ELECTRONIC WORKSHOPS IN COMPUTING

Series edited by Professor C.J. van Rijsbergen

**He Jifeng, Oxford University Computing Laboratory, UK, John Cooke,
Loughborough University, UK, and Peter Wallis, University of Bath, UK (Eds)**

BCS-FACS 7th Refinement Workshop

Proceedings of the BCS-FACS 7th Refinement Workshop, Bath, 3-5
July 1996

Development of Concurrent Systems in B AMN

K. Lano and J. Dick

Published in Collaboration with the
British Computer Society

©Copyright in this paper belongs to the author(s)

ISBN 3-540-76104-7

Development of Concurrent Systems in B AMN

K. Lano*, J. Dick†

Abstract

This paper outlines an approach to extending B AMN to support concurrent specification, using a combination of linear temporal logic and Ada style task definitions. The extension is applied to the “production cell” case study.

Keywords: B AMN, Object calculus, concurrent specification, safety and liveness properties.

1 Introduction

The B Abstract Machine Notation (AMN) is a formal specification and development language which is related to Z [24], and incorporates modularity mechanisms which support the construction of large systems. It has been used to develop automatic train control systems [3, 6, 5], medical information systems and modules of the CICS software [12]. It has mature and commercially available tool support for the complete development lifecycle, and is now being taught at a number of universities. Commercial courses on the language and method are also available.

Industrial users of B have identified the lack of treatment of concurrency as the major outstanding problem with the language [12]. Although reactive systems have been developed using B AMN, the lack of support for concurrency has limited the general application of the language in this domain, and the extent to which industrial users can benefit from the language.

This paper will provide some techniques for concurrent specification using extensions to the B AMN language.

2 B Abstract Machine Notation

2.1 Syntax

The B AMN formal method was originally developed by J. R. Abrial and others working at BP Research and GEC Alsthom as an industrially usable, tool supported technique for high-integrity software production. Major applications for train protection and control systems were undertaken in France [3], and more recently IBM have adopted the method as a means of extending their work on the CICS system using Z.

B AMN uses a mathematical notation and logic which is very close to that of Z. A major distinction between B and Z is in the use of ‘abstract machines’ or specification modules, rather than schemas, as the means of decomposing a system description. These modules have the schematic form:

```
MACHINE M(p)  
SETS St  
CONSTANTS k  
PROPERTIES B  
VARIABLES v  
INVARIANT I
```

---

*Dept. of Computing, Imperial College, 180 Queens Gate, London SW7 2BZ

†B-Core (UK) Ltd., Oxford Science Park, Oxford OX4 4GA

```

INITIALISATION T
OPERATIONS
  y  $\leftarrow$  op(x) =
    PRE P
    THEN
      S
    END
  . . . .
END

```

They encapsulate constant data (described in the SETS, CONSTANTS and PROPERTIES clauses), varying data (described in the VARIABLES and INVARIANT clauses), and provide a public interface consisting of a set of OPERATIONS with inputs **x** and outputs **y**. The INITIALISATION describes how the module is to be initialised. Operations use a mathematical programming language termed the ‘generalised substitution notation’. A *generalised substitution* is an abstract mathematical programming construct, built up from basic substitutions $\mathbf{x} := \mathbf{e}$, corresponding to assignments to state variables, via the following operators:

SKIP	No-op
CHOICE S ₁ OR S ₂ END	Bounded choice
PRE P THEN S END	Preconditioning
SELECT P THEN S END	Guarding
ANY v WHERE P THEN S END	Unbounded non-determinism
S ₁ ; S ₂	Sequential composition
S ₁ S ₂	Multiple generalised substitution
WHILE E DO S	
INVARIANT I	
VARIANT e END	Looping

where **S**, **S**₁ and **S**₂ represent generalised substitutions, **e** an expression, **I**, **E** and **P** predicates, and **v** a variable or list of variables.

Machines can import features from other machines by means of the SEES, INCLUDES, EXTENDS and IMPORTS clauses, and others. Operations with input variables **x** and output variables **y** can be invoked by statements of the form $\mathbf{b} \leftarrow \mathbf{op}(\mathbf{a})$ in machines that access a machine in which **op** is defined. More details on the notation are given in [15]. The locality principle of the object calculus [9] is used in B AMN: it is not possible for operations of a machine directly to update variables which have been imported from another machine. Instead, these variables can only be updated by invoking an operation of the machine in which they were declared.

Sequential composition can only be used in REFINEMENT and IMPLEMENTATION components, and WHILE loops only in IMPLEMENTATIONS. Abstract constructs (PRE, CHOICE, ANY, SELECT and ||) cannot be used in an IMPLEMENTATION.

The semantics of operations is specified by means of weakest preconditions $[\mathbf{S}]\mathbf{P}$ for generalised substi-

tutions $\mathbf{S}$ and predicates $\mathbf{P}$. $[\ ]$ is defined as follows:

$$\begin{array}{ll}
[\mathbf{x} := \mathbf{e}]\mathbf{P} & \equiv \mathbf{P}[\mathbf{e}/\mathbf{x}] \\
[\text{SKIP}]\mathbf{P} & \equiv \mathbf{P} \\
[\text{CHOICE } \mathbf{S}_1 \text{ OR } \mathbf{S}_2 \text{ END}]\mathbf{P} & \equiv [\mathbf{S}_1]\mathbf{P} \wedge [\mathbf{S}_2]\mathbf{P} \\
[\text{PRE } \mathbf{E} \text{ THEN } \mathbf{S} \text{ END}]\mathbf{P} & \equiv \mathbf{E} \wedge [\mathbf{S}]\mathbf{P} \\
[\text{SELECT } \mathbf{E} \text{ THEN } \mathbf{S} \text{ END}]\mathbf{P} & \equiv \mathbf{E} \Rightarrow [\mathbf{S}]\mathbf{P} \\
[\text{ANY } \mathbf{v} \text{ WHERE } \mathbf{E} \text{ THEN } \mathbf{S} \text{ END}]\mathbf{P} & \equiv \forall \mathbf{v}.(\mathbf{E} \Rightarrow [\mathbf{S}]\mathbf{P}) \\
[\mathbf{S}_1; \mathbf{S}_2]\mathbf{P} & \equiv [\mathbf{S}_1][\mathbf{S}_2]\mathbf{P} \\
[\text{WHILE } \mathbf{E} \text{ DO } \mathbf{S} \\
\quad \text{INVARIANT } \mathbf{I} \\
\quad \text{VARIANT } \mathbf{v} \text{ END}]\mathbf{P} & \Leftarrow \mathbf{I} \wedge \\
& \quad 1 : \forall \mathbf{l}.(\mathbf{I} \wedge \mathbf{E} \Rightarrow [\mathbf{S}]\mathbf{I}) \wedge \\
& \quad 2 : \forall \mathbf{l}.(\mathbf{I} \wedge \neg \mathbf{E} \Rightarrow \mathbf{P}) \wedge \\
& \quad 3 : \forall \mathbf{l}.(\mathbf{I} \wedge \mathbf{E} \Rightarrow \mathbf{v} \in \mathbb{N}) \wedge \\
& \quad 4 : \forall \mathbf{l}.(\mathbf{I} \wedge \mathbf{E} \wedge \mathbf{v} = \gamma \Rightarrow [\mathbf{S}](\mathbf{v} < \gamma))
\end{array}$$

In the case of $[\mathbf{v}.\mathbf{S}]\mathbf{P}$ $\mathbf{v}$ is not free in $\mathbf{P}$, and in the last case, γ is a new variable not free in the WHILE substitution or the predicates concerned, and $\mathbf{l}$ is the list of variables modified within the loop.

The semantics of $[\ ]$ is defined partly by the requirement:

$$\text{prd}_{\mathbf{x},\mathbf{y}}(\mathbf{S}_1 \parallel \mathbf{S}_2) \equiv \text{prd}_{\mathbf{x}}(\mathbf{S}_1) \wedge \text{prd}_{\mathbf{y}}(\mathbf{S}_2)$$

where $\mathbf{x}$ is the list of variables updated in $\mathbf{S}_1$, and $\mathbf{y}$ those in $\mathbf{S}_2$. $\text{prd}_{\mathbf{v}}$ converts a generalised substitution $\mathbf{S}$ on variables $\mathbf{v}$ into a predicate on $\mathbf{v}$ and $\mathbf{v}'$ which expresses the effect of the substitution as a relation between initial states $\mathbf{v}$ and final states $\mathbf{v}'$:

$$\text{prd}_{\mathbf{x}}(\mathbf{S}) \equiv \neg [\mathbf{S}]\neg(\mathbf{x}' = \mathbf{x})$$

$[\mathbf{S}]\mathbf{P}$ means “ $\mathbf{S}$ establishes $\mathbf{P}$ ” or, that every execution of $\mathbf{S}$ terminates in a state satisfying $\mathbf{P}$.

As a result, we can interpret $\text{SELECT } \mathbf{P} \text{ THEN } \mathbf{S} \text{ END}$ as meaning “wait for $\mathbf{P}$ to become true, and then execute $\mathbf{S}$ ”. That is, the only execution paths through the guarded statement are those beginning in a state satisfying $\mathbf{P}$. In contrast, $\text{PRE } \mathbf{P} \text{ THEN } \mathbf{S} \text{ END}$ may execute when the prestate does not satisfy $\mathbf{P}$, however there is then no guarantee over its behaviour (it may block, fail to terminate or produce an arbitrary result).

B is a sequential specification language in that operations are regarded as single atomic actions, and it is not possible, at the implementation level, for two or more machines to be executing concurrently. The extensions described in this paper will allow concurrent execution between machine implementations, but not concurrent execution within a machine.

3 Extending B with Concurrency

The advantage of B over other formal languages such as Z and RSL is primarily its emphasis on a small, easily learnt, set of notations (which also makes tool development for the language more practicable). Thus care has to be taken in extending B not to change this property. In particular, we want to add concurrency in an orthogonal manner, with as little modification to the original language as possible in the case of sequential specifications.

Thus we do not want to take the approach of RSL and closely integrate concurrent and sequential language elements [11]. Instead we propose three new notations, based on elements of the VDM^{++} language [8], but given a uniform semantics in the object calculus [9]. These are traces, linear temporal logic formulae, and threads.

There is a close relationship between B AMN and the object calculus, both in terms of the (object-based) modularity approach of both formalisms, their refinement concepts, and in terms of their mathematical basis.

Integration of these formalisms provides a means to specify safety properties, permission constraints on operations, and many other constraints on the possible sequences of operations which a module can execute.

The following sections introduce these notations and give illustrative examples of their use.

3.1 Trace Specifications

A *trace* specification describes in a concise manner what the allowed histories of a B module can be. It consists of a set of trace set definitions and an overall history specification of the acceptable histories.

A trace set definition has the form

Identifier = TraceSetExpression

where a trace set expression is either an explicit definition

(trace, alphabet)

of a trace and an alphabet, or is built up from trace set identifiers and explicit trace set definitions using operators such as **w** (weave, or synchronised interleaving [23]).

An alphabet is a set of operation names from the machine and a trace is a regular expression in operation names, built using the operators ; (sequence), * (iteration zero or more times), + (iteration one or more times) and | (binary choice). By default the **alphabet** is the set of operation names mentioned in the **trace**, and only needs to be explicitly written if this is not the case.

Thus a machine with cyclic behavior could be specified as:

```
MACHINE Cyclic
OPERATIONS
  opa;
  opb;
  opc
TRACES
  GENERAL (opa; opb; opc)*
END
```

The **GENERAL** clause defines the overall machine trace set. An operation name listed without a definition defaults to the completely non-deterministic operation on the state.

An example of the effect of the **w** operator is the following combination, corresponding to the concurrent composition of two binary statecharts which share a transition:

$$(\text{opa}; \text{opb})^* \text{ w } (\text{opb}; \text{opc})^* = \text{opa}; (\text{opb}; ((\text{opa}; \text{opc}) \mid (\text{opc}; \text{opa})))^*$$

w is associative and commutative.

3.2 Permission Guards

A permission guard **G** on an operation **op** is a condition which must be satisfied before execution of **op** can proceed. In the case that **G** refers only to the current values of machine variables this form of specification is already available in B via select statements: we simply define **op** using the construct **SELECT G THEN S** **END**.

In the case that **G** refers to a previous history of actions however it is not so direct to express guarding by a **SELECT** statement, and so a separate part of a machine definition giving permission guards is useful. This has the form

```

GUARDS
  op  $\Rightarrow$  Guard;
  :

```

where each **op** is an operation name (possibly with input parameters), and **Guard** is a condition over the state variables, operation parameters, and counts **count**(**op'**) of the number of completed executions of operations **op'** of the machine.

For example, the condition that there can be a lag of at most **del** between an occurrence of **opa** and a “responding” occurrence of **opb** is specified using:

```

MACHINE Delay(del)
OPERATIONS
  opa;
  opb
GUARDS
  opa  $\Rightarrow$  count(opa) < count(opb) + del
END

```

A history $\langle \mathbf{opa}, \mathbf{opa}, \mathbf{opb}, \mathbf{opa}, \mathbf{opb} \rangle$ would be allowed by this specification in the case that **del** = 2, whilst a history $\langle \mathbf{opa}, \mathbf{opa}, \mathbf{opb}, \mathbf{opa}, \mathbf{opa} \rangle$ would not.

3.3 Threads

A *thread* is a new clause of a refinement or implementation, written as

```

THREAD
  statement

```

where **statement** usually consists of an unbounded loop WHILE **true** DO **body** INVARIANT **I** END, and **body** is a select statement. We extend the syntax of select statements to include ANSWER clauses (akin to **accept** in Ada):

```

SELECT
  cond1 ANSWER op1
  THEN stat1
WHEN
  cond2 ANSWER op2
  THEN stat2
WHEN
  :
END

```

If a guarded command within a select statement contains an ANSWER **op**_{**i**} clause, then it cannot proceed to execute the corresponding **stat**_{**i**} until an external caller attempts to call **op**_{**i**}, and **cond**_{**i**} must also be true at this point in time. The guarded command is executed by executing **op**_{**i**} and then executing **stat**_{**i**} (without interruption from other operations of the machine). It is possible to mix guarded commands without ANSWER clauses with guarded commands with such clauses within the same select statement. If more than one guarded command can be executed, then the choice between them remains non-deterministic. This form of statement is essentially the same as that described in [10], where the labels of guarded commands are the names of actions.

Concurrency is introduced by a thread because the caller of an operation **op** is only blocked waiting for **op** to return whilst the ANSWER clause is executing – once the text of the operation is completed, the thread

of the called machine can execute concurrently with the caller. Using this, an operator `|||` to perform two non-interfering segments of code in parallel can be defined.

A thread defines exactly what external operation calls can be accepted by the machine: any such call must correspond with the execution of an ANSWER statement for that operation. The thread of a machine is initiated at the start of the execution of the system. No operation which occurs in an ANSWER statement in the thread can also appear in the remainder of the thread code, since otherwise deadlock could occur. The operations that appear in ANSWER statements are a subset of the external services of the machine, whilst those that appear in the remainder of the code should usually be internal operations of the machine.

Threads are used to implement permission guards and trace specifications in a procedural manner, suitable for translation into Ada or OCCAM code.

3.4 Histories

Finally, a section is needed to describe behaviours which concern liveness or fairness constraints which cannot be expressed using the other constructs. In this section we allow the use of temporal logic operators $\diamond$ (eventually), $\square$ (henceforth) and $\bigcirc$ (next). For example, to assert that after any occurrence of **opa** there will eventually be an occurrence of **opb**, we could write:

```
MACHINE Liveness
OPERATIONS
    opa;
    opb
HISTORY
     $\square(\text{opa} \Rightarrow \diamond \text{opb})$ 
END
```

3.5 Summary

The notations introduced above give complementary descriptions of features of a reactive module:

- static invariant properties of the module data are expressed in the PROPERTIES and INVARIANT clauses. This covers many cases of safety properties;
- the local state transition achieved by an operation, together with invocations of operations from accessed machines, is defined in the OPERATIONS clause;
- global properties of the allowed sequences of events (represented by operation executions), termed “entity life histories” in methods such as SSADM, and traces in CSP, are expressed in the TRACES clause;
- conditions which must hold before an operation can execute (“await” conditions) are expressed in the GUARDS clause. Some fairness properties can be given in this form;
- liveness properties and some fairness properties can be expressed in the HISTORY clause.

The thread gives an implementation-oriented definition of global history properties and permission constraints. Permission guards are distinguished from preconditions – preconditions of an operation are expected to hold before its execution, but do not (necessarily) cause a delay if they fail to hold, but rather an unpredictable behaviour.

4 Semantics of Extended B

In this section we will describe how the semantics of extended B can be given in terms of object calculus theories, and how the concept of refinement arising from this semantics relates to that of conventional B.

4.1 The Object Calculus

An object calculus theory consists of a set of type and constant symbols, *attribute symbols* (denoting time-varying data) *action symbols* (denoting atomic operations) and axioms describing the types of the attributes and the effects, permission constraints and other dynamic properties of the actions. The axioms are specified using linear temporal logic operators $\bigcirc$ (next), $\bullet$ (in the previous state), $\mathcal{U}$ (strong until), $\mathcal{S}$ (strong since), $\square$ (always in the future) and $\diamond$ (sometime in the future). There is assumed to be a first moment. The predicate **BEG** is true exactly at this time point. We add the least fixed point operator μ and greatest fixed point operators ν [21]: this means that if φ is a formula both $\mu \mathbf{x} : \varphi$ and $\nu \mathbf{x} : \varphi$ are formulae.

$\bigcirc$ and $\bullet$ are also expression constructors. If $\mathbf{e}$ is an expression, $\bigcirc \mathbf{e}$ denotes the value of $\mathbf{e}$ in the next time interval, whilst $\bullet \mathbf{e}$ denotes the value of $\mathbf{e}$ in the previous time interval.

A **count** operator which gives the number of completed occurrences of an action symbol can also be defined. We have

$$\mathbf{BEG} \Rightarrow \mathbf{count}(\mathbf{op}) = 0$$

and

$$\mathbf{op} \Rightarrow \bigcirc \mathbf{count}(\mathbf{op}) = \mathbf{count}(\mathbf{op}) + 1$$

$$\neg \mathbf{op} \Rightarrow \bigcirc \mathbf{count}(\mathbf{op}) = \mathbf{count}(\mathbf{op})$$

for any action $\mathbf{op}$.

An important element of the object calculus is the *locality principle*. This asserts that the attributes may only change in value over an interval in which at least one of the actions of the module are executing:

$$\mathbf{att} = \bigcirc \mathbf{att} \vee \mathbf{op}_1 \vee \dots \vee \mathbf{op}_n$$

where $\mathbf{op}_1, \dots, \mathbf{op}_n$ are all the actions of the module $\mathbf{M}$ declaring $\mathbf{att}$. This can also be written as

$$\mathbf{idle}_M \Rightarrow \mathbf{att} = \bigcirc \mathbf{att}$$

where $\mathbf{idle}_M$ abbreviates $\neg (\mathbf{op}_1 \vee \dots \vee \mathbf{op}_n)$.

4.2 Interpreting B AMN in the Object Calculus

A B AMN machine can be viewed as describing a particular form of object calculus specification. A machine

```

MACHINE M
SETS St1, ...
CONSTANTS Con1, ...
PROPERTIES Prop
VARIABLES v1, ...
INVARIANT Inv
INITIALISATION T
OPERATIONS
  y1  $\longleftarrow$  op1 = Def1;
   $\vdots$ 
  yn  $\longleftarrow$  opn = Defn
END

```

gives rise to a theory with:

- action symbols $\mathbf{op}_1, \dots, \mathbf{op}_n$ (possibly parameterised), together with an action symbol **init** for the initialisation;

- constant and type symbols for the machine constants and sets;
- attribute symbols for the machine variables $\mathbf{v}_i$ and operation results $\mathbf{y}_j$;
- axioms for the properties and invariant;
- axioms $\mathbf{P} \wedge \mathbf{op}_i \Rightarrow \bigcirc \mathbf{Q}$ for every pair $\mathbf{P}, \mathbf{Q}$ of predicates over the machine data items such that $\mathbf{P} \Rightarrow [\mathbf{Def}_i]\mathbf{Q}$.

An alternative (finitary) way of expressing the operation effects is to use axioms

- $\mathbf{Pre}_i \wedge \mathbf{op}_i \Rightarrow (\neg [\mathbf{S}_i](\mathbf{v}' \neq \mathbf{v}))[\bigcirc \mathbf{v}/\mathbf{v}']$ where $\mathbf{v}$ is the tuple of machine variables, $\mathbf{Def}_i$ is $\mathbf{PRE} \mathbf{Pre}_i$ THEN $\mathbf{S}_i$ END and $\chi[\mathbf{e}/\mathbf{v}]$ denotes substitution of $\mathbf{e}$ for $\mathbf{v}$ in χ .

The predicate $\neg [\mathbf{S}_i](\mathbf{v}' \neq \mathbf{v})$ expresses the possible state transitions which can be achieved by executions of $\mathbf{S}_i$ in terms of a relation between a pre-state $\mathbf{v}$ and post-state $\mathbf{v}'$. For example:

$$\neg [\mathbf{CHOICE} \mathbf{x} := 1 \text{ OR } \mathbf{x} := 2 \text{ END}](\mathbf{x} \neq \mathbf{x}') \equiv \mathbf{x}' = 1 \vee \mathbf{x}' = 2$$

For a B machine $\mathbf{M}$, all operations declared in $\mathbf{M}$ will be assumed to be mutually exclusive in their execution. However they can execute concurrently with operations from machines included in $\mathbf{M}$ since such operations update disjoint sets of variables from those of $\mathbf{M}$. The operators $\bigcirc$ and $\bullet$ are not relative to the machine theory in which they appear: they refer to the successor and predecessor global intervals, respectively. We also need past and future time operators $\bullet_{\mathbf{M}}$ and $\bigcirc_{\mathbf{M}}$ which refer to the most recent past interval at which an operation of a particular machine executed and the closest future interval at which an operation of the machine will execute, respectively. That is, they abbreviate $\bigcirc_{\mathbf{M}}\varphi \equiv \mathbf{idle}_{\mathbf{M}}\mathcal{U}\varphi$ and $\bullet_{\mathbf{M}}\varphi \equiv \mathbf{idle}_{\mathbf{M}}\mathcal{S}\varphi$. Operations of the machine include external and internal operations of the machine.

If machine $\mathbf{M}$ makes use of another machine $\mathbf{N}$ via the SEES, USES or INCLUDES mechanisms, then the data of $\mathbf{N}$ can be considered part of the data of $\mathbf{M}$ for the purpose of the interpretation (restrictions on the use of data and operations of $\mathbf{N}$ in $\mathbf{M}$ being enforced by tools where necessary). Since sequential composition is not allowed in the definition of operations of $\mathbf{M}$, it can be considered that any invocation of an operation $\mathbf{op}_{\mathbf{N}}$ of $\mathbf{N}$ from an operation $\mathbf{op}$ of $\mathbf{M}$ occurs “concurrently” with $\mathbf{op}$ in the sense of the object calculus: the two executions occur in the same conceptual interval of time, reading data values from the same time point, and updating the state in a consistent manner.

In contrast, for B AMN refinements and implementations, an execution of an operation of machine $\mathbf{M}$ may correspond to a “transaction” or series of executions of operations from machines which it SEES or IMPORTS. An example is given in Section 7 below.

A permission constraint is interpreted as an axiom which constrains an action to only be executable under certain conditions. The guard statement $\mathbf{op} \Rightarrow \mathbf{Guard}$ is interpreted as the corresponding object calculus formula. For example if $\mathbf{opa}$ was only allowed to occur if $\mathbf{att} \leq 10$ we would have:

$$\mathbf{opa} \Rightarrow (\mathbf{att} \leq 10)$$

Likewise, axioms written in the HISTORY constraint are directly translated into object calculus formulae.

Traces can (only) be interpreted as temporal logic formulae in the object calculus language by using the least-fixed point and greatest fixed-point operators μ and ν [21]. As such, traces are strictly more expressive than the linear temporal logic of Object-Z histories [7].

For a thread $\mathbf{T}$ in machine $\mathbf{M}$, we can extract a temporal logic formula $\mathbf{Proc}_{\mathbf{M}}(\mathbf{T})$ which expresses the histories allowed by $\mathbf{T}$. Details of this translation are given in [18].

We can also express properties of thread histories in the loop invariant. For example, in a refinement with two operations $\mathbf{aop}$ and $\mathbf{bop}$ we could have:

```

WHILE true
DO
  SELECT

```

```

    ANSWER aop
  THEN
    SELECT
      ANSWER bop
    THEN SKIP
  END
END
INVARIANT history  $\models$  (aop; bop)*
END

```

The invariant of this thread definition asserts that the sequence **history** of completed operation invocations obeys the trace specification (**aop**; **bop**)* at the execution points beginning and ending each loop iteration. This invariant is proved by weakest precondition reasoning as usual: the key step in this case is the valid inference

$$\mathbf{history} \models (\mathbf{aop}; \mathbf{bop})^* \Rightarrow \mathbf{history} \cap [\mathbf{aop}, \mathbf{bop}] \models (\mathbf{aop}; \mathbf{bop})^*$$

As a result, we can infer that the thread implements the trace specification

GENERAL (**aop**; **bop**)*

for this machine.

history $\models$ **trace** is an abbreviation for a formula over sequences of tokens for operation names.

The above semantics is closely related to the standard set-theoretic semantics of B [1], and extends it by making the sequences of states experienced by machine models explicit.

4.3 Internal Consistency

The internal consistency of an extended B machine is equivalent to the consistency of its derived theory. In general, this cannot be feasibly checked, instead we would aim to generate proof obligations which detect common errors made in formalisation. An extended B machine has the form:

```

MACHINE M(p)
CONSTRAINTS ConstraintsM
SETS St
CONSTANTS k
PROPERTIES PropertiesM
VARIABLES v
INVARIANT InvM
INITIALISATION InitM
OPERATIONS
  y  $\leftarrow$  op(x) =
    PRE Preop,M
    THEN
      Defop,M
    END
  . . . .
TRACES
  Traces
GUARDS
  op  $\Rightarrow$  Gop;
  . . . .
END

```

without a thread or history, for simplicity.

Then the conventional internal consistency obligations of **M** (without traces or guards) are given in Table 1. The obligations (1), (2) and (3) of existence of suitable constants and variables satisfying the declared

(1)	$\exists p.\mathbf{Constraints}_M$
(2)	$\mathbf{Constraints}_M \Rightarrow \exists(\mathbf{St}, \mathbf{k}).\mathbf{Properties}_M$
(3)	$\mathbf{Constraints}_M \wedge \mathbf{Properties}_M \Rightarrow \exists \mathbf{v}.\mathbf{Inv}_M$
(4)	$\mathbf{Constraints}_M \wedge \mathbf{Properties}_M \Rightarrow [\mathbf{Init}_M]\mathbf{Inv}_M$
(5)	$\mathbf{Constraints}_M \wedge \mathbf{Properties}_M \wedge \mathbf{Inv}_M \wedge \mathbf{Pre}_{\mathbf{op}, M} \Rightarrow [\mathbf{Def}_{\mathbf{op}, M}]\mathbf{Inv}_M$

Table 1: Proof Obligations for Internal Consistency

invariants and properties of this data are retained, as is the initialisation obligation (4). In obligation (5) we can additionally assume $\mathbf{G}_{\mathbf{op}}$ in the antecedent. However a new existence constraint is:

$$(6) \quad \exists(\mathbf{v}, \mathbf{x}).(\mathbf{Pre}_{\mathbf{op}, M} \wedge \mathbf{Inv}_M \wedge \mathbf{G}_{\mathbf{op}})$$

That is, there is at least one state of the machine which satisfies both the guard and the precondition, for each operation $\mathbf{op}$.

For each operation $\mathbf{op}$ we also require that there exists at least one non-trivial history which satisfies the traces and guard conditions of the machine, and which contains $\mathbf{op}$:

$$(7) \quad \exists \mathbf{h} : \text{seq}(\underline{\mathbf{operations}}(\mathbf{M})).(\mathbf{op} \in \text{ran}(\mathbf{h}) \wedge \mathbf{h} \models \mathbf{Traces} \wedge \mathbf{Guards})$$

where $\mathbf{Guards}$ is the conjunction of the temporal formulae corresponding to the permission statements, and $\underline{\mathbf{operations}}(\mathbf{M})$ is the list of (tokens for) the operations of $\mathbf{M}$.

More generally, we could formulate the set of “normative histories”, which are sequences of operation invocations where the post-state of one invocation is within the precondition of the successive invocation. We should expect that the trace sets defined for a machine are a subset of this set of normative histories.

These additional obligations will detect errors such as initial deadlock (no non-trivial traces of the machine), and failing to provide any possibility of valid execution of a particular operation. They also help check the consistency of the permission guards with the trace specifications.

Animation techniques for B could be extended with checks that the history of operations up to the current interval is consistent with the traces, and that an operation guard is true when the operation is selected for execution. Prototype animation tools which cover these issues, and more complex problems, such as timing constraints, have already been developed [16].

Threads are required to implement permission guards by ensuring, via their control flow and SELECT guards, that each $\mathbf{G}_{\mathbf{op}}$ holds before execution of $\mathbf{op}$ can take place. They are also required to ensure trace constraints, usually via invariants of the unbounded loops within the thread.

4.4 Refinement Concepts

An important result is that refinement between B AMN modules implies refinement (ie, theory extension) between their translations as object calculus theories. In the language of category theory, this means that the translation can form the basis of a *functor* from the category of B modules with functional refinements as the morphisms, to the category of object calculus specifications with theory interpretations as morphisms.

This follows since if machine $\mathbf{N}$ with data $\mathbf{v} : \mathbf{T}$ is a refinement of machine $\mathbf{M}$ with data $\mathbf{u} : \mathbf{S}$, via a data refinement $\mathbf{R} : \mathbf{T} \rightarrow \mathbf{S}$, then (by definition of refinement in B) we have:

$$\begin{aligned} &\mathbf{Inv}_N(\mathbf{v}) \wedge \mathbf{Inv}_M(\mathbf{R}(\mathbf{v})) \wedge \mathbf{Pre}_{M, \mathbf{op}}(\mathbf{R}(\mathbf{v})) \Rightarrow \\ &\quad \mathbf{Pre}_{N, \mathbf{op}}(\mathbf{v}) \wedge [\mathbf{S}_{N, \mathbf{op}}[\mathbf{y}'/\mathbf{y}]] \neg [\mathbf{S}_{M, \mathbf{op}}] \neg (\mathbf{u} = \mathbf{R}(\mathbf{v}) \wedge \mathbf{y}' = \mathbf{y}) \end{aligned}$$

where $\mathbf{y}$ is the output variable of $\mathbf{op}$ (in both $\mathbf{M}$ and $\mathbf{N}$), and the definition of $\mathbf{op}$ is assumed to be of the form PRE $\mathbf{Pre}_{M, \mathbf{op}}$ THEN $\mathbf{S}_{M, \mathbf{op}}$ END in $\mathbf{M}$ and similarly in $\mathbf{N}$.

The theory interpretation from $\Gamma_{\mathbf{M}}$ to $\Gamma_{\mathbf{N}}$ is defined by mapping $\mathbf{u}_i$ to $\mathbf{R}_{\mathbf{u}_i}(\mathbf{v})$ (in practice, $\mathbf{R}$ will be defined by a collection of term interpretations $(\mathbf{R}_{\mathbf{u}_1}(\mathbf{v}), \dots, \mathbf{R}_{\mathbf{u}_i}(\mathbf{v}))$ where each $\mathbf{R}_{\mathbf{u}_i}(\mathbf{v})$ defines the representation of the abstract attribute $\mathbf{u}_i$ in terms of the concrete variables $\mathbf{v}$), and mapping action symbols to themselves. Since

$$\mathbf{S}_{\mathbf{N}, \mathbf{op}} \equiv \text{ANY } \mathbf{v}' \text{ WHERE } \mathbf{v}' \in \mathbf{V} \wedge \neg [\mathbf{S}_{\mathbf{N}, \mathbf{op}}](\mathbf{v}' \neq \mathbf{v}) \text{ THEN } \mathbf{v} := \mathbf{v}' \text{ END}$$

we also have:

$$\begin{aligned} & \mathbf{Inv}_{\mathbf{N}}(\mathbf{v}) \wedge \mathbf{Inv}_{\mathbf{M}}(\mathbf{R}(\mathbf{v})) \wedge \mathbf{Pre}_{\mathbf{M}, \mathbf{op}}(\mathbf{R}(\mathbf{v})) \Rightarrow \\ & \mathbf{Pre}_{\mathbf{N}, \mathbf{op}}(\mathbf{v}) \wedge \\ & \forall \mathbf{v}'. (\mathbf{v}' \in \mathbf{V} \wedge \neg [\mathbf{S}_{\mathbf{N}, \mathbf{op}}[\mathbf{y}'/\mathbf{y}]](\mathbf{v}' \neq \mathbf{v}) \Rightarrow \\ & \neg ([\mathbf{S}_{\mathbf{M}, \mathbf{op}}] \neg (\mathbf{u} = \mathbf{R}(\mathbf{v}) \wedge \mathbf{y}' = \mathbf{y}))[\mathbf{v}'/\mathbf{v}]) \end{aligned}$$

We can move the $\forall \mathbf{v}'$ quantifier to the outside of the main implication, and substitute $\bigcirc \mathbf{v}$ for $\mathbf{v}'$ to obtain:

$$\begin{aligned} & \mathbf{Inv}_{\mathbf{N}}(\mathbf{v}) \wedge \mathbf{Inv}_{\mathbf{M}}(\mathbf{R}(\mathbf{v})) \wedge \mathbf{Pre}_{\mathbf{M}, \mathbf{op}}(\mathbf{R}(\mathbf{v})) \wedge \bigcirc \mathbf{v} \in \mathbf{V} \wedge \neg ([\mathbf{S}_{\mathbf{N}, \mathbf{op}}[\mathbf{y}'/\mathbf{y}]](\mathbf{v}' \neq \mathbf{v}))[\bigcirc \mathbf{v}/\mathbf{v}'] \Rightarrow \\ & \neg [\mathbf{S}_{\mathbf{M}, \mathbf{op}}] \neg (\mathbf{u} = \bigcirc \mathbf{R}(\mathbf{v}) \wedge \mathbf{y}' = \mathbf{y}) \end{aligned}$$

since $\mathbf{R}(\bigcirc \mathbf{v}) = \bigcirc \mathbf{R}(\mathbf{v})$ as $\mathbf{R}$ is a constant transformation on terms.

But then substituting $\mathbf{R}(\mathbf{v})$ for $\mathbf{u}$ yields the required inference that the consequent of the axiom defining the effect of $\mathbf{op}$ in $\mathbf{N}$ implies that of the translation of the corresponding axiom in $\mathbf{M}$. But since

$$\mathbf{op} \wedge \mathbf{Pre}_{\mathbf{M}, \mathbf{op}}(\mathbf{R}(\mathbf{v})) \Rightarrow \mathbf{op} \wedge \mathbf{Pre}_{\mathbf{N}, \mathbf{op}}(\mathbf{v})$$

this means that the axiom for $\mathbf{op}$ in $\Gamma_{\mathbf{N}}$ establishes the translation of the axiom for $\mathbf{op}$ in $\Gamma_{\mathbf{M}}$, as required.

Other forms of axiom in the translation of $\mathbf{M}$ are directly provable in the translation of $\mathbf{N}$, so the result follows.

In practical terms this means that users of the extended language can carry out refinement steps either within the object calculus or using B.

Refinement is also compositional, i.e., if component $\mathbf{C}$ makes use of component $\mathbf{S}$, then refining $\mathbf{S}$ to $\mathbf{S}_1$ yields a refinement $\mathbf{C}'$ of $\mathbf{C}$ if $\mathbf{S}$ is replaced by $\mathbf{S}_1$ in $\mathbf{C}$ to produce $\mathbf{C}'$. Thus if we prove a property such as liveness of a system on the basis of the abstract specifications of its components, then this property will also be ensured in the eventual implementation, provided that refinement proofs have been carried out correctly.

5 Development Process

The development process which can be envisaged for this extended language is:

- formalise requirements using highly abstract declarative machine definitions, which describe the desired behaviour by means of temporal logic axioms and traces, with operations being listed but not specified in detail. The boundary of the system is established at this point. For safety critical systems this stage may also assign certain safety functions to hardware rather than to software;
- refine the abstract specifications using theory extension, towards a form where they can be translated into B machines. This involves partitioning the theory into parts which describe a main machine and possible subordinate machines (which will be used via SEES, USES, EXTENDS or INCLUDES in the main machine);

- translate into conventional B, possibly with threads. Guards are expressed by `SELECT` conditions. Information about the past history may need to be recorded in new attributes (eg, which give the value of `count(opi)` for some machine operations `opi`) to express these guards without the use of temporal operators.

Liveness and fairness constraints and other forms of predicate not covered by the list above are simply written in the `HISTORY` clause of the extended B AMN machine. They become instructions to the eventual implementor that these properties should be ensured (perhaps by means of process scheduling regimes at the operating system level), rather than being necessarily implementable via the B development process;

- carry out refinement and implementation in the extended B language – for the most part this is the same as in standard B, with the exception of the thread.

We will illustrate some of these steps for a standard example of a reactive system, the production cell case study [19].

6 Case Study: Production Cell

The production cell case study is described in [19]. It is a complex and realistic example of a discrete event control system, with required safety, liveness and timing properties. It is also an example of a *hybrid* system, that is, containing both discrete and continuous elements.

The components of the system are:

- A press, which responds to signals to close and open, and whose position can be determined;
- A robot, which can rotate clockwise or anti-clockwise, and which has two arms which can be separately extended or retracted. Each arm has an electromagnetic gripper at its end;
- A feed belt, which conveys metal blanks to the rotating table. It can be started and stopped, and it can be determined if a blank is at the end of the belt;
- a deposit belt, which transports work pieces unloaded by the robot to a travelling crane. The belt has a sensor to detect if a piece has reached the point where it can be picked up by the crane;
- a travelling crane, which picks up metal plates from the deposit belt, moving them to the feed belt and unloading them onto this belt;
- a rotating table, which is capable of vertical and rotary movement.

In this paper we will focus on the internal control of the robot, and the interaction between the robot and press. We will assume that the rotating table operates correctly and that the presence of a blank at the table may occur ‘spontaneously’ without being controllable from the robot/press subsystem. Thus the architecture of the subsystem is as shown in Figure 1, in the notation of Ward/Mellor RTSA [20]: data flows are indicated by solid lines, whilst dashed lines indicate event flows. The controller is a transformer on events rather than on data.

The safety requirements for the robot and press interaction and individual behaviour are as follows:

1. the robot must not be rotated further than necessary;
2. the robot arms must not be extended more than necessary for picking up or releasing items;
3. the press must not be moved downwards from its lower position, or upwards from its top position;
4. the press may only close when the robot arm is not positioned inside it;

Figure 1: Architecture of Robot/Press System

5. the robot arms must not drop metal blanks apart from onto the deposit belt and into the press when the press is open in its middle position;
6. no blank can be deposited in the press if the press already holds a blank;
7. a new blank may only be deposited by the robot on the deposit belt if this belt is ‘clear’ in a certain sense.

A liveness constraint is that every blank present at the end of the feed belt will eventually arrive, forged, at the beginning of the deposit belt.

An efficiency requirement is that concurrency within the system should be maximised.

The processing of the robot/press subsystem is as follows:

- when a blank appears on the rotary table, the robot picks it up with its first arm, then rotates anticlockwise until the second arm is pointing to the press (which must be in the lower position);
- the robot picks up a forged piece from the press with its second arm, then rotates anticlockwise until its second arm is pointing towards the deposit belt;
- the robot deposits the forged piece on the deposit belt, then rotates anticlockwise until the first arm is pointing to the press (which must be in the middle position – because the first and second arms are at different heights), and deposits the unforged piece in the press. It then rotates clockwise until its first arm is pointing towards the table.

The requirements for **Press** can be partly formalised as:

```

MACHINE Press
SEES PCTypes
VARIABLES
  pstate
INVARIANT
  pstate ∈ PSTATE
INITIALISATION
  
```

```

    pstate := middle
OPERATIONS
    close =
        pstate := upper;
    fully_open =
        pstate := lower;
    to_middle =
        pstate := middle
GUARDS
    close ⇒ pstate = middle;
    fully_open ⇒ pstate = upper;
    to_middle ⇒ pstate = lower
END

```

This abstracts from the precise control of the press movements via vertical motors. Such detail could be introduced in a refinement or implementation of this component.

PCTypes encapsulates basic type definitions:

```

MACHINE PCTypes
SETS
    VDIRECTION = { up, down };
    HDIRECTION = { out, in };
    RDIRECTION = { clockwise, anticlockwise };

    RASTATE = { extending, retracting, stationary };
    MAGSTATE = { holding, released };

    MSTATE = { on, off };
    PSTATE = { upper, middle, lower }
DEFINITIONS
    DISTANCE == N;
    ANGLE == 0..359
END

```

For components such as the robot arm, and for the robot itself, we need to represent in some way the continuous change which occurs as a result of the arm extension (or contraction) and the robot rotation. This is modelled by defining three separate actions for the commencement, continuation and termination of such a durative activity. As a result the **RobotArm** is specified by:

```

MACHINE RobotArm
INCLUDES Motor
SEES PCTypes
DEFINITIONS
    DISTANCE == N;
    ANGLE == 0..359
VARIABLES
    stop_position, extension, rastate, magstate
INVARIANT
    stop_position ∈ DISTANCE ∧
    extension ∈ DISTANCE ∧
    rastate ∈ RASTATE ∧
    magstate ∈ MAGSTATE ∧

    extension ≤ stop_position
INITIALISATION
    stop_position :∈ DISTANCE ||
    extension := 0 ||

```

```

    rastate := stationary ||
    magstate := released
OPERATIONS
    start_extend(dest) =
        PRE dest ∈ DISTANCE ∧ extension ≤ dest
        THEN
            stop_position := dest ||
            start_motor(out) ||
            rastate := extending
        END;

    continue_extend =
        ANY newpos
        WHERE
            newpos ∈ extension + 1 .. stop_position
        THEN
            extension := newpos
        END;

    stop_extend =
        BEGIN
            rastate := stationary ||
            stop_motor
        END;

    start_retract = ...;
    continue_retract = ...;
    stop_retract = ...;

    pick_up =
        magstate := holding;

    release =
        magstate := released

TRACES
    ExtendSequence = (start_extend; continue_extend*; stop_extend);

    RetractSequence = (start_retract; continue_retract*; stop_retract);

    PickUpRelease = (pick_up; release)*;

    GENERAL (ExtendSequence | RetractSequence | pick_up | release)* w PickUpRelease

```

The traces clause identifies that the histories of the machine must be of the form:

```

    ((start_extend; continue_extend*; stop_extend) |
     (start_retract; continue_retract*; stop_retract) |
     pick_up | release)*

```

and that they must also be compatible with (pick_up; release)*. w corresponds to concurrent composition of statecharts.

```

GUARDS
    continue_extend ⇒ extension < stop_position;
    stop_extend ⇒ extension = stop_position;

```



```

    continue_retract  $\Rightarrow$  extension > 0;
    stop_retract  $\Rightarrow$  extension = 0;

    pick_up  $\Rightarrow$  rastate = stationary  $\wedge$  magstate = released;
    release  $\Rightarrow$  rastate = stationary  $\wedge$  magstate = holding
HISTORY
    start_extend  $\Rightarrow$   $\diamond$ stop_extend  $\wedge$ 
    start_retract  $\Rightarrow$   $\diamond$ stop_retract
END

```

The guards for **pick_up** assert that it can only execute if the arm is stationary and is not currently holding an object. Similarly for **release**.

start_extend is a parameterised action – in the history clauses a mention of this action without any parameter refers to any instantiation of the action. The definitions of the operations for **release** actions are similar to those for **extend** and have been omitted.

The **Motor** is defined as a subordinate component of the **RobotArm**:

```

MACHINE Motor
SEES PCTypes
VARIABLES
    mstate
INVARIANT
    mstate  $\in$  MSTATE
INITIALISATION
    mstate := off
OPERATIONS
    start_motor(dir) =
        PRE dir  $\in$  HDIRECTION
        THEN
            mstate := on
        END;

    stop_motor =
        mstate := off
GUARDS
    stop_motor  $\Rightarrow$  mstate = on;
    start_motor  $\Rightarrow$  mstate = off
END

```

The intended history of the motor could be more abstractly specified as a trace **(start_motor; stop_motor)***.

Finally, the **Robot** itself is an aggregate of two (renamed) copies of **RobotArm**. We express it as an implementation in order to make use of sequential composition, although the data components have been presented in the same form as the abstract specification, for improved clarity¹. The use of **SELECT** as an “await” statement is exploited by statements of the form **arm1.start_retract; arm1.stop_retract**, whereby the flow of control of **Robot_1** is suspended at **arm1.stop_retract** until the retraction process has been completed.

```

IMPLEMENTATION Robot_1
REFINES Robot
SEES PCTypes
IMPORTS arm1.RobotArm, arm2.RobotArm,
        motor.RotaryMotor, dbelt.DepositBelt
DEFINITIONS
    DISTANCE == N;
    ANGLE == 0..359

```

---

¹ Declaration of variables of unstructured values within implementations is a proposed extension of B AMN.

```

CONSTANTS
  press1_rot, press2_rot, dbelt_rot,
  press_dist, dbelt_dist, table_dist
SETS
  RSTATE = { positioned_at_table, positioned_press_pickup,
              positioned_press_deposit, positioned_at_deposit,
              rotating_table_press, rotating_press_deposit,
              rotating_deposit_press, rotating_press_table }
PROPERTIES
  press1_rot ∈ ANGLE ∧ press2_rot ∈ ANGLE ∧
  dbelt_rot ∈ ANGLE ∧

  press2_rot < dbelt_rot ∧ dbelt_rot < press1_rot ∧
  press1_rot < 360 ∧

  press_dist ∈ DISTANCE ∧ dbelt_dist ∈ DISTANCE ∧
  table_dist ∈ DISTANCE
VARIABLES
  rstate, rotation
INVARIANT
  rstate ∈ RSTATE ∧
  rotation ∈ ANGLE ∧

  rotation ≤ press1_rot
INITIALISATION
  BEGIN
    rstate := positioned_at_table;
    rotation := 0
  END
OPERATIONS
  pickup_from_table =
    BEGIN
      arm1.start_extend(table_dist);
      arm1.stop_extend;
      arm1.pick_up;
      arm1.start_retract;
      arm1.stop_retract;
      rstate := rotating_table_press;
      motor.start_rotate(anticlockwise)
    END;

  pickup_from_press =
    BEGIN
      arm2.start_extend(press_dist);
      arm2.stop_extend;
      arm2.pick_up;
      arm2.start_retract;
      arm2.stop_retract;
      rstate := rotating_press_deposit;
      motor.start_rotate(anticlockwise)
    END;

  deposit_on_belt = ...; /* similar */

  deposit_at_press = ...;

```

```

continue_rotation =
  ANY newrot
  WHERE
    newrot ∈ 0..359 ∧
    (rstate = rotating_table_press ⇒
      newrot > rotation ∧
      newrot ≤ press2_rot) ∧
    (rstate = rotating_press_deposit ⇒
      newrot > rotation ∧
      newrot ≤ dbelt_rot) ∧
    (rstate = rotating_deposit_press ⇒
      newrot > rotation ∧
      newrot ≤ press1_rot) ∧
    (rstate = rotating_press_table ⇒
      newrot < rotation ∧
      newrot ≥ 0)
  THEN
    rotation := newrot
  END;

stop_rotation =
  BEGIN
    motor.stop_rotate;
    IF rstate = rotating_table_press
    THEN
      rstate := positioned_press_pickup
    ELSE
      IF rstate = rotating_press_deposit
      THEN
        rstate := positioned_at_deposit
      ELSE
        ...
      END
    END
  END
END

TRACES
  Rotate = ((pickup_from_table | pickup_from_press |
    deposit_on_belt | deposit_at_press); continue_rotation*;
    stop_rotation)*;

  MainSequence = (pickup_from_table; pickup_from_press;
    deposit_on_belt; deposit_at_press)*

  GENERAL Rotate w MainSequence

GUARDS
  pickup_from_table ⇒ rotation = 0 ∧
    rstate = positioned_at_table;
  pickup_from_press ⇒ rotation = press2_rot ∧
    rstate = positioned_press_pickup;

  deposit_on_belt ⇒ dbelt.dbelt_clear = TRUE ∧
    rstate = positioned_at_deposit ∧

```

```

    rotation = dbelt_rot;

    deposit_at_press ⇒ rstate = positioned_press_deposit ∧
                       rotation = press1_rot;

    continue_rotation ⇒
      (rstate = rotating_table_press ⇒ rotation < press2_rot) ∧
      (rstate = rotating_press_deposit ⇒ rotation < dbelt_rot) ∧
      (rstate = rotating_deposit_press ⇒ rotation < press1_rot) ∧
      (rstate = rotating_press_table ⇒ rotation > 0);

    stop_rotation ⇒
      (rstate = rotating_table_press ⇒ rotation = press2_rot) ∧
      (rstate = rotating_press_deposit ⇒ rotation = dbelt_rot) ∧
      (rstate = rotating_deposit_press ⇒ rotation = press1_rot) ∧
      (rstate = rotating_press_table ⇒ rotation = 0)
  HISTORY
    pickup_from_table ∨ pickup_from_press ∨
    deposit_on_belt ∨ deposit_at_press ⇒ ◇stop_rotation
  END

```

Preconditions $\text{arm1.extension} \leq \text{table.dist}$ for **pickup_from_table**, $\text{arm2.extension} \leq \text{dbelt.dist}$ for **deposit_on_belt**, $\text{arm2.extension} \leq \text{press.dist}$ for **pickup_from_press** and $\text{arm1.extension} \leq \text{press.dist}$ for **deposit_at_press** are assumed to be given in the specification of the robot, so do not need to be repeated in its implementation. These preconditions are needed in order to ensure the preconditions of the **start_extend** operations of the robot arms. However, they are in fact redundant given the trace and operation definitions of the robot, because the arms are initially fully retracted, and each robot operation that extends an arm also ensures that the arm is retracted before it completes.

Preconditions of an operation are used to express conditions that we expect to be true when the operation is invoked – in contrast a guard of an operation is used to specify waiting for a condition to become true before the operation is invoked. All the safety requirements of the system can be expressed by these constructs.

The overall liveness constraint that any blank which enters the system must eventually emerge can be partly expressed by the formula

$$\Box(\text{pickup\_from\_table} \Rightarrow \Diamond \text{deposit\_at\_press})$$

This can be proved in stages. We need the intermediate assertions that

$$\text{pickup\_from\_table} \Rightarrow \Diamond(\text{stop\_rotation} \wedge \text{rstate} = \text{positioned\_press\_pickup})$$

and similarly for the other pairs of successive actions of the robot and press subsystem. These follow from the HISTORY of the above machine and the definition of **stop_rotation**. Additionally, the properties that

$$\text{rstate} = \text{positioned\_press\_pickup} \Rightarrow \Diamond \text{pickup\_from\_press}$$

and similar assertions for the other actions, are needed. These can only be proved in the context of the machine which invokes operations from the **Robot**, given in the following section. A similar decomposition of system liveness into liveness constraints for individual components was also used in the statecharts case study of [19] as a way of rendering automated proof of these properties feasible.

A possible execution of the **pickup_from_table** operation, showing the different granularity of actions from different machines, is given in Figure 2. Concurrent execution of press actions is also shown. The single step **pickup_from_table** in the robot history thus corresponds to a series of steps in the individual components of the robot, starting with **arm1.start_extend** and completing with **motor.start_rotate**.

Figure 2: Concurrent Execution of Robot and Press

7 Refinement and Design

It is possible to translate the object calculus specification of the concurrent behaviour of a B machine given in the trace and guard clauses into specification elements entirely within the standard B AMN specification language, extended with threads.

Permission constraints can be interpreted by using the **SELECT** statement construct of B AMN. If **op** has permission constraints **per**₁, ..., **per**_n, then its definition in **M** will be of the form:

```

op =
  SELECT per1 ∧ ... ∧ pern
  THEN
    ⋮
  END

```

Alternatively, the select condition can be placed in the thread of an active module:

```

THREAD
  WHILE true
  DO
    SELECT
      per1 ∧ ... ∧ pern ANSWER op
    THEN
      ⋮
    END
  INVARIANT ...
END

```

Refinement can then proceed as in the B method, with laws for **SELECT** being used to carry out refinements of the thread. For example, strengthening the guard condition of a select clause is a refining transformation.

As an example of the above translation, the **Press** machine can be refined by:

```

REFINEMENT Press1
REFINES Press
SEES PCTypes
VARIABLES
  pstate
INVARIANT
  pstate ∈ PSTATE

```

```

INITIALISATION
  pstate := middle
OPERATIONS

  close =
    SELECT pstate = middle
    THEN
      pstate := upper
    END;

  fully_open =
    SELECT pstate = upper
    THEN
      pstate := lower
    END;

  to_middle =
    SELECT pstate = lower
    THEN
      pstate := middle
    END

END

```

The **RobotArm** machine can be implemented by:

```

IMPLEMENTATION RobotArm_1
REFINES RobotArm
IMPORTS Motor
SEES PCTypes
DEFINITIONS
  DISTANCE == N;
  ANGLE == 0.359
VARIABLES
  stop_position, extension, rastate, magstate
INVARIANT
  stop_position ∈ DISTANCE ∧ extension ∈ DISTANCE ∧
  rastate ∈ RASTATE ∧ magstate ∈ MAGSTATE ∧

  extension ≤ stop_position
INITIALISATION
  BEGIN
    stop_position := 0;
    extension := 0;
    rastate := stationary;
    magstate := released
  END
OPERATIONS

  start_extend(dest) =
    BEGIN
      start_motor(out);
      stop_position := dest;
      rastate := extending
    END;

```

```

stop_extend =
  BEGIN
    stop_motor;
    rastate := stationary
  END;

continue_extend =
  extension  $\leftarrow$  get_extension;

```

The retraction actions are defined in a similar manner.

```

pick_up =
  BEGIN
    rastate := holding
  END;

release =
  BEGIN
    rastate := released
  END

THREAD
  WHILE true
  DO
    SELECT
      rastate = stationary ANSWER start_extend
    THEN
      WHILE extension < stop_position
      DO
        continue_extend
      INVARIANT extension  $\leq$  stop_position
      VARIANT stop_position - extension
      END
    WHEN
      rastate = extending  $\wedge$ 
      extension = stop_position ANSWER stop_extend
    THEN
      SKIP
    WHEN
      rastate = stationary ANSWER start_retract
    THEN
      WHILE extension > 0
      DO
        continue_retract
      INVARIANT
        extension  $\geq$  0
      VARIANT
        extension
      END
    WHEN
      rastate = retracting  $\wedge$ 
      extension = 0 ANSWER stop_retract
    THEN
      SKIP
    WHEN rastate = stationary  $\wedge$ 

```

```

        magstate = released ANSWER pick_up
    THEN SKIP
    WHEN rastate = stationary ∧
        magstate = holding ANSWER release
    THEN SKIP
END
INVARIANT history ⊨ ...
END

```

END

get_extension is an operation of **Motor** which accesses a sensor which records how far the arm has been extended. A timing assumption which is made here is that the cycle time of the inner loops in the thread are low enough that it cannot be the case that **extension** < **stop_position** on one iteration and that **extension** > **stop_position** on the next (in the case of extension, and similarly for retraction).

The loop invariant consists of an assertion that the thread obeys the specification trace. This follows from the thread structure and the combination of guard conditions (ie., when the first select choice terminates, the second is the only possible successor, and so forth).

The above implementation features a mixture of passive and active behaviour, whereby the thread defines how external events (**start_extend**, **stop_extend**, etc) are responded to, but also defines autonomous behaviour (the polling loops used to control the arm motor).

A similar implementation strategy can be used for the robot. Finally the controller for the robot and press subsystem can be defined as:

```

IMPLEMENTATION RobotPressController_1
REFINES RobotPressController
IMPORTS
    table.Rotary_table,
    robot.Robot,
    press.Press
SEES PCTypes, Bool_TYPE
OPERATIONS
    process_blank =
        BEGIN
            BEGIN
                BEGIN
                    robot.pickup_from_table; robot.stop_rotation
                END |||
                BEGIN press.close; press.fully_open
                END
            END;
            robot.pickup_from_press;
            robot.stop_rotation;
            BEGIN
                press.to_middle |||
                BEGIN
                    robot.deposit_on_belt;
                    robot.stop_rotation
                END
            END;
            robot.deposit_at_press;
            robot.stop_rotation
        END
THREAD

```



```

VAR bpres, robstate, pressstate
IN
  bpres ← table.blank_present;
  robstate ← robot.robot_state;
  pressstate ← press.press_state;
  WHILE true
  DO
    IF bpres = TRUE ∧
       robstate = positioned_at_table ∧
       pressstate = middle
    THEN
      process_blank
    END;
    bpres ← table.blank_present;
    robstate ← robot.robot_state;
    pressstate ← press.press_state
  INVARIANT ...
END
END
END

```

assuming the existence of suitable enquiry operations of **Rotary_table**, **Robot** and **Press**. $|||$ denotes an operation combination construct which defines independent execution of two statements, without interference or communication. It can be defined in terms of asynchronous calling. Some assumptions about the relative time of execution of operations are essential here:

- that **press.close** always completes before the robot begins its rotation towards the press from the table;
- that **press.to_middle** always completes before the robot begins its rotation from the belt to the press.

These assumptions are reasonable, but could be avoided by further decomposition of the robot operations.

8 Comparisons with Other Approaches

The extensions to B proposed here retain its modular and model-based character, yielding a well-defined concept of (verifiable) refinement within a single language. In contrast, for languages such as LOTOS, it has not been clear what is the correct definition of bisimulation to use for refinement [13], resulting in a “dual language” approach whereby logic is used to specify required properties and which are then verified for particular LOTOS process descriptions. The abstraction level in the B specifications can be very high, as shown in the above example, where continuous change in controlled variables was modelled in the initial specifications. In the refinement process such abstractions can be replaced by more concrete representations, such as discrete value spaces (**DISTANCE**) for arm extensions and inter-machine distances.

B has a comparable or stronger concept of modularity than languages such as ESTEREL [2] or LUSTRE, and allows higher-level abstractions than these languages. This expressiveness however makes automated verification more difficult, and work is needed to improve the proof capabilities of the B Toolkit by incorporating decision procedures for decidable domains. The use of PVS for B has been investigated [22]. Extended B provides better facilities for expressing liveness properties than RSL or statecharts, in the approaches of [19]. The language facilities proposed for extended B are based on features of VDM⁺⁺ which have undergone substantial critical review and testing in industrial case studies.

9 Conclusions

We have described some extensions of the B specification language for the description of properties of concurrent systems.

Future extensions of this work will address how the B Toolkit can be extended to support the definition of refinement as theory extension, and the animation of temporal logic specifications via event checking [17]. Connections with action system formalisms [4] are also being investigated.

Timing specifications can be added to B using the approach of [14].

Acknowledgement

The work described here was carried out in conjunction with B-Core (UK) Ltd., Oxford Science Park, Oxford OX4 4GA, and in particular with Ib Sørensen and Dave Neilson.

References

- [1] J.-R. Abrial. *The B Book*, Cambridge University Press, 1995, to appear.
- [2] F. Boussinot and R. de Simone. *The Esterel Language. Another Look at Real-time Programming*, Proceedings of the IEEE, 79(9): 1293–1304, 1991.
- [3] J. Bowen and V. Stavridou. *Safety-critical systems, formal methods and standards*, Software Engineering Journal, 8(4):189–209, July 1993.
- [4] M. Butler. *Stepwise Refinement of Communicating Systems*, Technical Report, Dept. of Computing, Southampton University, 1996.
- [5] P. Chapront. Vital coded processor and safety related software design. In H H Frey, editor, *Safety of Computer Control Systems 1992 (SAFECOMP '92), Computer Systems in Safety Critical Applications, Proc IFAC Symp.*, pages 141 – 145. Pergamon Press, 1992.
- [6] C. DaSilva, B. Dehbonei, and F. Mejia. Formal specification in the development of industrial applications: The subway speed control mechanism. In *FORTE '91*, pages 207 – 221. North-Holland, 1991.
- [7] R. Duke, P. King, G. Smith. Formalising Behavioural Compatibility for Reactive Object-oriented Systems, *Proc 14th Australian Compt. Sci. Conf. (ACSC-14)*, 1991.
- [8] E. Durr and E. Dusink. The role of VDM⁺⁺ in the development of a real-time tracking and tracing system. In J Woodcock and P Larsen, editors, *FME '93, Lecture Notes in Computer Science*. Springer-Verlag, 1993.
- [9] J. Fiadeiro and T. Maibaum. *Temporal Theories as Modularisation Units for Concurrent System Specification*, Formal Aspects of Computing 4(3), pp. 239–272, 1992
- [10] J. Fiadeiro and T. Maibaum. *Interconnecting formalisms: supporting modularity, reuse and incrementality*, in Proc. 3rd Symposium on the Foundations of Software Engineering, ACM Press, 1995, to appear.
- [11] C. George, P. Haff, K. Havelund, A. E. Haxthausen, R. Milne, C. B. Nielsen, S. Prehn, and K. R. Wagner. *The RAISE Specification Language*. Prentice Hall, 1992.
- [12] J. Hoare. *The Use of B in CICS*, in **Applications of Formal Methods**, J. Bowen and M. Hinchey (Eds.), Prentice Hall, 1995.

- [13] C. Kirkwood. *Specifying Properties of Basic LOTOS processes using Temporal Logic*, in FORTE '95 proceedings, to appear, 1995.
- [14] K. Lano. *Reactive System Specification and Refinement*, Proceedings of TAPSOFT '95, Springer-Verlag LNCS, 1995.
- [15] K. Lano, H. Haughton. *Formal Development in B AMN*, Information and Software Technology, Vol. 37, No. 5-6, 1995, 303-316.
- [16] K. Lano. *Reasoning Techniques in VDM⁺⁺*, AFRODITE Project report AFRO/IC/KL/RT/V1, 1994.
- [17] K. Lano. *Formal Object-oriented Development*, FACIT series, Springer-Verlag, 1995.
- [18] K. Lano, J. Fiadeiro, J. Dick. *Extending B AMN with Concurrency*, Theory and Formal Methods Workshop, Imperial College, 1996.
- [19] C. Lewerentz, T. Lindner (Editors). *Case Study "Production Cell": A comparative study in formal software development*, FZI Publication 1/94, University of Karlsruhe, 1994. To appear in LNCS, Springer-Verlag, 1996.
- [20] S. Mellor and P. Ward. *Structured Development for Real-time Systems (3 Volumes)*. Yourdon Press, 1985.
- [21] A. Pnueli. Applications of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends, *Current Trends in Concurrency*, de Bakker J., de Roever W.-P., Rozenberg G. (Eds), Springer-Verlag Lecture Notes in Computer Science, Vol. 224, 1986.
- [22] C. H. Pratten. An Introduction to Proving AMN Specifications with PVS and the AMN-Proof Tool, B User Group Workshop, Paris, 1995.
- [23] J. van de Snepscheunt. Trace Theory and VSLI Design, PhD thesis, Technische Logeschool Eindhoven, Eindhoven, The Netherlands, 1983.
- [24] M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 2nd edition, 1992.