

Augmenting B with Control Annotations

Wilson Ifill^{1,2}, Steve Schneider¹, and Helen Treharne¹

¹ Department of Computing, University of Surrey

² Atomic Weapons Establishment, Aldermaston, Reading, U.K.

Abstract. CSP||B is an integration of the process algebra Communicating Sequential Processes (CSP), and the B-Method, which enables consistent controllers to be written for B machines in a verifiable way. Controllers are consistent if they call operations only when they are enabled. Previous work has established a way of verifying consistency between controllers and machines by translating control flow to AMN and showing that a *control loop invariant* is preserved. This paper offers an alternative approach, which allows fragments of control flow expressed as *annotations* to be associated with machine operations. 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. Annotations provide a bridge between controllers and machines, expressing the relevant aspects of control flow so that controllers can be verified simply by reference to the annotations without the need to consider the details of the machine operations. This paper presents the approach through two instances of annotations with their associated control languages, covering recursion, prefixing, choice, and interrupt.

1 Introduction

The design and implementation of critical systems benefits from development in a formal method such as the B-Method, which models systems in terms of state and operations. However, this approach does not support specifications of execution patterns directly, and so approaches such as Event-B [6] and CSP||B [8] have been proposed to incorporate action specification with B. This paper develops the CSP||B approach, which offers a clean separation of control from data manipulation. The developments presented here fall within the scope of AWE's System-B project, which involves collaborative research into the use of CSP||B to specify co-designs [5] and to formally investigate systems designs of large scale developments.

One motivation for the work is a desire to enable Engineers to describe many aspects of design within a single notation. We introduce control annotations into the B-Method to enable the formal capture of control flow fragments in B during the development of the B machines. We generate proof obligations to

demonstrate that the set of executions allowable by the annotations do not cause operations to diverge. The benefit of this approach is that only the semantics of the machine operations is required in checking the annotations, and these checks are similar in size and difficulty to standard B machine consistency checks. Annotations can be checked against controllers written in CSP, which describe the flow of control explicitly. There is no need to check the CSP directly against the full B description, in contrast to previous CSP||B work where it was necessary to translate the entire CSP controller into AMN in order to check it. 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.

This paper describes the extendable framework for introducing annotations and controllers and presents two exemplars. This paper is not concerned with I/O operations. In Section 2, we briefly introduce the approach. In Section 3 we demonstrate the framework by using a simple language for controllers, the NEXT annotation for B operations, and define the notion of consistency between them. Section 4 presents a worked example of a simple traffic control system. In Section 5 we introduce an interrupting annotation FROM-ANY, add the CSP interrupt operator to the controller language, and extend the notion of consistency. We develop the worked example in Section 6 to illustrate the new annotation and its use. Finally in Section 7 we discuss further directions and related work.

We assume the reader is familiar with the Abstract Machine Notation of the B-Method [1]. 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 [4, 7]. 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.

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, whilst maintaining the separation of concerns. The approach consists of the following components:

- **Machine definition:** the controlled component must first be defined.

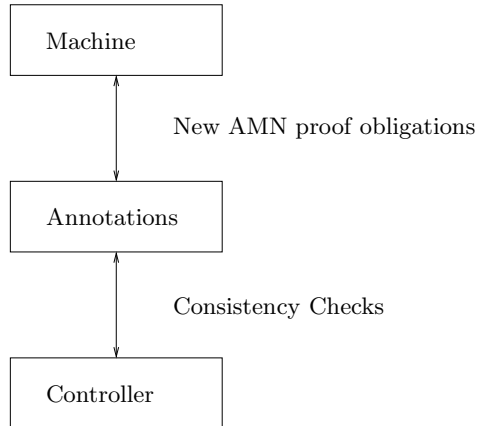


Fig. 1. Relationship between the different parts of the approach

- **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—that every part of the control flow is supported by some annotation.

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 is illustrated in Figure 1.

The framework presented here is quite general, in that it may be applied to a variety of annotations and control languages. The first step to be taken is therefore to fix on the control language and the associated annotations to be incorporated into the AMN machine descriptions. The key result that these build up to is expressed in Theorem 1, though the underlying theory will not be expanded in this paper for reasons of space.

3 A first approach

We will demonstrate the approach firstly with a simple model to illustrate how the aspects of the approach interrelate. The first kind of annotation we consider

is the NEXT annotation, and we use an extremely simple controller language consisting only of prefixing, choice, and recursion. These go naturally together because the NEXT annotation is concerned with successive operations, and the controller language allows simple loops of sequences of operations.

3.1 The NEXT annotation

We annotate an operation of a B machine with a NEXT annotation. Currently, we introduce this as a comment included with the description of the operation, so that it is invisible to current tools. However, in principle tools could be modified to recognise an additional ANNOTATION clause to introduce the additional information into operation descriptions.

A NEXT annotation on an operation Op_i introduces another operation Op_j , or set of operations Op_j, \dots, Op_k , which should be enabled after Op_i is executed.

The NEXT annotation is written as follows:

$$Op_i \hat{=} \mathbf{PRE} \ P_i \ \mathbf{THEN} \ B_i \ \mathbf{END} \ /* \ \{ Op_j, \dots, Op_k \} \ \mathbf{NEXT} \ */$$

3.2 Annotation proof obligations

The annotation corresponds to the assertion that, following the execution of Op_i , operations Op_j through to Op_k are available for execution. This gives rise to the following proof obligation, which requires that the precondition of each of the listed operations is enabled:

Definition 1 (NEXT Proof Obligation for Operations). *The proof obligation associated with a NEXT annotated operation Op_i is given as:*

$$\begin{aligned} & (I \wedge P_i \Rightarrow [B_i](P_j)) \\ & \wedge \dots \\ & \wedge (I \wedge P_i \Rightarrow [B_i](P_k)) \end{aligned}$$

If the conjunction of proof obligations for all the annotations are discharged then we say that the annotations are consistent with the machine. This ensures that any controller which only calls operations that are listed, following execution of Op_i , can be sure that those operations will be enabled.

Definition 2 (NEXT Proof Obligation for Initialisation). *The proof obligation associated with the annotation*

$$T /* \{ Op_j, \dots, Op_k \} \ \mathbf{NEXT} \ */$$

on initialisation T is given by

$$\begin{aligned} & [T](P_j) \\ & \wedge \dots \\ & \wedge [T](P_k) \end{aligned}$$

This establishes that all of the listed operations are enabled following initialisation. Thus, any controller which only begins with such operations will be consistent with the annotations.

We will use $next(Op_i)$ to identify the set of operations given in the NEXT annotation. Thus from the annotation above we have that $next(Op_i) = \{Op_j, \dots, Op_k\}$. We also use $next(INITIALISATION)$ to identify the set of operations in the annotation of the INITIALISATION clause. To ensure that there is no deadlock in the system, it is sufficient that every operation, and the INITIALISATION, has a NEXT annotation.

3.3 A simple controller language

We will begin with the following simple controller language, which allows only event prefix, choice, and recursion:

Definition 3 (Controller Syntax).

$$R ::= a \rightarrow R \mid R \square R \mid S$$

Here, the event a is an operation name, and S is a process variable. Recursive definitions are then given as $S \hat{=} R$. In a controller definition, all process variables used are bound by some recursive definition. 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.

3.4 Consistency

We can now give a definition of consistency between a controller and the annotations on a B machine. The key underlying idea is that whenever one event Op_j follows another Op_i in the controller's execution, then there must be an annotation that underpins this, ensuring that the associated operation Op_j is guaranteed to be enabled after Op_i has occurred.

To do this, we first capture the initial events $init(R)$ for a controller R :

Definition 4 (initial elements of CSP controller process).

$$\begin{aligned} init(a \rightarrow R1) &= \{a\} \\ init(R1 \square R2) &= init(R1) \cup init(R2) \\ init(S) &= init(R) \quad \text{where } S \hat{=} R \end{aligned}$$

Note that in a controller definition the process variable S must be bound by some recursive definition $S \hat{=} R$, and this defines $init(S)$.

For example, if $LOOP \hat{=} a \rightarrow b \rightarrow LOOP$, then $init(LOOP) = a$.

A controller will be *step-consistent* with a collection of annotations if all consecutive events are allowed by the occurrence of some annotation. In the case where the only kind of annotation is NEXT, it is straightforward to define step-consistency, and we do this over the structure of the syntax.

Definition 5. [*Step-consistency of NEXT Annotated Machines and Controllers*] The step-consistency of a controller R with the annotations of machine M is defined structurally over the syntax of R as follows:

1. $a \rightarrow R$ is step-consistent with M 's annotations if $init(R) \subseteq next(a)$ and R is step-consistent with M 's annotations.
2. $R1 \square R2$ is step-consistent with M 's annotations if $R1$ is step-consistent with M 's annotations and $R2$ is step-consistent with M 's annotations.
3. S is step-consistent with M 's annotations.

A family of recursive definitions $S \hat{=} R$ is step-consistent with M 's annotations if each R is step-consistent with M 's annotations.

There is one additional aspect of consistency required: that the initial state of the machine is consistent with the starting point of the controller. This is captured as *initial-consistency*:

Definition 6 (Initial-Consistency of NEXT Annotated Machines and Controllers). A controller R is initially-consistent with the annotations of machine M if $init(R) \subseteq next(INITIALISATION)$.

Definition 7 (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 the following theorem:

Theorem 1. 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 key feature of the proof of this theorem is an argument that no trace of R leads to an operation of M called outside its precondition. This is established by building up the traces of R and showing that at each step an operation

```

MACHINE Lights
SETS COMMAND = { Stop , Go }
VARIABLES Moat , Square
INVARIANT (Moat = Stop  $\vee$  Square = Stop)
            $\wedge$  Moat  $\in$  COMMAND  $\wedge$  Square  $\in$  COMMAND
INITIALISATION Moat , Square := Stop , Stop /* { Stop_All } NEXT */

OPERATIONS
  Stop_All  $\hat{=}$  PRE true THEN Moat , Square := Stop , Stop END
    /* { Go_Moat , Go_Square } NEXT */ ;
  Go_Moat  $\hat{=}$  PRE Moat = Stop  $\wedge$  Square = Stop THEN Moat := Go END
    /* { Stop_All , Stop_Moat } NEXT */ ;
  Stop_Moat  $\hat{=}$  PRE Moat = Go THEN Moat := Stop END
    /* { Go_Moat , Go_Square } NEXT */ ;
  Go_Square  $\hat{=}$  PRE Moat = Stop  $\wedge$  Square = Stop THEN Square := Go END
    /* { Stop_All , Stop_Square } NEXT */ ;
  Stop_Square  $\hat{=}$  PRE Square = Go THEN Square := Stop END
    /* { Go_Moat , Go_Square } NEXT */
END

```

Fig. 2. Lights machine

$$\begin{aligned}
Lights_CTRL &\hat{=} Stop_All \rightarrow S_CTRL \\
S_CTRL &\hat{=} (Go_Moat \rightarrow Stop_Moat \rightarrow S_CTRL) \\
&\quad \square (Go_Square \rightarrow Stop_Square \rightarrow S_CTRL)
\end{aligned}$$

Fig. 3. Lights Controller

called outside its precondition cannot be introduced, by appealing to the relevant annotation and applying its proof obligation.

The benefit of this theorem 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.

4 Example: Carcassonne Traffic Control System

We use the example of a traffic light system to illustrate the ideas introduced in the previous section.

A traffic control system for the main street of the walled Cité of Carcassonne is specified. The main street is narrow and is heavily used by tourists and some motor vehicles brave enough to edge through the alley. The system must allow traffic up into the cité market square from the moat or down from the square to the moat gate along the same single width road. The system must allow time for motor vehicles to clear the road before changing direction. A B machine that offers a choice between the traffic flows is given in Figure 2. A controller consistent with the annotations is given in Figure 3. We note that the controller given here is more restrictive than necessary; it is not the weakest controller consistent with the annotations.

In order to show that *Lights_CTRL* is an appropriate controller for *Lights*, we make use of the annotations. We must show that the annotations are consistent with the machine, and we must also show that the controller is consistent with the annotations. We consider each of these in turn.

4.1 Consistency of annotations with the machine

The proof obligations associated with the annotations (eliding the invariant) are as follows:

- **Initialisation:** the initialisation clause must establish the precondition of all the operations identified in its annotation; in this case this is *Stop_All*, with precondition *true*. From Definition 2, we must prove

$$[Moat, Square := Stop, Stop](true).$$

- **Stop_All:** there are two next operations, *Go_Moat* and *Go_Square*, and so there will be a proof obligation associated with each of them. In fact each of them have the same precondition: $Moat = Stop \wedge Square = Stop$. Hence the two proof obligations are identical, and correspond to

$$I \wedge P_{Stop_All} \Rightarrow [Moat, Square := Stop, Stop](Moat = Stop \wedge Square = Stop).$$

- **Go_Moat:** there are two next operations, *Stop_All* and *Stop_Moat*, identified in the annotation. For *Stop_All*, the precondition is *true*, so the proof obligation is $I \wedge P_{Go_Moat} \Rightarrow [Moat := Go](true)$. Considering *Stop_Moat*, its precondition is $Moat = Go$, so the corresponding proof obligation is

$$I \wedge P_{Go_Moat} \Rightarrow [Moat := Go](Moat = Go).$$

- **Go_Square:** the annotation and hence the proof obligations for this operation are entirely similar to those for *Go_Moat*, but assigning to *Square* this time.

- **Stop_Moat**: there are two next operations, Go_Moat and Go_Square , so there will be a proof obligation associated with each. The proof obligation associated with Go_Moat is given by $I \wedge P_{Stop_Moat} \Rightarrow [B_{Stop_Moat}](P_{Go_Moat})$, which expands to

$$(Moat = Stop \vee Square = Stop) \wedge (Moat = Go) \\ \Rightarrow [Moat := Stop](Moat = Stop \wedge Square = Stop)$$

The proof obligation associated with Go_Square is entirely similar, since the precondition for Go_Square is the same as that of Go_Moat .

- **Stop_Square**: the annotation and hence the proof obligations for this operation are entirely similar to that for $Stop_Moat$.

In all cases the proof obligations are discharged. Note that in the case of $Stop_Moat$ the invariant and its precondition are necessary for establishing that the operation body establishes the precondition of the next operations.

Discharging the proof obligations means that the annotations are a correct description of allowable sequences of operations. Thus they can be used to verify the appropriateness of the controller.

4.2 Consistency of the controller with the annotations

To show that the controller $Lights_CTRL$ is consistent with $Lights$ we apply the definitions of step-consistent and initially-consistent. Let R_CTRL be the body of the definition of S_CTRL . Then it is necessary to show that R_CTRL is step-consistent with the annotations of the $Lights$ machine.

Step-consistency is established by considering the parts of the definition of R_CTRL :

- The process variable S_CTRL is step-consistent, by the definition of step-consistency for process variables.
- $Stop_All \rightarrow S_CTRL$: the prefix rule for step-consistency from Definition 5 requires that $init(S_CTRL) \subseteq next(Stop_All)$. This is true in this case, since the process variable S_CTRL is step-consistent and

$$init(S_CTRL) = \{Go_Moat, Go_Square\} = next(Stop_All).$$
- $Stop_Moat \rightarrow S_CTRL$: this is step-consistent, since the process variable S_CTRL is step-consistent and

$$init(S_CTRL) = \{Go_Moat, Go_Square\} = next(Stop_Moat).$$

- $Go_Moat \rightarrow Stop_Moat \rightarrow S_CTRL$: step-consistency follows from the fact that $Stop_Moat \rightarrow S_CTRL$ is step-consistent, and

$$\begin{aligned} init(Stop_Moat \rightarrow S_CTRL) &= \{Stop_Moat\} \\ &\subseteq \{Stop_Moat, Stop_All\} \\ &= next(Go_Moat). \end{aligned}$$

- $Stop_Square \rightarrow S_CTRL$: this is step-consistent, since the process variable S_CTRL is step-consistent, and

$$init(S_CTRL) = \{Go_Moat, Go_Square\} = next(Go_Square).$$

- $Go_Square \rightarrow Stop_Square \rightarrow S_CTRL$: step-consistency follows from the fact that the process $Stop_Square \rightarrow S_CTRL$ is step-consistent, and

$$\begin{aligned} init(Stop_Square \rightarrow S_CTRL) &= \{Stop_Square\} \\ &\subseteq \{Stop_Square, Stop_All\} \\ &= next(Go_Square). \end{aligned}$$

- $Go_Moat \rightarrow Stop_Moat \rightarrow S_CTRL \sqcap Go_Square \rightarrow Stop_Square \rightarrow S_CTRL$: this is step-consistent, due to the step-consistency of both sides of the choice.

Initial-consistency follows from the fact that

$$next(INITIALISATION) = init(Lights_CTRL).$$

Thus, S_CTRL is consistent with the annotations of the machine $Lights$, and so the controller is appropriate for the machine.

5 Introducing FROM-ANY annotations and interrupts

Section 2 introduced the key components of the annotation approach that provide a framework for developing controlled systems. In general there will be a variety of annotations that we will want to make use of, and a richer language for controllers. These will have an impact on the consistency relationship, and on the underlying proofs which will need to be adapted to accommodate the changes.

In this section we will extend the controller language to include interrupts, which are commonly used to introduce interruptions in the control flow between one operation and the next. A further annotation will be introduced to accompany this extension to the controller language, and we will see the impact on the notion of consistency.

5.1 The FROM-ANY Annotation

The introduction of interrupts in the control language gives rise to another annotation, the FROM-ANY annotation.

The FROM-ANY annotation is written `/* FROM-ANY */`. This annotation is added to an operation which can follow *any* previous operation (including itself), and can also follow initialisation. It will naturally be used on an operation which follows an interrupt, since such an operation might follow any previous operation, allowing for the fact that the operation might happen anywhere.

Its use in an arbitrary operation Op_i is given as follows:

$$Op_i \hat{=} \mathbf{PRE} \ P_i \ \mathbf{THEN} \ B_i \ \mathbf{END} \ \text{/* FROM-ANY */} ;$$

5.2 Annotation proof obligation

The annotation corresponds to the claim that after the execution of any operation, Op_i will always be available to execute. The annotation gives rise to the following proof obligation: that the precondition P_i of Op_i is enabled after any precondition, and also that it is enabled after initialisation:

Definition 8 (FROM-ANY Proof Obligations). *The proof obligation associated with a FROM-ANY annotated operation Op_i is given as:*

$$\forall op \in OPERATIONS \bullet P_{op} \wedge I \Rightarrow [B_{op}]P_i \wedge [T]P_i$$

where T is the initialisation of the machine.

A condition sufficient to establish the proof obligation of Definition 8 is the assertion $I \Rightarrow P_i$. Its use is captured as a lemma:

Lemma 1. *If $I \Rightarrow P_i$ for an operation Op_i with a FROM-ANY annotation, then the proof obligations on Op_i associated with this annotation are all true.*

When it holds, this is a simpler condition to establish. It may not always hold, since it is stronger than the FROM-ANY proof obligations. However, note that if those proof obligations hold then the invariant of the machine could be strengthened to include P_i , in which case the lemma will then hold.

Operations can be annotated with both a FROM-ANY annotation and a NEXT annotation. The former indicates what the operation can follow, and the latter indicates what can come next.

For a machine M , we define $from-any(M)$ to be the set of operations of M that are annotated with a FROM-ANY clause.

5.3 Controller language

We introduce an interrupt operator to the control language as follows:

Definition 9 (Controller Syntax).

$$R ::= a \rightarrow R \mid R \square R \mid R \triangle R \mid S$$

The global interrupt operator, \triangle , permits the second controller fragment to interrupt the former at any point, even before the first action of the former has been performed. However, we do not rely on an interrupt establishing initial-consistency. There should always be a *next* annotation in the INITIALISATION.

The $init(R)$ function was defined on controllers by means of a structural induction over the controller syntax. Thus the introduction of an interrupt clause into the controller syntax necessitates a revision to the definition of $init(R)$, as follows:

Definition 10 (init on CSP controller process).

$$\begin{aligned} init(a \rightarrow R1) &= \{a\} \\ init(R1 \square R2) &= init(R1) \cup init(R2) \\ init(R1 \triangle R2) &= init(R1) \cup init(R2) \\ init(S) &= init(R) \quad \text{where } S \hat{=} R \end{aligned}$$

The first event that can be performed by $R1 \triangle R2$ is either a first event from $R1$, or else a first event from $R2$ following the occurrence of the interrupt.

5.4 Consistency

We again provide a definition for consistency between a controller and the annotations. This is again separated into a notion of step-consistency, which is concerned with successive events; and initial-consistency, regarding the initial state of the system.

The notion of step-consistency now needs to take account of a further clause in the controller language, and the fact that the machine M has more than one kind of annotation. Thus the definition has one additional clause. We introduce $from-any(M)$ to denote the set of operations of M with a FROM-ANY annotation.

Definition 11 (Step-consistency of NEXT and FROM-ANY Annotated Machines and Controllers). *The step-consistency of a controller R with the annotations of machine M is defined structurally over the syntax of R as follows:*

1. $a \rightarrow R$ is step-consistent with M 's annotations if $\text{init}(R) \subseteq (\text{next}(a) \cup \text{from-any}(M))$ and R is step-consistent with M 's annotations.
2. $R1 \square R2$ is step-consistent with M 's annotations if $R1$ is step-consistent with M 's annotations and $R2$ is step-consistent with M 's annotations.
3. $R1 \triangle R2$ is step-consistent with M 's annotations if $R1$ is step-consistent with M 's annotations, $R2$ is step-consistent with M , and $\text{init}(R2) \subseteq \text{from-any}(M)$.
4. S is step-consistent with M 's annotations.

A family of recursive definitions $S \hat{=} R$ is step-consistent with M 's annotations if each R is step-consistent with M 's annotations.

In the case for $a \rightarrow R$, we require that every operation b that R can perform first, which are those operations in $\text{init}(R)$, must be able to follow a , either because b is in $\text{next}(a)$ and hence identified explicitly as an operation that can follow a , or because b is in $\text{from-any}(M)$, and hence can follow anything.

In the case for interrupt, we have that $R1$ must be step-consistent with M because all executions of $R1$ are possible executions of $R1 \triangle R2$; $R2$ must also be step-consistent since control can pass to $R2$; and every operation that $R2$ can initially perform must be able to follow anything, since the interrupt can occur at any point.

The cases for choice and for recursion are similar to the previous version of step-consistency.

Definition 12 (Initial-Consistency of NEXT and FROM-ANY Annotated Machines and Controllers). A controller R is initially-consistent with the annotations of machine M if $\text{init}(R) \subseteq \text{next}(\text{INITIALISATION}) \cup \text{from-any}(M)$.

As stated previously, a controller R is consistent with the annotations of a B machine M if it is step-consistent and initially-consistent with the annotations of M .

Once again we have everything in place to establish the main theorem of this section:

Theorem 2. *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 pre-conditions.*

6 Example continued

We develop the example of the Carcassonne traffic control system. We wish to extend the controller so that normal operation can be interrupted at any point with all lights being set to *Stop*. The resulting controller is given in Figure 4.

$$\begin{aligned}
Lights_CTRL2 &\hat{=} Stop_All \rightarrow S_CTRL2 \\
S_CTRL2 &\hat{=} S_INNER \triangle Stop_All \rightarrow S_CTRL2 \\
S_INNER &\hat{=} (Go_Moat \rightarrow Stop_Moat \rightarrow S_INNER) \\
&\quad \square (Go_Square \rightarrow Stop_Square \rightarrow S_INNER)
\end{aligned}$$

Fig. 4. A Second Lights Controller

The NEXT annotations of the machine *Lights* are not sufficient to establish consistency with *Lights_CTRL2*, and in particular the interrupt requires consideration. The only event immediately following the interrupt is *Stop_All*, so we require a FROM-ANY annotation on that operation in addition to the NEXT annotation it already has. The resulting operation is as follows:

```

Stop_All  $\hat{=}$  PRE true THEN Moat, Square := Stop, Stop END
/* { Go_Moat, Go_Square } NEXT */
/* FROM-ANY */

```

This annotation introduces an additional proof obligation. Since the precondition P_{Stop_All} of *Stop_All* is *true*, it follows that $I \Rightarrow P_{Stop_All}$, and hence by Lemma 1 that the annotation is consistent with the machine.

It remains to show that the new controller *Lights_CTRL2* is consistent with the annotated machine. Initial-consistency, and most of the step-consistency cases are similar to those seen in the consistency check for *Lights_CTRL* and we do not repeat them here. However, the new conditions are required to consider the interrupt construction $S_INNER \triangle Stop_All \rightarrow S_CTRL2$. Step-consistency requires us to check three conditions:

1. S_INNER is step-consistent: this follows from the definition of step-consistency on process variables.
2. $Stop_All \rightarrow S_CTRL2$ is step-consistent: this follows since S_CTRL2 is step-consistent and

$$\begin{aligned}
init(S_CTRL2) &= \{Go_Moat, Go_Square, Stop_All\} \\
&= \{Go_Moat, Go_Square\} \cup \{Stop_All\} \\
&= next(Stop_All) \cup from-any(Lights)
\end{aligned}$$

3. $init(Stop_All \rightarrow S_CTRL2) \subseteq from-any(Lights)$. This follows from the fact that $init(Stop_All \rightarrow S_CTRL2) = \{Stop_All\}$ and that *Stop_All* has a FROM-ANY annotation.

Thus we conclude that *Lights_CTRL2* is an appropriate controller for the machine *Lights*.

7 Discussion

We are currently investigating further extensions to the framework. Operations with input and output arise naturally in B machines, and can have the annotations described previously. However, the situation is more complex, since controllers can also pass information from one operation call to another. This can lead to complications in the definitions of step-consistency, and it is necessary to carry around information obtained from previous operation calls when reasoning about step-consistency. This also gives rise to parameterised recursive definitions. Query operations are of particular interest, since it appears that different considerations apply: they do not change the state of the machine, but require output which can affect control flow.

The current approach requires separate construction of annotations and of controllers. One longer term aim of this line of research is the ability to synthesise controllers from the machine annotations. Such a controller would be the weakest controller consistent with a machine, and other consistent controllers would then be refinements. This is a topic of future research.

We now consider related work. The notion of incorporating temporal properties in B is not new. Abrial and Mussat [2] introduced the temporal operators of *next*, *eventually* and *leadsto*. In the case of *leadsto* (written \rightsquigarrow) they focus on identifying predicates P and Q such that if P holds at any point then Q eventually must hold, together with a list of events that make progress towards satisfying the final predicate Q . Their approach does not explicitly define the order in which these events must occur nor whether they occur more than once, the clause simply identifies which events can be performed in order to satisfy the $P \rightsquigarrow Q$ predicate. They use these predicates to express properties of the system which must hold when the temporal ordering of events is considered. We are using annotations to give us a handle on what operations are allowed to be performed when considering the temporal ordering of operations. We do not use them as a basis for expressing properties of a system and therefore use distinct clauses to define a possible ordering on operations in a novel way. Our approach does resonate with [2] in that we do not change the proof obligations that already exist but also identify additional proof obligations in order to ensure that the temporal orderings suggested by the annotations are sensible ones.

More recent work on Event-B and B[#] [6] introduces proof obligations both for feasibility (that events and initialisation do not block), and for deadlock-freedom of a system (that the disjunction of the event guards in the system is true). The control is left implicit, but the properties are sufficient to ensure that at any stage the execution can continue.

The approach presented in this paper makes the control flow explicit, and thus deadlock-freedom would follow directly from the deadlock-freedom of a consistent controller, provided the operations are non-blocking. Feasibility and deadlock-freedom obligations cannot be expressed directly in terms of the operation annotations presented here, but this is to be expected: The information in an

annotation is more directive, whereas deadlock-freedom is a general property which does not contain specific control information within it.

We could represent our running example in Event-B as follows:

```
stopall = when true then c := 1 end  
gosquare = when c=1 then c:= 2 end  
stopsquare = when c=2 then c:= 1 end  
gomoat = when c=1 then c := 3 end  
stopmoat = when c=3 then c:= 1 end
```

The predicates in the guards determine whether an event is enabled or not and governs when an event can be performed. In this approach the control flow is implicit and not always straightforward to understand or extract. As we saw in our example it is possible to allow either **gosquare** or **gomoat** to be performed when both variables are in the state *Stop* because the next annotation of the stop all operation refers to both these operations. In our approach this was translated to an external choice in CSP. In Event-B if more than one guard is true then the decision as to which event is performed is internal. Because we are only dealing with temporal ordering at the level of traces this distinction is not significant. However, contrary to normal B consistency one important difference is that we do not need to examine the preconditions of all the operations to identify the next set of possible operations. By using annotations we can clearly see which operations should be available to be performed following an operation because we can look at each operation in isolation. Note that there may be other operations that are enabled but not mentioned in an annotation—this could possibly be because the designer providing the annotations does not want them to happen at that point.

The approach in [9] combines CSP and B so that CSP captures, primarily, the event aspect of the design, whereas the B captures the state evolution. Each CSP controller directs a single B machine via communication channels. Controllers may also interact with other controllers. In [9], consistency between the pre-conditioned B machine and the CSP controllers is established in two ways. Firstly, by showing that operations are always called within their preconditions, which establishes divergence freedom. Guarded controllers present the possibility of controller deadlock. A second, consistency condition establishes that controllers are deadlock free. Consistency is investigated using the weakest preconditions of guarded commands [3], by translating the controller into AMN and demonstrating that it preserves a control loop invariant (CLI). In contrast, in this paper we establish divergence freedom by showing that the controller is consistent with the machine annotations. If every operation has a next annotation then the machine is also deadlock-free.

Acknowledgements We are grateful to Neil Evans for his comments and suggestions on this paper, and to the anonymous reviewers for their thorough reviewing and insightful comments.

References

1. J-R. Abrial. *The B-Book: Assigning Programs to Meaning*. Cambridge University Press, 1996.
2. J-R. Abrial and L. Mussat. Introducing dynamic constraints in B. In *B'98*, number 1393 in LNCS. Springer, 1998.
3. E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, 1997.
4. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
5. A. McEwan and S. Schneider. A verified hardware development using CSP||B. In *Fourth ACM-IEEE International Conference on Formal Methods and Models for Codesign*, 2006.
6. C. Métayer, J-R. Abrial, and L. Voisin. Event-B language, 2005. RODIN deliverable 3.2, Project IST-5111599.
7. S. Schneider. *Concurrent and Real-time Systems: The CSP Approach*. John Wiley and Sons, 1999.
8. S.A. Schneider and H.E. Treharne. CSP theorems for communicating B machines. *Formal Aspects of Computing*, 17(4):390–422, 2005.
9. H. Treharne. *Combining Control Executives and Software Specifications*. PhD thesis, Royal Holloway, University of London, 2000.