Communicating Process Architectures 2007
Alistair A. McEwan, Steve Schneider, Wilson Ifill, and Peter Welch
IOS Press, 2007

A Step Towards Refining and Translating B Control Annotations to Handel-C

Wilson IFILL [a,b] and Steve SCHNEIDER [b]
[a] *AWE Aldermaston, Reading, Berks, England;*
[b] *Department of Computing, University of Surrey, Guildford, Surrey, England.*
{ W.Ifill , S.Schneider } @surrey.ac.uk

Abstract. Research augmenting B machines presented at B2007 has demonstrated how fragments of control flow expressed as annotations can be added to associated machine operations, and shown to be consistent. This enables designers' understanding about local relationships between successive operations to be captured at the point the operations are written, and used later when the controller is developed. This paper introduces several new annotations and I/O into the framework to take advantage of hardware's parallelism and to facilitate refinement and translation. To support the new annotations additional CSP control operations are added to the control language that now includes: recursion, prefixing, external choice, if-then-else, and sequencing. We informally sketch out a translation to Handel-C for prototyping.

Keywords. B Method, CSP, Hardware Description Language,

Introduction

Annotating B-Method specifications with control flow directives enables engineers to describe many aspects of design within a single notation. We generate proof obligations (*pobs*) to demonstrate that the set of executions allowable by the annotations of a B [1] [2] 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 CSP, which describe the flow of control explicitly, can be checked against the annotations. There is no need to check the CSP [3] [4] [5] directly against the full B description. Once the annotations are shown to be correct with respect to the B machine we can evaluate controllers against the annotations without further reference to the machine. Machines can be refined and implemented in the normal way while remaining consistent with the controller. In previous work [6] we presented the NEXT and FROM annotations, which permitted simple annotated B specifications and controllers to be written. Before that [7] we presented a route to VHDL [8], a hardware description language, from B. 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 begin to present an informal refinement theory for annotations and a route to implementation via Handel-C. The refinement theory outline in this paper allows the annotations to be independently refined and remain consistent with the Machine.

Previous work obtaining hardware implementations from B approached the problem by using B as a Hardware Description Language (HDL) that translates to VHDL [9] [10]. Our approach 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 [11] do not support the development process directly. Event B [12] has been

used to support the development of hardware circuits [13] 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 both state and control equally strongly. CSP||B [14] [15] 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 Moto and Sampaio [16] and CSP and Object Z (CSP-OZ) Fischer [17] require a CSP semantics to be given to Z in order for integration to be analysable as a whole. Our approach differs to other formal language integrations in two ways. Firstly, The control flow behaviour is capture during the development of the state operation in the form of annotation. The annotations are control specifications. 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 [6]. This papers contribution is the introduction of additional next annotations, incorporation of I/O into the annotations, and an informal treatment of refinement and translation. In Section 1, the general framework is introduced. In Section 2 a B machine is introduced along with the NEXT annotation. The proof obligations associated with the annotations and control language are given in Section 3. The consistency of the annotations are given in Section 4. A refinement and translation outline is given in Section 5. An example illustration of some refinements and translations are given in section 6. A discussion on the benefits and future work is had in Section 7.

We restrict our attention in this paper to correct B machines: those for which all proof obligations have already been discharged. We use I to refer to the invariant of the 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.

Controllers will be written in a simple subset of the CSP process algebraic language [3,5]. The language will be explained as it is introduced. Controllers are considered as *processes* performing *events*, which correspond to operations in the controlled B machine. Thus operation names will appear in the controller descriptions as well as the B machine definitions. The Handel-C translations are shallow and in a few cases performed in accordance with existing translation work [18] [19].

1. 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. The approach consists of the following components:

- **Machine definition**: the controlled component must first be defined.
- **Annotations**: the initialisation and the operations in the machine definition are annotated with fragments of control flow.
- **Annotation proof obligations**: verification conditions that 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.
- **Controller**: this is a process that describes the overall flow of control for the B machine.

- **Consistency checking**: establishing that the controller is consistent with the annotations by showing that that every part of the control flow is supported by some annotation.
- **Refine/Translate**: refinement may be needed before a translations can be achieved. The translation is the final step and requires additional annotation directives to set type sizes and I/O ports.

Checking a CSP controller against a machine is thus reduced to checking it against the annotations and verifying that the annotations are appropriate for the machine. The relationship between the different parts of the approach are given in Figure 1.

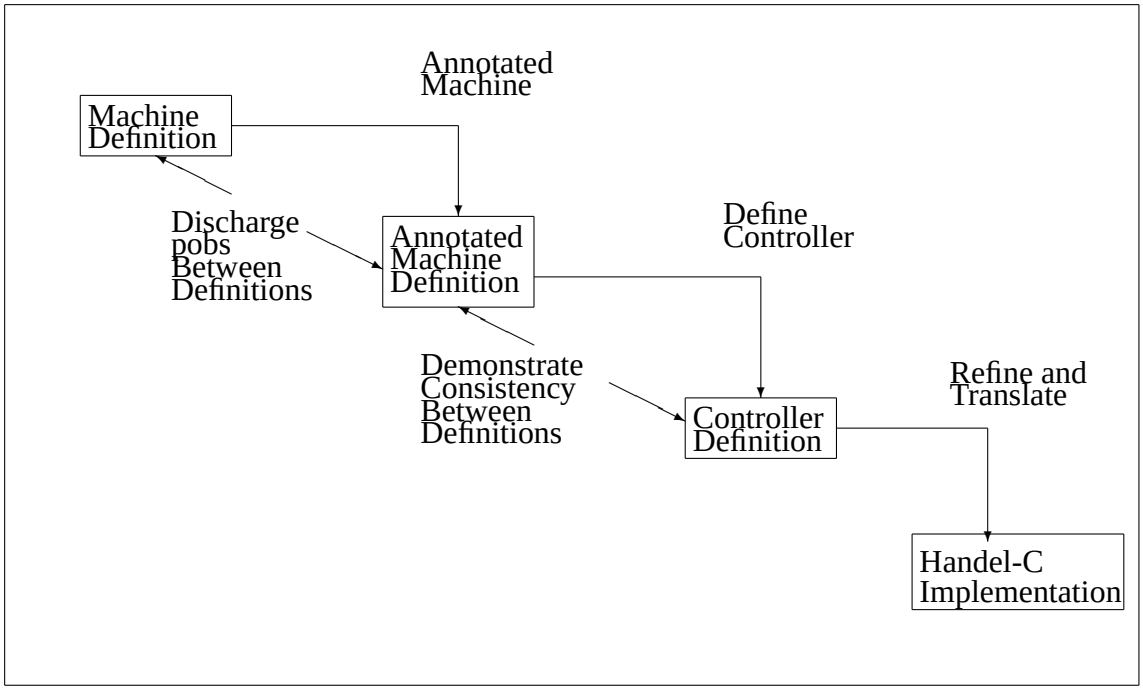

Figure 1. The Process Flow in the Approach.

The framework presented here is quite general, in that it may be applied to both Event-B and classical B. Additional annotations maybe added along with supporting control operations as required. Provided that a consistency argument can be developed. 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.

2. The Approach

We will demonstrate the approach 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, parallel, if-then-else, and recursion is used to develop the example.

2.1. A B Machine

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. B Machines are used in the examples. The generic classical B $MACHINE\ S$, 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.

$MACHINE\ S$

$VARIABLES\ v$

$INVARIANT\ v$

$INITIALISATION\ v :\in u$

$OPERATIONS$

$y \longleftarrow OP1(z_1) \mathrel{\widehat{=}} P_1 \mid B_1;$

$y_2 \longleftarrow OP2(z_2) \mathrel{\widehat{=}} G_2 \Longrightarrow B_2;$

...

$y_n \longleftarrow OPn(z_n) \mathrel{\widehat{=}} P_n \mid B_n$

END

The operations are defined in Guarded Substitution Language (GSL). It is asserted that the machine is consistent when each operation can be shown to establish the machine invariant, I, and the machine cannot deadlock. 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 do not in this paper adapt the proof obligations for Event-B refinement. The refinement process may involve adding detail to the specification in a consistent way to realise an implementation, which is a key notion in B. 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.

3. 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 proof obligations 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 $OP_{j_1}, \ldots, OP_{j_n}$, which will be enabled after OP_i has executed (where an operation in the annotation 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. The variables become ports in the hardware implementation. The value of these variable is not considered when calculating the proof obligations. Only the type of the variables is checked.

3.1. The Basic NEXT Annotation

$OP_i \quad \widehat{=} \quad$ **PRE** P_i **THEN** B_i **END** /* $\{OP_{j_1}, \ldots, OP_{j_n}\}$ NEXT */ **;**

Definition 3.1 (Proof Obligations of the Basic NEXT on INITIALISATION) *Given the following B initialisation:*

INITIALISATION T /* { $Op_j ?v_j$ } *NEXT* */ ;

the related proof obligations follow:

$$[T]((v_j \in T_j) \Rightarrow P_j)$$

The NEXT annotation following the initialisation indicates the first enabled operation. 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 ($?v_{j_1} \ldots ?v_{j_m}$) must be of the type required in the operation definition.

Definition 3.2 (Proof Obligations of the Basic NEXT on Operations) *Given the following B operation:*

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

the related proof obligations follow:

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

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

3.2. The NEXT_PAR Annotation

I/O operations can be annotated to indicate parallel execution NEXT_PAR. 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 3.3 (Proof Obligations of NEXT_PAR) *Given the following B operation:*

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

the related proof obligations follow:

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

$$(P_i \wedge I \Rightarrow [B_i]((v_{p_1} \in T_{p_1}) \Rightarrow P_{p_1})) \wedge$$
$$\ldots$$
$$(P_i \wedge I \Rightarrow [B_i]((v_{p_n} \in T_{p_n}) \Rightarrow P_{p_n})) \wedge$$

$$variable\_used(\{OP_{j_1}, \ldots, OP_{j_n}\}) \cap variable\_used(\{Op_{p_1}, \ldots, OP_{p_n}\}) = \{\}$$

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.

3.3. The NEXT_SEQ Annotation

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

Definition 3.4 (Proof Obligations of NEXT_SEQ) *Given the following B operation:*

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

the related proof obligations follow:

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

$$(P_{j_1} \wedge I \Rightarrow [B_{j_1}]((v_{p_1} \in T_{p_1}) \Rightarrow P_{p_1})) \wedge$$
$$\ldots$$
$$(P_{j_1} \wedge I \Rightarrow [B_{j_1}]((v_{p_n} \in T_{p_n}) \Rightarrow P_{p_n})) \wedge$$

$$\ldots$$

$$(P_{j_n} \wedge I \Rightarrow [B_{j_n}]((v_{p_1} \in T_{p_1}) \Rightarrow P_{p_1})) \wedge$$
$$\ldots$$
$$(P_{j_n} \wedge I \Rightarrow [B_{j_n}]((v_{p_n} \in T_{p_n}) \Rightarrow P_{p_n}))$$

where the elements of z_i and v_j and v_p 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 must enable each operation in $\{Op_{j_1}(v_{j_1}), \ldots, Op_{j_n}(v_{j_n})\}$, and each operation in that set must enable each operation in the set $\{Op_{p_1}(v_{p_1}), \ldots, Op_{p_n}(v_{p_n})\}$. Practically, this annotation should be used to depict particular paths: one operation per set.

3.4. 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 3.5 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. If however the current operation returns false then the operations OP_{p_1} through to OP_{p_n} are guaranteed to be available to execute. The proof of this claim can be verified by discharging the following proof obligation given in definition 3.5:

Definition 3.5 (Proof Obligation of NEXT_COND) *Given the following B operation:*

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

the related proof obligations follow:

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

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

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.

3.5. A Simple Controller Language

The next annotation represents a control fragment specification of the whole system. The CSP controller represents a refined view of the annotated B system. The annotated B machine hasn't 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. On translation both the B and the CSP are used to build the 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 5.

Definition 3.6 details the CSP subset of control fragments used in this paper: event prefix, choice, interleaving, if-then-else, and recursion control.

Definition 3.6 (Controller Syntax with I/O)

$$\begin{aligned} R ::= & \underset{y}{\Box}\, a!y?z \rightarrow R \mid \\ & R1 \,\Box\, R2 \mid \\ & (\underset{y_1}{\Box}\, a_1!y_1?z_1 \rightarrow \mathit{skip} \mid\mid\mid \ldots \mid\mid\mid \underset{y_n}{\Box}\, a_n!y_n?z_n \rightarrow \mathit{skip};\ R) \mid \\ & \underset{y}{\Box}\, e!y \ \rightarrow \mathit{if}\ y\ \mathit{then}\ R1\ \mathit{else}\ R2 \ \mid \\ & S(p) \end{aligned}$$

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 represented in figure 2. An annotated B machine output is the same as a CSP controller output. In definition 3.6 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 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 does not exist in the B. Hence the CSP representation of the B operation may not be a true representation in terms of input and output, which may be a subset 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 $if - then - else$ 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.

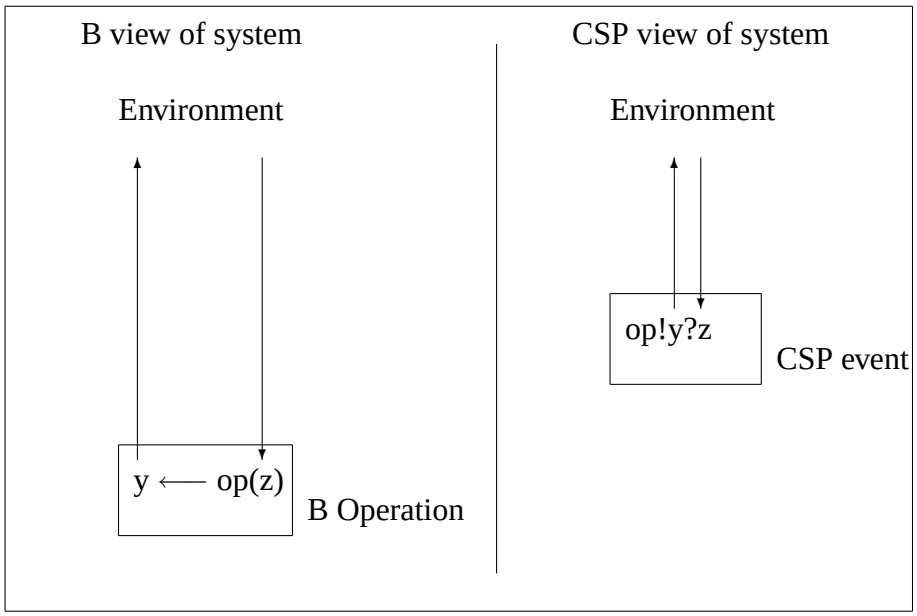

Figure 2. Different views of the same action.

A major constraint is enforced on the way controllers can be written. It facilitates translations, but turns out not to be so troublesome as it first appears. Controllers must start with an initialisation ($R1$), then enter a main loop ($S \mathrel{\widehat{=}} R2$). This is summarised in definition 3.7. A controller $CTRL$ has a definition, $R1$, given in definition 3.6, in which all the parameterised process variables are the same, S. The definition of S is $R2$ and is also given in definition 3.6. The only recursive calls allowed are to S.

Definition 3.7 (Controller Syntax with I/O)

$$CTRL \mathrel{\widehat{=}} R1$$
$$S \mathrel{\widehat{=}} R2$$
where $R1$ *and* $R2$ *are terms from definition* 3.6 *and*
S *is the only recursive variable allowed and*
$R2$ *is guarded as defined in definition* 3.9

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 $init$ function returns a set of operations available next and is developed in definition 3.8.

Definition 3.8 (init on CSP controller process with I/O extensions)

$$init(\underset{y}{\Box}\, a!y?z \rightarrow R1) = \{a\}$$
$$init(R1 \,\Box\, R2) = init(R1) \;\cup\; init(R2)$$
$$init(\underset{y_1}{\Box}\, a_1!y_1?z_1 \rightarrow skip \;|||\; \ldots \;|||\; \underset{y_n}{\Box}\, a_n!y_n?z_n \rightarrow skip);\; R = \{a_1, a_2, ..., a_n\}$$
$$init(if\ y\ then\ R1\ else\ R2) = init(R1) \cup init(R2)$$
$$init(S(p)) = init(R(p))$$

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 3.9. 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 3.9 (guarded on CSP controller process with I/O)

$$
\begin{aligned}
guarded(\underset{y}{\Box}\, a!y?z \rightarrow R1) &= true \\
guarded(R1 \,\Box\, R2) &= guarded(R1) \wedge\ guarded(R2) \\
guarded((\underset{y_1}{\Box}\, a_1!y_1?z_1 \rightarrow skip \;|||\; & \\
\ldots \;|||\; & \\
\underset{y_n}{\Box}\, a_n!y_n?z_n \rightarrow skip);\ R) &= true \\
guarded(if\ TRUE\ then\ R1\ else\ R2) &= guarded(R1) \wedge\ guarded(R2) \\
guarded(if\ FALSE\ then\ R1\ else\ R2) &= guarded(R1) \wedge\ guarded(R2) \\
guarded(S(p)) &= false
\end{aligned}
$$

4. I/O NEXT Consistency

Consistency between a $guarded$ controller and the annotated B machine is broken down into initial (definition 4.1) and step-consistency (definition 4.2).

Definition 4.1 (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
   A family of recursive definitions $S \mathrel{\widehat{=}} R$ is initially-consistent with M's annotations if each R is initially-consistent with M's annotations.

   [*We define $next(a)$ as the set of operations in the annotation of a.*]

A controller that starts with an interleaving or a conditional control fragment is not initially-consistent and should be avoided. An initialisation can not 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 4.2 (Step-Consistency of M with respect to M_CTRL) *The step-consistency of the controller fragment R is defined as follows:*

1. $\Box_y\ a!y?z \rightarrow R$
   is step-consistent with M if $\forall\, b \bullet b \in init(R) \Rightarrow b \in next(a)$, and R is step-consistent with M.

2. $R1 \,\Box\, R2$
   is step-consistent with M if $R1$ and $R2$ are step-consistent with M.

3. $(\Box_y\ a!y_a?z_a \rightarrow skip \ |||\ \Box_y\ b!y_b?z_b \rightarrow skip); R$
   is step-consistent with M if $\forall\, e \bullet e \in init(R) \Rightarrow e \in next(a)$ and $e \in next(b)$, and R is step-consistent with M, and $update(a!y_a?z_a) \cap update(b!y_b?z_b) = \{\}$.

4. $\Box_y\ e \rightarrow if\ y\ then\ R1\ else\ R2$
   is step-consistent with M if $y \in$ BOOL and $R1$ and $R2$ are step-consistent with M and
   $\forall\ b \in init\ (R1) \Rightarrow b \in condition\_true(e)$ *and*
   $\forall\ c \in init\ (R2) \Rightarrow c \in condition\_false(e)$
   where condition_true(e) returns the actions that are enabled when y=true and condition_false(e) returns the actions that are enabled when y=false.

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

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 4.3 (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. We have [6] proven a theorem that shows that this holds for the basic NEXT, and the NEXT_COND annotations. The annotations are lose enough to permit a large set of possible consistent controllers. As such the controller is viewed as a a trace refinement of the annotations. The controllers do not refine the annotations in a failures divergence sense. We believe, but have not yet proven, that the NEXT_PAR and NEXT_SEQ can be rewritten in the basic NEXT form.

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.

5. Refinement and Translation to Handel-C

Refining should be considered where an otherwise cumbersome translation would result. 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 1[1] replace 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 non-determinism in the choices offered in the next step. The NEXT interleave refinement reduces the non-determinism in one or more branches of the interleave execution. 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.

Table 2 outlines some structural refinements. 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 translation guide for annotations is given in table 3 and table 4. 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 6, and to some extent table 5, 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: $wait\_on\_\ldots$. 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 differentiate between internal and external operations. Hence the tables 3, 4, 7, 6, and 5 are all required to obtain a translation.

In tables 3 and 4 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.

[1] All tables for this section are given in the Appendix.

In table 4 case 5 is the translation of the NEXT_SEQ. In the previous section the NEXT_SEQ was introduced to support refinement: a basic NEXT is refined into a sequence of operations NEXT_SEQ. The refine an operation that both inputs and outputs to a sequence of operations must input at the beginning of the sequence and output at the end of the sequence. Case 5 reflects this requirement: the first operation in the sequence is an external operation that inputs and the final operation is an internal operation that outputs.

The translations of Stepney [19], and Phillips and Stilles [18] are given in table 5. 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. In the table the CSP language construct and translation are mapped. A tick is inserted if they are supported by Stepney (SS) or Phillips and Stilles (PS). 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 7 has been developed. A discussion of the example is given in section 6.

6. 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.

6.1. The Example's State and Control Flow

In figure 3 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 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. 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 $Unlocked$ was 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 4, 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 is repeated starting again at $CTRL$, if it is false then control is repeated at S.

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** ≙
PRE *Door* ∈ *COMMAND* **THEN** *Door* := *BrokenOpen* ‖ *Alarm* := *1* **END**
/* { BreakOpen } NEXT */ **;**

END

Figure 3. Safe Machine

$$CTRL = Initialisation \rightarrow \Box_{y}\, Lock!y \rightarrow S$$

$$S = (\Box_{y}\, Unlock!y \rightarrow (if\ y\ then\ \Box_{y}\, Lock!y \rightarrow CTRL\ else\ S)) \Box$$
$$(\Box_{y}\, BreakOpen!y \rightarrow B\_CTRL)$$

$$B\_CTRL = \Box_{y}\, BreakOpen!y \rightarrow B\_CTRL$$

Figure 4. Safe Machine Controller.

6.2. A Refined Example

A refinement of the $Safe$ machine, called SafeR, is given in figure 5 and figure 6 . It is a classical B refinement that mimicking a refinement in Event-B. The operation $UnlockR1$ and $UnlockR1$ are introduced to refine $Unlock$. The laws of refinement of Event-B are not fully justified. The $Safe$ REFINEMENT, $SafeR$, breaks down the Unlocking process into two stages. Firstly, a two new operation are slotted into the control in parallel: $UnlockR1(Comb1a, Comb1b)$ and $UnlockR2(Comb2a, Comb2b)$. Both have a combina-

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 */ **;**

Figure 5. Safe Refinement Part 1.

tion 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 obligations can be discharged. The bodies of the $UnlockR$ and $Rekey(Comb2)$ 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).

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

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

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

Figure 6. Safe Refinement Part 2

$$CTRL = Initialisation \rightarrow \Box_{y} Lock!y \rightarrow S$$
$$S = (UnlockR1?Cx1a?Cx1b \rightarrow skip \;|||\; UnlockR2?Cx2a?Cx2b \rightarrow skip) \rightarrow \Box_{y} Unlock!y \rightarrow (if\ y\ then\ \Box_{y} Lock!y \rightarrow S\ else\ S)$$

Figure 7. Refined Safe Controller.

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 ($UnlockR1$ and $UnlockR2$), and the output occurring on the final operation of the sequence (the original $Unlock$ operation).

The controller given in figure 7 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 $UnlockR1$and $UnlockR2$ are offered with $Cx1a$ and $Cx1b$ and $Cx2a$ and $Cx2b$ are offered as an input, respectively. The $UnlockR$ process is the first in a sequence of processes that

refines the original $UnLock$ process. The refined sequence starts with a parallel combination of the $UnlockR1$ and the $UnlockR2$ 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 the $Lock \rightarrow S$ could have been replaced by $CTRL$. However, the former is easier to translate.

6.3. A Hand Translation into Handel-C

The refined B specification provides the details of the types, variables, and functions. The CSP controller provides the executions details that are use later to construct the Handel-C main section. Summaries of hand translations of the refined B specification and the CSP controller are given in Figures 8, 9, and 10 (in Appendix B).

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 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 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 out 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 table 6. 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.

7. Discussion

This paper has introduced a way of refining annotations that support Event-B style refinement, and 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 both classical B and Event-B. Our approach sits naturally with refinement. Refinement and translation are still being considered for CSP||B. In fact the B annotation approach offerers 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 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. Work on CSP state and defining recursive points in the annotations is currently ongoing. More work is required to automate the translation and develop the proof of the theorem to cover interleaving.

Acknowledgements

The extensions to the refinement have benefited from conversations with Stefan Hallestede and Helen Treharne. Thank you for the positive comments from the referees and detailed lists of error eta, improvements and additions.

References

[1] J-R. Abrial. *The B-Book: Assigning Programs to Meaning*. Cambridge University Press, 1996.
[2] S. Schneider. *The B-Method: An introduction*. Palgrave, 2002.
[3] C. A. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, Englewood Cliffs, New Jersey, 1985.
[4] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998.
[5] S. Schneider. *Concurrent and Real-time Systems: The CSP Approach*. John Wiley and Sons, 1999.
[6] W. Ifill, S. Schneider, and H. Treharne. Augmenting B with control annotations. In J. Julliand and O. Kouchnarenko, editors, *B2007:Formal Specification and Development in B*, volume 4355 of *LNCS*. Springer, January 2007.
[7] W. Ifill, I. Sorensen, and S. Schneider. *High Integrity Software*, chapter The Use of B to Specify, Design and Verify Hardware. Kluwer Academic Publishers, 2001.
[8] P. T. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann, 1996.
[9] W. Ifill. Formal development of an example processor (AEP) in AMN, C and VHDL. Computer science, University of London, Computer Science Department, Royal Holloway, University of London, Egham, Surrey TW20 OEX, Sept 1999.
[10] 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, unknown 2003.
[11] 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.
[12] J-R. Abrial and L. Mussat. *Event B Reference Manual*. ClearSy, 1999.
[13] J-R. Abrial. Event driven circuit construction version 5. MATISSE project, August 2001.
[14] H. Treharne and S. Schneider. Communication B machines. In *ZB2002*, 2002.
[15] H. Treharne. *Combining Control Executives and Software Specifications*. PhD thesis, Royal Holloway, University of London, 2000.
[16] 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.
[17] C. Fischer. CSP-OZ: A combination of Object-Z and CSP.
[18] 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, 2004.
[19] S. Stepney. CSP/FDR2 to Handel-C translation. Technical report, University of York, June 2003.

A. Refinement and Translation Tables

Table 1. 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
	$OP_{j1} \mathrel{\widehat{=}} ...OP_X\ NEXT$ ... $OP_{jn} \mathrel{\widehat{=}} ...OP_X\ NEXT$	$OP_{j1} \mathrel{\widehat{=}} ...OP_X\ NEXT$ ... $OP_{jn} \mathrel{\widehat{=}} ...OP_X\ NEXT$	
	$OP_{k1} \mathrel{\widehat{=}} ...OP_X\ NEXT$ ... $OP_{kn} \mathrel{\widehat{=}} ...OP_X\ NEXT$	$OP_{k1} \mathrel{\widehat{=}} ...OP_X\ NEXT$ ... $OP_{kn} \mathrel{\widehat{=}} ...OP_X\ NEXT$	
3	$OP_i \mathrel{\widehat{=}} ...OP_J OP_P\ NEXT\_SEQ$	$OP_i \mathrel{\widehat{=}} ...OP'_J OP'_P NEXT\_SEQ$	next sequential refinement
	$OP_{j1} \mathrel{\widehat{=}} ...OP_P\ NEXT$ ... $OP_{jn} \mathrel{\widehat{=}} ...OP_P\ NEXT$	$OP_{j1} \mathrel{\widehat{=}} ...OP_P\ NEXT$ ... $OP_{jn} \mathrel{\widehat{=}} ...OP_P\ NEXT$	
4	$OP_i \mathrel{\widehat{=}} ...OP_J OP_P\ NEXT\_COND$	$OP_i \mathrel{\widehat{=}} ...OP'_J OP'_P NEX\_COND$	next condition refinement
	$OP_{j1} \mathrel{\widehat{=}} ...OP_P\ NEXT$ ... $OP_{jn} \mathrel{\widehat{=}} ...OP_P\ NEXT$	$OP_{j1} \mathrel{\widehat{=}} ...OP_P\ NEXT$ ... $OP_{jn} \mathrel{\widehat{=}} ...OP_P\ NEXT$	
		$where\ OP'_J \subseteq OP_J\ and$ $OP'_K \subseteq OP_K$	

Table 2. NEXT Refinements - Structural Refinements.

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

Table 3. NEXT Annotation Translation Guide Part 1.

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

Table 4. NEXT Annotation Translation Guide Part 2.

	Annotation	Handel-C Translation Fragment	Comment
5	$OP_i \mathrel{\widehat{=}} ...OP_J\ OP_K\ NEXT\_SEQ$	$y_i = OP_i(v_i);\ \ wait\_on\_OP_J$	next sequential translation
		if $in = OP_{j\,1}$	
	$/*\ ext\ */ OP_{j\,1} \mathrel{\widehat{=}} ...OP_K\ NEXT$	*then* $y_{j\,1} = OP_{j\,1}(v_{j\,1})$	
	...	*else* ...	
	$/*\ ext\ */ OP_{j\,n} \mathrel{\widehat{=}} ...OP_K\ NEXT$	...	
		if $in = OP_{j\,n}$	
	$OP_{k\,1} \mathrel{\widehat{=}} ...$	*then* $y_{j\,n} = OP_{j\,n}(v_{j\,n})$	
		else skip	
	$OP_{k\,n} \mathrel{\widehat{=}} ...$	;	
		$y_{k\,1} = OP_{k\,1}(v_{k\,1})$	
	$op_i!y_i?z_i \rightarrow (op_{j\,1}!y_{j\,1}?z_{j\,1} \rightarrow ... \Box$		
	$... \Box$		
	$op_{j\,n}!y_{j\,n}?z_{j\,n} \rightarrow ...);$		
	$(op_{k\,1}!y_{k\,1}?z_{k\,1} \rightarrow ... \Box$		
	$... \Box$		
	$op_{k\,n}!y_{k\,n}?z_{k\,n} \rightarrow ...)$		
6	$/*\ ext\ */ OP_i \mathrel{\widehat{=}} ...$	$y = OP_i(v_i);$	external
	$OP_J\ \ OP_K\ NEXT\_COND$	*if* y	next
		$\{wait\_on\_OP_J$;	condition
		if $in = OP_{j\,1}$	translation
		then $y_{j\,1} = OP_{j\,1}(v_{j\,1})$	
		else ...	
	$OP_{j\,1} \mathrel{\widehat{=}} ...OP_K\ NEXT$	...	
	...	*if* $in = OP_{j\,n}$	
		then $y_{j\,n} = OP_{j\,n}(v_{j\,n})$	
		else skip	
	$OP_{j\,n} \mathrel{\widehat{=}} ...OP_K\ NEXT$	}	
		else	
	$OP_{k\,1} \mathrel{\widehat{=}} ...OP_K\ NEXT$	{wait,	
	...	*if* $in = OP_{k\,1}$	
		then $y_{k\,1} = OP_{k\,1}(v_{k\,1})$	
		else ...	
	$OP_{k\,n} \mathrel{\widehat{=}} ...OP_K\ NEXT$	...	
		if $in = OP_{k\,n}$	
		then $y_{k\,n} = OP_{k\,n}(v_{k\,n})$	
		else skip	
	$op_i!y_i?z_i \rightarrow (op_{j\,1}!y_{j\,1}?z_{j\,1} \rightarrow ... \Box$	}	
	$... \Box$		
	$op_{j\,n}!y_{j\,n}?z_{j\,n} \rightarrow ...);$		
	$(op_{k\,1}!y_{k\,1}?z_{k\,1} \rightarrow ... \Box$		
	$... \Box$		
	$op_{k\,n}!y_{k\,n}?z_{k\,n} \rightarrow ...)$		

Table 5. 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	{- ... -}	...	✓	

Table 6. 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	

Table 7. 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; ... }

B. Hand Translations

```
// 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();
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 annotations
interface bus_out() Status_Unlock1 (unsigned 1 OutPort=Status_Unlock);
```

Figure 8. SafeR Translation Part 1a.

```
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;}
}
```

Figure 9. SafeR Translation Part 1b.

```
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();}
```

Figure 10. SafeR Translation Part 2.