Towards Dynamically Communicating Abstract Machines in the B Method

Nazareno Aguirre[1], Marcelo Arroyo[1], Juan Bicarregui[2], Lucio Guzmán[1], and Tom Maibaum[3]

[1] Departamento de Computación, FCEFQyN,
Universidad Nacional de Río Cuarto,
Ruta 36 Km. 601, Río Cuarto (5800), Córdoba, Argentina
{naguirre, marroyo, lucio}@dc.exa.unrc.edu.ar

[2] Rutherford Appleton Laboratory, Chilton, Didcot,
OXON, OX11 0QX, United Kingdom
J.C.Bicarregui@rl.ac.uk

[3] Department of Computing & Software, McMaster University,
1280 Main St. West, Hamilton, Ontario, Canada L8S 4K1
tom@maibaum.org

Abstract. In this paper we present an attempt to represent dynamic communication links between abstract machines in the B method. The approach complements a previously proposed extension to B, that supports dynamic creation and deletion of machine instances, providing a mechanism for dynamically connecting or disconnecting machine instances for communication. This mechanism is based on the concept of *connector*, in the software architectures sense.

We propose an extension to B's notation to support the definition of connectors. The extension has been defined with the intention of making it fully compatible with the standard B method, and allows one to enable communication, under certain restrictions, between abstract machines in a specification which presents dynamic creation and deletion of machine instances. We present the extension, its semantics and an example illustrating its use based on a producer-consumer specification. We also discuss possible ways of extending the proposed connector definitions to more general forms of communication.

Keywords: B method, structuring mechanisms, dynamic reconfiguration, object orientation.

1 Introduction

The B formal specification language is one of the most successful model based formalisms for software specification. It has an associated method, the B Method [1], and commercial tool support, including proof assistance [5][7]. As all formal methods, the B method provides a formal language in which one can describe systems, allowing for analysis and verification of certain system properties prior to implementation. Moreover, the B method and its associated tools also cover refinement, implementation and code generation [5][7].

Various facilities for structuring specifications are provided in B, helping to make the specification and refinement activities scalable. However, the B method has an important restriction regarding structuring mechanisms, namely, it does not provide, as a structuring feature, the dynamic creation and deletion of modules or components. More precisely, all structuring mechanisms of B are *static*, in the sense that they allow one to define abstract machines whose architectural structure in terms of other components is fixed, i.e., it does not change at run time [8].

In this context of *static* configurations of abstract machines in the B method, communication between machines is achieved, essentially, in one of the following ways:

– either one of the machines is contained in the other (and therefore the second can access information from the first one respecting the visibility rules of the corresponding structuring mechanism employed), or
– a common substructure of these machines is "factored away" as a separate machine, and is shared in a "one writer-many readers" fashion [1]. Note that in this case the communication is in an *asynchronous* mode.

As we mentioned, B lacks dynamic management of abstract machines. Dynamic management of the population of components is a feature often associated with object oriented languages, since the replication of *objects* is intrinsic to these languages [12]. In fact, dynamic management of "objects" is currently accepted as a common software design practice, perhaps due to the success of object oriented methodologies and programming languages. In order to allow for dynamic management of components in B, it is not necessarily a good approach to extend B to support fully fledged object orientation, since this would imply a significant change to B's neat syntax and semantics, and would excessively complicate the tool support implementation (especially in relation to proof support). Nevertheless, we have been engaged in the development of extensions to the B method to support dynamic creation and deletion of machines [2][3], but we have done so trying to make the extensions fully compatible with the standard B method. Indeed, we have complemented B's structuring mechanisms with an extra clause, the AGGREGATES. A clause AGGREGATES M' within the definition of a machine M intuitively indicates that M counts on a dynamic set of *instances* of machines of type M', which can be created or deleted at run time [2]. Moreover, we have also shown how this clause can be treated at the refinement and implementation stages of the B method [3].

As we have advocated, having a structuring mechanism that allows for the dynamic creation/deletion of machine instances can favour the structuring of specifications, and therefore, contribute to the decomposability of proof obligations and the understandability of system specifications [2]. However, allowing for the dynamic creation and deletion of machine instances via the use of the AGGREGATES clause restricts the applicability of the above mentioned approaches for communicating machines. This is not surprising, since these were designed for static architectural configurations of abstract machines. In fact, when trying to achieve communication between dynamically generated machines (i.e., in "dynamic architecture scenarios"), one usually ends up building complex and unstructured specifications. The complexities associated with these specifications are related to the fact that the specifier has to manually construct the machinery for dynamic creation of "objects", for managing the communication, etc, usually all encapsulated in a single, *flat* abstract machine. Thus, we propose an alternative, based on an extension to B's notation to support the definition of *connectors* [4]. The extension is built on top of standard B, i.e., specifications written using the extension can be systematically translated into standard B specifications. The extension we propose here allows one to define synchronous communication between abstract machines in a specification with dynamic creation and deletion of machine instances. The mechanism we present is limited to some specific kinds of communications, but, as we will also show, more general forms of communication can also be characterised (although these need to be *asynchronous*). As for our previously proposed extensions, this extension has been defined with the intention of making it fully compatible with the standard B method. We present the extension, its semantics and an example illustrating its use based on a producer-consumer specification.

The remainder of this paper is organised as follows: In Section 2 we start by showing how machines are typically communicated (statically) in B. We argue about the unsuitability of these mechanisms when combined with dynamic aggregations of machines (Section 2.1). We then briefly describe the AGGREGATES structuring mechanism, its use and semantics. We show how a system with dynamic creation and deletion of components can be specified using aggregation, but these cannot be connected for communication using B's mechanisms (Section 2.2). In Section 3 we present the syntax we propose for connectors in B, and connectors' intuitive meaning. In Section 4 we are engaged with the description of the semantics of connectors, in terms of standard B constructs. We discuss the proof obligations associated with connector definitions (Section 4.2), and show how connectors are used by means of an example. We end Section 4 enumerating some of the limitations that our connectors have for defining communication between abstract machines, and discuss the characterisation of more general forms of communication. Finally, in Section 5 we present our conclusions, some comparison with related work and lines for future work.

2 Communicating Abstract Machines in the B Method

Let us introduce the problems that we attempt to solve (at least partially) in this paper by means of an example. This example is a simple variant of a component based specification given in [9] for the producer-consumer problem. Let us suppose that we need to specify a system consisting of a producer that sends "products" to a consumer. Assuming that the products are encoded as non-zero integer numbers, and choosing to specify the producer and the consumer as separate machines, one might define a basic machine *Channel* in order to

```
MACHINE
    Channel
VARIABLES
    var
INVARIANT
    $var \in \mathsf{INT}$
INITIALISATION
    $var := 0$
OPERATIONS
    $set(i) \;\hat{=}\;$ PRE $i \in \mathsf{INT}$ THEN $var := i$ END;
    $x \longleftarrow get \;\hat{=}\;$ BEGIN $x := var$ END
END
```

Fig. 1. A simple abstract machine used for communication between a producer and a consumer

"implement" the communication, as in Fig. 1. Then, we can define the producer and the consumer as structured definitions on top of machine *Channel*, as shown in Figs. 2 and 3. Note that, for the sake of simplicity, we do not model the corresponding acknowledgement that the consumer should send after consuming a product (note that without the acknowledgement, a producer can overwrite a product before the consumer gets it). Such acknowledgements can be easily "implemented" in a structured way, by taking a renamed copy of *Channel*, say *Channel'*, and use it for "backward communication", i.e., the consumer writes on it (machine *Consumer* includes *Channel'*) and the producer reads it (machine *Producer* sees *Channel'*). Notice that machine *Producer* simultaneously includes *Channel* (for forward communication) and sees *Channel'* (for backward communication), and the converse is true for machine *Consumer*.

This corresponds to one of the alternatives that a specifier has for specifying the system, with producer and consumer as separate machines. Of course, one could also decide to specify the whole system as a sole, flat abstract machine; following this latter approach, although sound, does not favour the decomposition of the proofs of consistency (i.e., the proofs of the proof obligations) nor the understandability of the specification (more detail on this in Section 5).

```
MACHINE
    Producer
INCLUDES
    Channel
VARIABLES
    p-var
INVARIANT
    $p\text{-}var \in \mathsf{INT}$
INITIALISATION
    $p\text{-}var := 0$
OPERATIONS
    $prod(x) \;\hat{=}\;$ PRE $x \neq 0 \wedge p\text{-}var = 0$ THEN $p\text{-}var := x$ END;
    $send \;\hat{=}\;$ PRE $p\text{-}var \neq 0$ THEN $p\text{-}var := 0 \;||\; set(p\text{-}var)$ END
END
```

Fig. 2. A specification of *Producer* including a channel

```
MACHINE
    Consumer
SEES
    Channel
VARIABLES
    c-var
INVARIANT
    c-var ∈ INT
INITIALISATION
    c-var := 0
OPERATIONS
    obtain  ≙  PRE var ≠ 0 ∧ c-var = 0 THEN c-var := var END;
    cons  ≙  PRE c-var ≠ 0 THEN c-var := 0 END
END
```

Fig. 3. A specification of *Consumer* “seeing” a channel

2.1 Communication in the Presence of Aggregation

Let us now suppose that we need to specify a system with a varying number of producers and consumers, and with dynamism in the communication. For instance, suppose that, besides dynamically creating and deleting producers and consumers, we want to change dynamically the consumer a particular producer produces for. In order to specify such a system, we are unable to use the option of a common machine, such as *Channel*, for implementing the communication, since changing the consumer to which a producer is “connected” would not be possible to specify (it would require a dynamic change in the structural organisation of the system specification, not supported in B). So, we are left with a flat, unstructured machine as the only option for specifying this system, at least if we use standard B.

However, in this paper, we will show how we can specify the above described dynamic system of producers and consumers in a *structured* way, by using the AGGREGATES clause. As we previously described, this structuring mechanism intuitively allows one to manipulate a dynamic set of instances of the aggregated machine. Of course, we cannot enable the communication between these machines directly, so we first have to consider the uncommunicating versions of producer and consumer, as shown in Fig. 4. (We will “connect” these machines later.) Then, we can easily obtain a system with dynamic populations of (uncommunicating) producers and consumers, by aggregating these machines, as in Fig. 5. Note that, in this case, we are able to create and delete producers and consumers dynamically, but there is no interaction/communication between producer instances and consumer instances.

2.2 The AGGREGATES Clause

The aggregation of an abstract machine relies on the (systematic) generation of a population manager for the aggregated machine. A population manager of an abstract machine M puts together the relativisation of the operations of M (so they work for multiple instances) with operations that mimic the creation and deletion of machines instances. To illustrate the use of AGGREGATES, and its semantics in terms of a population manager, consider the simple machine

```
MACHINE                              MACHINE
    Producer                             Consumer
VARIABLES                            VARIABLES
    p-var                                c-var
INVARIANT                            INVARIANT
    p-var ∈ INT                          c-var ∈ INT
INITIALISATION                       INITIALISATION
    p-var := 0                           c-var := 0
OPERATIONS                           OPERATIONS
    prod(x)  ≙                           obtain(x)  ≙
        PRE   p-var = 0 ∧ x ∈ INT−{0}        PRE   c-var = 0 ∧ x ∈ INT−{0}
        THEN  p-var := x                     THEN  c-var := x
        END;                                 END;
    send()  ≙                            cons()  ≙
        PRE   p-var ≠ 0                      PRE   c-var ≠ 0
        THEN  p-var := 0                     THEN  c-var := 0
        END                                  END
END                                  END
```

Fig. 4. Abstract Machines *Producer* and *Consumer*

```
MACHINE
    SYSTEM
AGGREGATES
    PRODUCER, CONSUMER
  .
  .
  .
END
```

Fig. 5. Fragment of a machine aggregating *Producer* and *Consumer*

Producer that we show in Fig. 4. The corresponding population manager for this abstract machine is shown in Figs. 6. Notice that, due to the automatic generation of the managers, some conjuncts in the invariants are redundant. For a detailed explanation of how these machines are synthesised see [2] .

The "AGGREGATES *Producer*" in the abstract machine *System* in Fig. 5 is simply interpreted as "EXTENDS *ProducerManager*". Within *System* we can manage the population of producers (resp. consumers) by invoking the implicitly defined *add_Producer* (resp. *add_Consumer*) and *del_Producer* (resp. *del_Consumer*). Furthermore, we can call *Producer* (resp. *Consumer*) operations, now operating on particular live instances. Consider, for instance, a *deliver* operation that enforces a live instance of *Consumer* to consume the product produced by a live *Producer p*.

```
deliver(p, c) ≙
    PRE
        p ∈ ProducerSet ∧ c ∈ ConsumerSet ∧ p.p-var ≠ 0 ∧ c.c-var = 0
    THEN
        p.send || c.obtain(p.p-var)
    END
```

Note that the AGGREGATES clause constitutes in effect a structuring mechanism: the consistency of machines aggregating other machines can be reduced to the consistency of the aggregate, and a number of further conditions on the aggregating machine. This is the case thanks to the fact that the population manager of a machine M is internally consistent *by construction*[1], provided that M is internally consistent. Then, the population manager of a basic machine M is *automatically constructed* and can be hidden from the specifier, who can use AGGREGATES as other conventional structuring mechanism of B [2]. Moreover, the population managers can be hidden from the developer even during refinement and implementation [3].

```
MACHINE
    $Producer\_Manager$
VARIABLES
    $p\text{-}var, ProducerSet$
INVARIANT
    $(\forall n \cdot n \in ProducerSet \Rightarrow p\text{-}var(n) \in \mathsf{INT}) \wedge$
    $(ProducerSet \subseteq \mathsf{NAME}) \wedge (p\text{-}var \in ProducerSet \rightarrow \mathsf{INT})$
INITIALISATION
    $p\text{-}var, ProducerSet := \emptyset, \emptyset$
OPERATIONS
    $prod(x, n) \;\hat{=}$
        PRE  $n \in ProducerSet \wedge x \in \mathsf{INT} - \{0\} \wedge p\text{-}var(n) = 0$
        THEN  $p\text{-}var(n) := x$
        END;
    $send(n) \;\hat{=}$
        PRE  $n \in ProducerSet \wedge p\text{-}var(n) \neq 0$
        THEN  $p\text{-}var(n) := 0$
        END;
    $add\_Producer(n) \;\hat{=}$
        PRE  $n \in \mathsf{NAME} - ProducerSet$
        THEN
            $ProducerSet := ProducerSet \cup \{n\} \,||\, p\text{-}var := p\text{-}var \cup \{n, 0\}$
        END;
    $del\_Producer(n) \;\hat{=}$
        PRE  $n \in ProducerSet$
        THEN
            $p\text{-}var := \{n\} \lhd p\text{-}var \;||$
            $ProducerSet := ProducerSet - \{n\}$
        END;
END
```

Fig. 6. *ProducerManager*: Population manager of abstract machine *Producer*

We adopt the "dot notation" to access variables and operations of the aggregated machines. For instance, the expression $m.y$ represents the value of variable y corresponding to instance m; analogously, the expression $m.op(x)$ represents a "call" to operation op, with argument x, corresponding to instance m (see the definition of operation *deliver*). The prefix of an expression employing the dot notation simply represents the "instance parameter" of the corresponding operation or variable (i.e., $m.y$ and $m.op(x)$ are convenient ways of writing $y(m)$ and $op(x,m)$, respectively).

[1] An abstract machine is *internally consistent* if it satisfies its *proof obligations*, i.e. if it satisfies the requirements imposed in the B method for considering a machine correct. Proof obligations for the correctness of an abstract machine include conditions such as nonemptiness of the state space determined by the machine, or the preservation of the machine's invariant by the machine's operations [1].

3 Communicating Dynamically Generated Machines

We now consider how to enable communication between dynamically generated machines. We propose to extend B's notation with the definition of *connectors* [4]. In software architectures [13], connectors represent communication "links" between components, and have the particularity of being external to the definition of the related components. This is precisely what we need, a mechanism for relating machine instances outside the definition of the interacting machines.

3.1 Connector Definitions in B

A connector consists of: *(i)* a name, *(ii)* a pair of participants (names of abstract machines), and *(iii)* a list of *connections*. The purpose of connections is to define how the operations of the participants are linked when two instances of the participants are connected. Consider, for instance, the connector definition in Fig. 7. This connector indicates that, whenever an instance p of *Producer* is connected to an instance c of *Consumer* via R, then the occurrence of $p.send()$ forces the occurrence of (or makes a call to) $c.obtain(p.p\text{-}var)$.

A connection can have one of the following forms:

$$\begin{array}{c} op1 \rightarrow op2 \\ op1 \leftarrow op2 \end{array}$$

where *op1* is an operation of the first participant and *op2* is an operation of the second one. The intended meaning of $op1 \rightarrow op2$ is that *op1* "calls" *op2*; the connection in the other direction is interpreted in a similar way. Subsequently, given a connection c, we will denote by $src(c)$ the operation at the source of the arrow, and by $tgt(c)$ the operation at the target of the arrow.

For a connector R between two machines M_1 and M_2 to be well defined, we have a number of syntactic conditions:

- Machines M_1 and M_2 must be unrelated (i. e., M_1 cannot be defined in terms of M_2 and vice versa),
- if we have a connection $op1 \rightarrow op2$ (resp. $op1 \leftarrow op2$), then *op1* must be an operation of M_1 and *op2* an operation of M_2,
- if we have a connection c, then the formal parameters of $src(c)$ must be distinct variables different from the state variables of M_1 and M_2, and the parameters of $tgt(c)$ can only be linguistic elements of M_1 or M_2, or formal parameters of $src(c)$,
- a connector cannot contain two different connections $c1$ and $c2$ such that $src(c1) = src(c2)$.

We need to enforce these conditions to guarantee that we can build the machinery for representing connections in B, as we will show in the next section.

3.2 Intuitive Meaning of Connectors

Connectors allow us to intuitively define relations between instances of abstract machines. These relations also define an interaction between the related instances, described by the connections. In order to clarify how connectors can

```
CONNECTOR
    R
PARTICIPANTS
    Producer, Consumer
CONNECTIONS
    send() → obtain(p-var)
END
```

Fig. 7. A connector for communicating producers and consumers

```
MACHINE
    Prod_Cons
AGGREGATES
    Producer, Consumer WITH R
OPERATIONS
    feedback(c, p)  ≙
        PRE  c ∈ ConsumerSet ∧ p ∈ ProducerSet ∧ (p, c) ∈ RSet ∧
        p.p-var = 0 ∧ c.c-var ≠ 0
        THEN  c.cons() || p.prod(c.c-var)
        END
END
```

Fig. 8. An example of a machine with aggregation and connectors

be used in practice, let us define an abstract machine *Prod_Cons* with dynamic sets of producers and consumers related by R, shown in Fig. 8. Within machine *Prod_Cons*, one can manage the population of producers and consumers as explained before; furthermore, one can dynamically connect and disconnect instances of producers and consumers by using two implicitly defined operations, $connect\_R(x, y)$ and $disconnect\_R(x, y)$. As for the case of the AGGREGATES clause, connectors have an "instance set"; for our example, the instance set corresponding to connector R is denoted by $RSet$ (see the definition of operation $feedback$).

Connector R then defines a relation between instances of producers and instances of consumers. Besides this relation, connector R has an important effect on the operations linked by R: whenever an instance p of *Producer* is related to an instance c of *Consumer*, a call to $p.send()$ enforces a synchronous call to $c.obtain(p.p\text{-}var)$, as the connection of R indicates. If a producer p is not connected to a consumer c, then the effect of $p.send()$ is not altered.

When a producer is connected to a consumer, we need to call the *obtain* operation on the corresponding consumer; therefore, we need to unequivocally determine which one is the consumer we have to call. So, we need to restrict the relation $RSet$ to be *functional*. Conversely, when we have a connection on the other direction (e.g., from consumer to producer), we have to restrict the relation $RSet$ to be *injective*. These restrictions will be clarified in the next section, regarding the semantics of connectors.

4 Semantics of Connectors

As we mentioned, it is our aim to extend the B method to support more sophisticated mechanisms for specification, but we want to do so in a way compatible with standard B. This would ensure that one could remain using the extensive work developed on B, as well as the traditional tool support. So, we have the same intention with our extension supporting connectors. Our connector definitions can be mapped into specifications in standard B. More precisely, we "implement" support for connectors as standard B specifications.

We build an abstract machine for the definition of a connector R relating two machine (types) M_1 and M_2. This machine, that we call $R\_Manager$, is structurally defined in terms of the managers for M_1 and M_2. Moreover, this machine contains the redefinition of those operations involved in connections. Then, a clause "AGGREGATES M_1, M_2 WITH R" within a machine M is simply interpreted as "EXTENDS $R\_Manager$." Note that $R\_Manager$ contains the population managers for M_1 and M_2.

4.1 Building the Connector Manager

Let M_1 and M_2 be two internally consistent abstract machines, and R a syntactically valid connector (see the syntactic conditions for valid connectors in the previous section) with M_1 and M_2 as participants. Let us assume that there are no name clashes between the definitions within M_1 and M_2 (note that name clashes can be avoided via renaming). We start by constructing the managers $M_1\_Manager$ and $M_2\_Manager$, for M_1 and M_2. Since machines M_1 and M_2 are internally consistent, we can ensure $M_1\_Manager$ and $M_2\_Manager$ are also consistent [2]. Then, we "prime" the operations in $M_1\_Manager$ and $M_2\_Manager$, and call the resulting machines $M_1\_Manager'$ and $M_2\_Manager'$, respectively. The priming is necessary, because we will need to redefine some of the operations (those involved in connections) of the related machines.

The general form of machine $R\_Manager$ is shown in Fig. 9. As it can be seen, this machine has a variable $RSet$, which represents the sets of active connectors. As is forced by the invariant, only instances of the corresponding participants can be connected. If the definition of R includes connections of the form $op1 \rightarrow op2$, then the invariant is complemented with $RSet \in M_1Set \mapsto\!\!\!\!\!+\ M_2Set$; if the definition of R includes connections of the form $op1 \leftarrow op2$, then the invariant is complemented with $RSet^{-1} \in M_1Set \mapsto\!\!\!\!\!+\ M_2Set$ (note that a given connector can have both types of connections).

The operation for disconnecting instances is easily defined. On the other hand, the operation for connecting machines depends on the types of connections. If a connection of type $op1 \rightarrow op2$ is present in the definition of R, then $R\_connect$ has to preserve the functionality of $RSet$; on the other hand, if a connection of type $op1 \leftarrow op2$ is present in the definition of R, then $R\_connect$ has to preserve the injectivity of $RSet$. This gives us four possible definitions for $R\_connect$, depending on the types of connections present. For instance, if we only have connections of type $op1 \rightarrow op2$, then $R\_connect$ is defined as follows:

$$\begin{array}{l} R\_connect(x, y) \ \hat{=} \\ \quad \text{PRE} \quad x \in M_1Set \wedge y \in M_2Set \quad \text{THEN} \quad RSet(x) := y \quad \text{END} \end{array}$$

The other three possibilities are also easily defined, and are left as an exercise for the interested reader.

```
MACHINE
    $R\_Manager$
EXTENDS
    $M_1\_Manager'$, $M_2\_Manager'$
VARIABLES
    $RSet$
INVARIANT
    $RSet \subseteq M_1Set \times M_2Set \ \wedge \ \ldots$
INITIALISATION
    $RSet := \emptyset$
OPERATIONS
    $R\_disconnect(x, y) \ \hat{=}$
        PRE $(x, y) \in RSet$ THEN $RSet := RSet - \{(x, y)\}$ END
    $R\_connect(x, y) \ \hat{=} \ \ldots$
    $op1(x) \ \hat{=}$
        PRE $pre(op1'(x))$ THEN
        $(RSet[\{x\}] \neq \emptyset \Longrightarrow op1'(x) || op2'(RSet(x)))\,[]\,(RSet[\{x\}] = \emptyset \Longrightarrow op1'(x))$ END
    $op3(x) \ \hat{=}$ PRE $pre(op3'(x))$ THEN $op3'(x)$ END
    ⋮
END
```

Fig. 9. The general form of a connector manager for machines M_1 and M_2

Notice the definition for operation *op1*(x) in Fig. 9. This definition assumes that we have a connection *op1* $\rightarrow$ *op2*. It has as a precondition the precondition of the original *op1*(x) operation (now primed). However, its effect, as shown in the THEN section of its definition, depends on whether x has a connected instance of M_2 or not. In the case it has a connected instance, the operation *op2*$'$ is called on the connected instance $R(x)$ (here it becomes clearer why we need to restrict *RSet* to be functional/injective), in parallel with *op1'*(x); in the case that x has no connected instance, we simply call *op1'*(x).

The definition of *op3*(x) in Fig. 9 assumes that *op3* is not at the source of a connection of R, and therefore, it only needs to call *op3*$'(x)$.

4.2 Proof Obligations for Connector Definitions

The initialisation of the connector manager trivially respects the machine's invariant (since the empty relation is both injective and functional). Also, operations *R_connect* and *R_disconnect* are defined to make them comply with the machine's invariant (as we said, *R_disconnect* trivially preserves the invariant, whereas *R_connect* has four possible definitions, according to the types of connections present). However, some proof obligations which cannot be automatically discharged will be generated from the connector manager. These have to do with the way in which the original operations (the primed ones) are called. Whenever the connector contains a connection *op1* $\rightarrow$ *op2*, the definition of *op1* in *R_Manager* invokes *op1'* respecting its precondition; however, it is not guaranteed by construction that the call to *op2'* within the definition of *op1* respects the precondition of *op2'*, and therefore this will have to be proved.

Essentially, these proof obligations force us to prove that, whenever we have a connection *op1* $\rightarrow$ *op2* (resp. *op1* $\leftarrow$ *op2*), we guarantee that the precondition of *op2* (resp. *op1*) is *subsumed* by the precondition of *op1* (resp. *op2*).

4.3 An Example

As an example, let us build the connector manager for our example of producers and consumers related by our previously defined R connector. The resulting machine is shown in Fig. 10. Note how operation *send*, which is involved in a connection, has changed its definition. Also, note how the *connect_R* operation and the invariant look like for this particular case. Operation *prod*, which was not involved in any connection, simply calls the original (now *prod'*) operation. Notice that, due to the automated generation of this specification, some (harmless) redundancies emerge (see the invariant of the connector manager, for instance).

```
MACHINE
    $R\_Manager$
EXTENDS
    $Producer\_Manager'$, $Consumer\_Manager'$
VARIABLES
    $RSet$
INVARIANT
    $RSet \subseteq ProducerSet \times ConsumerSet \;\wedge\; RSet \subseteq ProducerSet \mapsto ConsumerSet$
INITIALISATION
    $RSet := \emptyset$
OPERATIONS
    $R\_disconnect(x, y) \;\hat{=}$
        PRE $(x, y) \in RSet$ THEN $RSet := RSet - \{(x, y)\}$ END
    $R\_connect(x, y) \;\hat{=}$
        PRE $x \in ProducerSet \wedge y \in ConsumerSet$
        THEN $RSet(x) := y$
        END
    $send(x) \;\hat{=}$
        PRE $x \in ProducerSet \wedge x.p\text{-}var \neq 0$ THEN
        $(RSet[\{x\}] \neq \emptyset \Longrightarrow x.send'() || RSet(x).obtain'(x.p\text{-}var)$ []
        $(RSet[\{x\}] = \emptyset \Longrightarrow x.send'())$
        END
    $prod(i, x) \;\hat{=}$
        PRE $x \in ProducerSet \wedge (i \neq 0) \wedge (x.p\text{-}var = 0)$
        THEN $prod'(i, x)$
        END
    ⋮
END
```

Fig. 10. The manager for connector *R*, relating machines *Producer* and *Consumer*

4.4 Current Limitations of Connector Definitions

According to our definition of connectors, there exist various limitations on how machine instances can be related.

First, connectors are just *binary*, i.e., they can involve only two participants. It is not difficult to think of more general n-ary connectors, although the functionality/injectivity restrictions on the corresponding relation will have to be generalised (restrictions of the kind of functional dependencies as in databases would be necessary for n-ary connectors).

Second, due to the functionality/injectivity restrictions, having a connector of type $op1 \rightarrow op2$ (resp. $op1 \leftarrow op2$) forbids having a one-to-many (resp. many-to-one) connector "topology". Clearly, the specifier cannot employ connectors, as we have defined them, for these kinds of communication. Nevertheless, in those cases where our extension is applicable, the specifier gets a structured definition of the dynamism, in which most of the proof obligations can be automatically discharged (except those related to preconditions of operations at the target of connections, as we explained). Furthermore, it is possible to generalise the mechanism to connectors of arbitrary multiplicities, but in the general case the communication needs to be done asynchronously, since it requires an iteration mechanism. Of course, iteration is not available at the specification stage in B, so it needs to be characterised as a transaction. Due to space restrictions, we are unable to show in full detail the generalised form of connectors. But, essentially, the mechanism is the following: Suppose that we have a connection $op1 \rightarrow op2$ in a connector R relating M_1 and M_2, and which requires a one-to-many multiplicity. Then, the characterisation of $n.op1$ within *R_Manager* consists of:

1. a set $remaining \subseteq M_2Set$, that keeps track of those instances waiting to be called,
2. a boolean variable *sending* to indicate if the manager is engaged in a "sending" transaction,
3. an extra operation *send*, that:
   - if *remaining* is nonempty and *sending* is true, it nondeterministically chooses an element m from *remaining*, removes it from *remaining* and calls $m.op2$,
   - if *remaining* is empty and *sending* is true, it finishes the sending transaction by setting *sending* to false.
4. the assertion $sending = false$ is added as an extra precondition of all other operations (i.e., the other operations are *blocked* if the connector manager is involved in a sending transaction).

5 Conclusions and Future Work

We have defined an extension to the B method to support the definition of *connectors* [4], in the software architectures sense [13]. The extension allows one to relate dynamically generated instances of abstracts machines for communication. The approach complements a previously proposed extension to B to support dynamic creation and deletion of machine instances, using additional structuring mechanisms. The extension has been defined with the intention of making it fully compatible with the standard B method. Indeed, specifications written using our extensions are systematically mapped into standard B specifications.

The connector definitions that we propose allow us to enable the communication, under certain restrictions, between abstract machines in a specification which presents dynamic creation and deletion of machine instances. The way in which we provide semantics to the extension is based on the work in [6]. Although the allowed forms of communication are restricted, the use of connectors favours specification structuring (with its well known advantages for understandability and simplification of proofs), and allows one to specify systems with structural dynamism at a higher level of abstraction. The provided level of abstraction is based on ideas developed by the software architectures community, particularly, the view of component based systems as modules related by means of connectors, and the externalisation of component interaction. This higher level of abstraction might have a positive impact in the effort needed to complement the B method with modern system design methodologies, such as object orientation. In fact, currently there exist various approaches to the modelling of object oriented features in B, particularly the work of H. Treharne [15], K. Lano et al. [10] and C. Snook and M. Butler [14]. We believe that these approaches might benefit from our characterisation of dynamic creation/deletion of machine and connector instances, and their advantages in the decomposability of specifications. For example, in [15], classes and associations are translated into B as an *ad hoc* unstructured specification; in [10], a notion similar to that of class manager is the smallest unit of modularity (as opposed to our finer grained aggregation mechanism); in [14], entire class diagrams are translated into B as single abstract machines. Being able to exploit the modularisation of specifications not only benefits the specifier by alleviating the proof efforts (by decomposing proofs in smaller "lemmas") and making specifications easier to understand [5]; modularisation also greatly improves analysability. For example, a structured B specification of a system allows us to validate modules *independently*, via animation. It also allows fully automated verification mechanisms, such as those based on model checking [11], to scale up and be applicable to a wider range of system specifications, by contributing to cope with the well known combinatorial state explosion problem.

As work in progress, we are currently working on a better developed generalisation of connectors, that covers more cases of communication. We are also studying the treatment of our connector definitions at refinement and implementation stages. Notice that, since our extensions to the B language are *definitional*, the feasibility of refinement and implementation is guaranteed. Nevertheless, we are exploring ways of systematically producing correct implementations for connector definitions, supplementing what has been done in [3] for machine aggregation. Also, we are currently studying how our AGGREGATES structuring mechanism, together with the support for connectors, combines with other structuring mechanisms of B when the architectural organisation of a specification involves various layers of abstract machines.

References

1. J.-R. Abrial, *The B-Book, Assigning Programs to Meanings*, Cambridge University Press, 1996.
2. N. Aguirre, J. Bicarregui, T. Dimitrakos and T. Maibaum, *Towards Dynamic Population Management of Components in the B Method*, in Proceedings of the 3rd International Conference of B and Z Users ZB2003, Turku, Finland, LNCS, Springer-Verlag, 2003.

3. N. Aguirre, J. Bicarregui, L. Guzmán and T. Maibaum, *Implementing Dynamic Aggregations of Abstract Machines in the B Method*, in Proceedings of the International Conference on Formal Engineering Methods ICFEM 2004, Seattle, USA, LNCS, Springer-Verlag, 2004.
4. R. Allen and D. Garlan, *Formalizing architectural connection*, in Proceedings of the Sixteenth International Conference on Software Engineering ICSE '94, IEEE Computer Society Press, 1994.
5. *The B-Toolkit User Manual*, B-Core (UK) Limited, 1996.
6. J. Bicarregui, K. Lano and T. Maibaum, *Towards a Compositional Interpretation of Object Diagrams*, in Proceedings of IFIP TC 2 working conference on Algorithmic Languages and Calculi, Bird and Meertens (eds), Chapman and Hall, 1997.
7. Digilog, *Atelier B - Générateur d'Obligation de Preuve, Spécifications*, Technical Report, RATP SNCF INRETS, 1994.
8. T. Dimitrakos, J. Bicarregui, B. Matthews and T. Maibaum, *Compositional Structuring in the B-Method: A Logical Viewpoint of the Static Context*, in Proceedings of the International Conference of B and Z Users ZB2000, York, United Kingdom, LNCS, Springer-Verlag, 2000.
9. J.L. Fiadeiro and T. Maibaum, *Design Structures For Object-Based Systems*, in S. Goldsack and S. Kent (eds.), Formal Aspects of Object-Oriented Systems. Prentice Hall, 1994.
10. K. Lano, D. Clark and K. Androutsopoulos, *UML to B: Formal Verification of Object-Oriented Models*, in Proceedings of the International Conference on Integrated Formal Methods IFM 2004, Canterbury, United Kingdom, LNCS, Springer-Verlag, 2004.
11. M. Leuschel and M. Butler, *ProB: A Model Checker for B*, in Proceedings of FM 2003, Pisa, Italy, LNCS, Springer-Verlag, 2003.
12. B. Meyer, *Object-Oriented Software Construction*, Second Edition, Prentice-Hall International, 2000.
13. M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.
14. C. Snook and M. Butler, *UML-B: Formal modelling and design aided by UML*, Technical Report, Electronics and Computer Science, University of Southampton, Southampton, United Kingdom, 2004.
15. H. Treharne, *Supplementing a UML Development Process with B*, in Proceedings of FME 2002: Formal Methods– Getting IT Right, Denmark, LNCS 2391, Springer, 2002.