*. However, the former is easier to translate given the framework provided.

Figure 8. SafeR Translation Part 1b.

```
void wait_on_Lock_fnc (){
    while (SafeR_Bus.inp != Lock){delay;}
    SafeR_Bus_var = Lock;
}
unsigned 2 Lock_fnc(void){
    par{
       Door = Locked;
       Status_Lock = Locked;
       Checked1 := 0;
       Checked2 := 0;
       }
    return Status_Lock;
}
void wait_on_UnlockR1_fnc(void){
    while (SafeR_Bus.inp != UnlockR1){delay;}
    SafeR_Bus_var = UnlockR1;
}
void UnlockR1_fnc(unsigned 16 Comb1a, unsigned 16 Comb1b){
    if (Comb1a == Master1) {
       par{Checked1 = 1; Master1 = Comb1b;}
    }
    else
       {Checked1 = 0;}
}
void wait_on_UnlockR2_fnc(void){
    while (SafeR_Bus.inp != UnlockR2){delay;}
    SafeR_Bus_var = UnlockR2;
}
void UnlockR2_fnc(unsigned 16 Comb2a, unsigned 16 Comb2b){
    if (Comb2a == Master2) {
       par{Checked2 = 1; Master2 = Comb2b;}
    }
    else
       {Checked2 = 0;}
}
```

A Hand Translation using the Guidance

A summary of the hand translations on the refined B specification is given in Table V. The B provides the details of the types, variables, and functions. The CSP controller provides the execution details that are used later to construct the Handel-C main section. A summary of the hand translation of the controller is given in table VI.

First we review the B translation. The *SETS* clause is translated into an enumerated type. The *INVARIANT* section is used to create the declarations. Variables annotated with a mode will be created as buses of the appropriate I/O type and size. Other variables will be created. Variables which will be bound to ports are created. Each operation which is external is associated with a

Figure 9. SafeR Translation Part 2.

```
unsigned 1 Unlock_fnc(void){
    par{
        if ((Checked1 = 1) & (Checked2 = 1)){
            par{Door=Unlocked; Status_Unlock=1;}
        }
        else {par{Door=Locked; Status_Unlock=0;}}
    }
    return Status_Unlock;
}
void Initialisation_fnc(void){
    Checked1 = 0;Master1  = 67;
    Checked2 = 0;Master2  = 76;Door = Unlocked; // INITIALISATION
    Status_Lock = 0;Status_Unlock = 0;        // SET OUTPUT DEFAULT
}
void CTRL_fnc(void){
    Initialisation_fnc(); wait_on_Lock_fnc();
    if (SafeR_Bus_var == Lock){Lock_fnc();}else{delay;}
}
void S_fnc(void){
    while(1){par{
                seq{wait_on_UnlockR1_fnc();
                   if (SafeR_Bus_var==UnlockR1){
                      UnlockR1_fnc(Cx1a.inp,Cx1b.inp);}
                   else
                      {delay;}
                } // seq
                seq{wait_on_UnlockR2_fnc();
                   if (SafeR_Bus_var==UnlockR2){
                      UnlockR_2fnc(Cx2a.inp,Cx2b.inp);}
                   else
                      {delay;}
                } // seq
             } // par
             Status_Unlock = Unlock_fnc();
             if (Status_Unlock){
                wait_on_Lock_fnc();
                if (SafeR_Bus_var==Lock){
                    Lock_fnc();
                }
                else {delay;}
             }
             else {delay;}
    } //while
} // S_fnc

void main(void){CTRL_fnc();S_fnc();}
```

command input bus of the same name as the machine. The mechanism for requesting an external operation to execute is to change the data on the command input bus to the same name as the operation required. The last requested operation is latched into variable of the same name as the refined machine with a *._var* post fix. Variables are declared for the operation outputs. The names of the output bus variables are a concatenation of the operation output name and the operation name. This avoids clashes with similar operation output names. Buses are defined for each $/*IN*/$ and $/*OUT*/$ annotation, external operation, and operation output. Each operation is translated into a function. If an operation has an output the function will return a value. Functions with outputs will have an assignment in them that assigns to the bus output function variable. The function will also return that output in the final statement of the function. Assigning to the function output variable and writing it to a output port as well allows it to be put on the output bus, and used internally in the Handel-C program. The bodies are translated in a straightforward manner. Assignments in the operations are put together in a *par* Handel-C statement. Assignment and the $if-then-else$ B constructs have straightforward translations. The refined B example is limited to assignment and $if-then-else$. The *INITIALISATION* is translated into a function called *Initialisation_fnc*.

The CSP controller is used to construct the main Handel-C body. A summary of the hand translations made on the CSP controller are given in the table in Figure VI. The controller design was structurally limited to facilitate translation: initialisation and setting up operations are performed before a main loop is entered. The first process definition *CTRL_fnc* is not recursive; it is an open process. It translates to a function call *CTRL_fnc*, which invokes the *Initalisation_fnc* and *lock_fnc* functions. On returning to the main program the next function called is the *S_fnc*, which implements the main loop. *S_fnc* is tail recursive and is implemented with a continuously looping while loop; it is a closed process. The first event in the main loop is the *UnlockR* commands. In the translation the *Unlock_fnc* is preceded by *wait_Unlock_fnc* as it is an external operation. The *UnlockR_fnc* functions inputs from the $Cx1a$, $Cx1b$, $Cx2$,and $Cx2$ input buses. The *Unlock_fnc* call follows. *Unlock_fnc* returns a value that is assigned to a variable that is output ported. The value is also used to decide the course of the following if-then-else. Either a *Lock_fnc* or an *UnlockR_fnc* is performed after a wait. Then the process recurses.

4. Discussion

This paper has introduced a way of refining annotations that support a more general style of machine refinement closer to that of Event-B, by enabling the introduction of new operations in a refinement step. The paper has also set out a guide for translation to an HDL, within the B annotation framework. We have demonstrated how the framework previously presented can be extended for classical B. Our approach sits naturally with refinement. Refinement and translation are still being considered for CSP||B. In fact the B annotation approach offers several approaches to refinement: refinement of control flow only, state only, or control flow and state. The extensions to the annotations are fairly rich and now include annotations to support: next selection, sequencing, conditional, parallel execution, and I/O. The inability to define points of recursion within annotations has led to a reliance on a CSP controller. We restricted this paper to the consideration of fixed variables as operation inputs, and permitted no scope for controller state, and addressing these limitations is the subject of ongoing research. Further work is also required to automate the translation, and to develop the proof of the theorem to cover interleaving.

Table V. B to Handel-C Translation Guide.

Feature	B	Handel-C
set declaration	SETS SS= AA,...,XX/*n*/	typedef enum { AA = (unsigned n) 0, ..., XX } SS;
B variable declaration	INVARIANT Vv ∈ TT /*OUTn*/	unsigned n Vv; interface bus_out() Vv1 (unsigned 2 OutPort=Vv);
	INVARIANT Vv ∈ TT /*INn*/	unsigned n Vv; interface bus_in(unsigned n inp) Vv();
	INVARIANT Vv ∈ TT /*n*/	unsigned n Vv;
Function Declaration	/*extN*/ **Oo** ⟵ /*ext*/ **Cc**(/* M */Zz)	unsigned 1 Cc_var; interface bus_out () Oo_Cc1 (unsigned N Oo_Cc); interface bus_in(unsigned 1 inp) Cc (); void wait_on_Cc_fnc() { while (Cc.inp == Cc_var) { delay; } Cc_var = Cc.inp; } unsigned N Cc_fnc(unsigned M Zz){ par{...;};return exp; }
Function Body	PRE P THEN B END	par{<< B >>}
	IF b THEN c ELSE d END	if << b >> { << c >> } else { << d >> } ;
	b := c	<< b >> = << c >> ;
initialisation	INITIALISATION ...	void Initialisation(void){ ... ; }
main	OPERATION	void main(void){ Initialisation; ... }

Table VI. CSP to Handel-C Translation Guide.

Feature	CSP	Handel-C
initialisation processes	P$\widehat{=}$ ... R	P_fnc();Q_fnc(); void P_fnc(void){...;}
main loop processes	R$\widehat{=}$ ... R	R_fnc(); void R_fnc(void){while(1){...;}}
prefix (internal)	< e→ P >	e_fnc ; <P>
prefix (external)	< e→ P >	wait_on_e; e_fnc ; <P>
choice (external)	< P1 □ P2 >	<P1>
interleaved	$< e1 \rightarrow skip$ ||| ⋯ ||| $e_n \rightarrow skip;\ P >$	PAR{$< e_1 \rightarrow skip >$; ...; $< e_n \rightarrow skip >$}; $< P >$
if-then-else	<if y then P else Q>	if y {<P>} else {<Q>}
	where $< P >$ is the translation of P	

Acknowledgments

The extensions to the refinement have benefited from conversations with Stefan Hallerstede and Helen Treharne. Thanks are also due to the referees for positive comments, and detailed lists of errors, improvements and additions.

REFERENCES

1. J-R. Abrial. *The B-Book: Assigning Programs to Meaning*. Cambridge University Press, 1996.
2. J-R. Abrial. Event driven circuit construction version 5. MATISSE project, 2001.
3. A. Aljer, J. L. Boulanger, P. Devienne, S. Tison, and G. Mariano. BHDL: Circuit design in B. In *Applications of Concurrency to System Design*, pages 241–242. IEEE Computer Society, Elsevier, 2003.
4. A. Aljer and P. Devienne. Co-design and refinement for safety critical systems. In *19th IEEE International Symposium on Defect and Fault Tolerance in VSLI Systems (DFT'04)*, pages 78–86, 2004.
5. P. T. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann, 1996.
6. C. Fischer. CSP-OZ: a combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Proc. 2nd IFIP Workshop on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 423–438, Canterbury, UK, 1997. Chapman and Hall, London.
7. C. A. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
8. W. Ifill. Formal development of an example processor (AEP) in AMN, C and VHDL. MSc thesis, Royal Holloway, University of London, 1999.
9. W. Ifill. *B Annotations in Critical Control Systems Development*. PhD thesis, Department of Computing, University of Surrey, 2008.
10. W. Ifill, S. Schneider, and H. Treharne. Augmenting B with control annotations. In *B2007:Formal Specification and Development in B*, volume 4355 of *LNCS*, pages 399 – 424. Springer, 2007.
11. W. Ifill, I. Sorensen, and S. Schneider. The use of B to specify, design and verify hardware. In *High Integrity Software*. Kluwer Academic Publishers, 2001.
12. C. Metayer, J-R. Abrial, and L. Voisin. *Event B Language*. RODIN, 2005.
13. Alexandre Mota and Augusto Sampaio. Model-checking CSP-Z: Strategy, tool support and industrial application. *Science of Computer Programming*, 40(1):59–96, May 2001.
14. J. D. Phillips and G. S. Stilles. An automatic translation of CSP to Handel-C. In I. East, J. Martin, P. Welch, D. Duce, and M. Green, editors, *Communicating Process Architecures 2004*. IOS Press, 2004.
15. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998.
16. S. Schneider. *Concurrent and Real-time Systems: The CSP Approach*. John Wiley and Sons, 2000.
17. S. Schneider. *The B-Method: An introduction*. Palgrave, 2001.
18. S. Stepney. CSP/FDR2 to Handel-C Translation. Technical report, University of York, June 2003.
19. H. Treharne. *Combining Control Executives and Software Specifications*. PhD thesis, Royal Holloway, University of London, 2000.
20. H. Treharne and S. Schneider. Communicating B machines. In *ZB2002: Formal Specification and Development in Z and B*, volume 2272 of *LNCS*, pages 416 – 435. Springer, 2002.