Towards Dynamic Population Management of Abstract Machines in the B Method*

Nazareno Aguirre[1**], Juan Bicarregui[2], Theo Dimitrakos[2], and Tom Maibaum[1]

[1] Department of Computer Science, King's College London,
Strand, London WC2R 2LS, United Kingdom, {aguirre, tom}@dcs.kcl.ac.uk
[2] Rutherford Appleton Laboratory, Chilton, Didcot,
OXON, OX11 0QX, United Kingdom {J.C.Bicarregui, T.Dimitrakos}@rl.ac.uk

Abstract. We study some restrictions associated with the mechanisms for structuring and modularising specifications in the B abstract machine notation. We propose an extension of the language that allows one to specify machines whose constituent modules (other abstract machines) may change dynamically, i.e., at run time. In this way, we increase the expressiveness of B by adding support for a common activity of the current systems design practice.
The extensions were made without having to make considerable changes in the semantics of standard B. We provide some examples to show the increased expressive power, and argue that our proposed extensions respect the methodological principles of the B method.

Keywords: Structuring mechanisms, modularisation, object orientation, dynamic reconfiguration.

1 Introduction

Formal methods support precise and rigorous specifications of those aspects of a computer system capable of being expressed in a formal language. One of the main advantages of formal methods is that, in addition to aiding in the elimination of ambiguities in specification, they allow for analysis and verification of system properties prior to implementation. Since defining what a system should do and understanding the implications of these decisions are amongst the most troublesome problems in software engineering, the use of formal methods has major benefits.

However, formal methods are hard and expensive to use, and they may require a strong background in formal reasoning in order to perform the analysis

* This work was partially supported by the Engineering and Physical Sciences Research Council of the U.K., through projects: *The Integration of Two Industrially Relevant Formal Methods (VDM+B)*, Grants GR/L68445 and GR/L68452, and *Objects, Associations and Subsystems: A Hierarchical Approach to Encapsulation*, Grants GR/M72630 and GR/N00814.

** Contact author. Phone number: +44 (0)207 848 1166. Fax: +44 (0)207 848 2851. Nazareno Aguirre is currently on leave from Departamento de Computación, FCE-FQyN, Universidad Nacional de Río Cuarto, Río Cuarto, Córdoba, Argentina.

and verification tasks. In the formal specification process, and even more so in the formal analysis process, appropriate tool support is then a necessity. In order to be able to provide tool support, it is generally required that the semantics underlying formal languages have to be simple. Simpler semantics usually leads to having restricted expressive power. So, a balance has to be found in order to achieve what is considered important for the take-up of a formal method (and its successful use in the development of systems), namely, simple but sufficiently expressive semantics, tool support, structure and relevance to the current systems engineering practice, etc.

Model based formal methods such as B [1], VDM [14] and Z [15] are among the few formal methods currently in use by industry and supported by commercial tools. They have been used in a variety of industrial case studies for the specification and verification of mission critical systems, in application domains varying from the rail industry to smart cards. Using such formal methods in the development of an information system is about: eliminating all ambiguity beginning right from the interpretation of the need, constructing a specification which is both coherent and conformant with the need (the model), and elaborating the software system which realises the specification, in successive stages. The coherence of the model and the conformity of the final program in relation to this model are guaranteed by mathematical proofs.

The languages referred to above are considerably less expressive than many object-oriented formalisms, but also considerably simpler, better structured, and in some cases with important tool support and proof assistance [9][4][12], due to their simpler semantics. One of these languages, the B language, has an associated method, described in [1], and commercial tool support [9][4]. A useful feature present in object-oriented languages is the possibility of dynamically creating or deleting modules or components (objects in the object-oriented terminology). In fact, dynamic object management has now become a common task in systems design practice. The B language and its associated method lack this useful feature of object-oriented languages: having it in the specification language of the B method would be equivalent to being able to dynamically create or delete abstract machines. The work in [16] and the different object-oriented variants of model-oriented specification languages provide evidence of the need for this feature.

In this paper, we make a first attempt to provide an extension of the B language and its semantics, in order to support dynamic management of abstract machine populations. In recognition of the fact that maintaining compatibility with the existing tool support for the B method is very important, we concentrate on "extending" (conservatively) the current language and semantics, rather than "changing" it. In effect, we ensure that:

1. one can possibly reduce the semantics associated with the proposed extension of the B specification language to the standard semantics of the B method,
2. the proposed extension does not affect the semantics of the core specification language of B.

The resulting language is in some aspects clearly more expressive than standard B, without being in the realm of object-oriented languages. We therefore increase the expressiveness of B by building into the language support for common activities of the current systems design practice, while avoiding the introduction of the complexity that is often associated with the semantics of fully-fledged object-oriented languages, including the object-oriented variants of the abovementioned model oriented formal methods.

2 Adding Dynamism to B

In B, the declaration of an abstract machine corresponds to the declaration of a kind of "template" of a component. An abstract machine is not a component itself, since it might prescribe the way many different components work. The creation of a number of different specification components corresponding to a single abstract machine declaration can be achieved by means of renaming and inclusion, using some of the structuring mechanisms available, of the renamed machines in some "super" machine M, in a way similar to what is called *cloning* in some object-oriented languages. However, the machines included in a super machine M are *fixed*: clearly, during the run time of M, neither the modular structure of M, in terms of the submachines it is built out of, nor the number of included machines can change. So, abstract machines cannot be considered as *objects* in an object-oriented sense, but instead they are closer to standard *modules* of traditional imperative programming languages. The advantages of the concept of *object* over that of *primitive module* are well-known; many of them could be considered differences between traditional imperative and object-oriented languages.

We extend the notation of abstract machines to allow for dynamic management of abstract machine populations. The notation of single, basic abstract machines is preserved. The changes are in the way we build bigger machines in terms of more primitive ones, i.e., in the structuring notation. In this paper, we restrict ourselves to studying a particular type of INCLUDES, the one characterised by the EXTENDS clause. For the sake of simplicity, we also ignore for the moment the issues related to the use of parameterised machines, and explain the concepts for machines without parameters, although it will be clear how the same concepts apply to parameterised machines straightforwardly.

3 Population Management: The Standard B Approach

To motivate our work, let us introduce an example that shows how a specification might be structured in B. This example is shown in Figure 1, and consists of an extension of a variant of the primitive machine *Scalar*, found in pages 320 and 321 of [1]. Machine *Scalar* consists only of an integer variable, and operations to update and return the value of the variable.

A structured machine built on top of *Scalar* is proposed in [1] as well, as machine *TwoScalars*. We show the definition of *TwoScalars* in Figure 2. As seen

```
MACHINE
    Scalar
VARIABLES
    var
INVARIANT
    var ∈ INT
INITIALIZATION
    var :∈ INT
OPERATIONS
    chg(v)  ≙  PRE  v ∈ INT  THEN  var := v  END
    v ⟵ val  ≙  BEGIN  v := var  END
END
```

Fig. 1. Abstract machine *Scalar*.

there, multiple copies of *Scalar* are "imported" in *TwoScalars*, by means of copy and renaming of (some of) the language elements of the original *Scalar* machine definition [1]. An extra operation *swap* is declared in this machine, calling in parallel the *chg* operations of machines xx and yy.

```
MACHINE
    TwoScalars
EXTENDS
    xx.Scalar, yy.Scalar
OPERATIONS
    swap  ≙  BEGIN  xx.chg(yy.var)  ||  yy.chg(xx.var)  END
END
```

Fig. 2. Abstract machine *TwoScalars*.

Now, suppose we decided we need a generalisation of this previous machine, one in which the number of scalars varies over time by creating or deleting dynamically new scalars, and where the *swap* operation might be applied to any two machines. The standard way of dealing with this problem in B, as shown in several examples of Chapter 8 in [1] and also in [13], is by defining a new machine, which includes both the operations of *Scalar*, relativised to names for the "instances", and the population management operations. Machine *SeveralScalars*, described in Figure 3, is defined using this approach. For this new machine, machine definition *Scalar* had to be discarded, and all the operations corresponding to it had to be adapted and included in *SeveralScalars*. A set, *scalars*, is used to denote the names of the active scalar instances. Operations *chg* and *val*, originally defined in *Scalar*, had to be rewritten in this machine specification, now relativised to the corresponding instances (see the extra parameter in each of these operations). Variable *var* was also incorporated to *SeveralScalars*, now representing the values of the original *var* for each of the active instances of scalar. The initialisation substitution of *Scalar* became an assignment (in fact, part of a parallel assignment) in *add_sc*, the operation that adds a new scalar in this machine.

```
MACHINE
    SeveralScalars
SETS
    SCALARSET
VARIABLES
    var, scalars
INVARIANT
    (var ∈ scalars → INT) ∧ (scalars ⊆ SCALARSET)
INITIALIZATION
    var, scalars := ∅, ∅
OPERATIONS
    chg(v, p)  ≙
        PRE   v ∈ INT ∧ p ∈ scalars
        THEN   var := (dom(var) − {p}) ◁ var) ∪ {(p, v)}
        END

    v ⟵ val(p)  ≙  PRE  p ∈ scalars  THEN  v := var(p)  END

    swap(p, q)  ≙
        PRE   p ∈ scalars ∧ q ∈ scalars
        THEN   var := var <+{(p, var(q)), (q, var(p))}
        END

    add_sc(p)  ≙
        PRE   p ∈ (SCALARSET − scalars)
        THEN   (scalars := scalars ∪ {p})  ||
        (ANY v WHERE v ∈ INT THEN var := var ∪ {(p, v)} END)
        END

    rem_sc(p)  ≙
        PRE   p ∈ scalars
        THEN   scalars := scalars − {p}  ||
            var := (dom(var) − {p}) ◁ var
        END
END
```

Fig. 3. Abstract machine *SeveralScalars*.

This is a standard approach to the management of multiple instances of certain objects. It is, certainly, a problem, since the whole specification of a scalar had to be rewritten. Imagine a case in which the machine whose population we need to manage, say M, is not as simple as our *Scalar* machine, and instead consists of a complex structure in terms of "submachines"; if we want to specify a machine that manages the population of M, then the whole specification of M must be rewritten. Therefore, specifications cannot be modularised into natural conceptual entities, proofs cannot be "localised" to relevant specification parts, etc.

4 A Notation for Dynamic Creation of Machines

We suggest that it is possible to provide B with a richer notation, that allows us to dynamically manage the population of abstract machines, partly overcoming the problems mentioned in the previous section. The general form of our notation is not difficult to understand. The AGGREGATES M_1 clause in a machine M indicates that multiple machines of type M_1 are available in M, in the same style as for EXTENDS, i.e., *promoting* all operations of the included machine. Included machines are declared to belong to an *instance set*, whose name is M_1*Set* (in our case *ScalarSet*). Instance sets are used to characterise live instances of machine types.

A machine equivalent to the *SeveralScalars* machine in Figure 3 is written in our proposed extended notation as follows:

MACHINE
    SeveralScalars'
AGGREGATES
    Scalar
OPERATIONS
    $swap(p, q) \;\hat{=}$
        PRE $p \in ScalarSet \wedge q \in ScalarSet$
        THEN $p.chg(q.var) \; ||| \; q.chg(p.var)$
        END
END

In contrast to machine *SeveralScalars*, machine *SeveralScalars'* is indeed defined in terms of the primitive machine *Scalar*. It does not include the declaration of a set of instances (*scalars* in machine *SeveralScalars*), since it is declared *implicitly*, by the AGGREGATES clause. Two operations, called *add_Scalar* and *del_Scalar*, are automatically generated and implicitly included by the AGGREGATES clause. These operations are meant to manipulate the population of instances of scalar. For our example, they are defined in the following way:

$add\_Scalar(p) \;\hat{=}$
  PRE $p \in (\mathsf{NAME} - ScalarSet)$
  THEN $ScalarSet := ScalarSet \cup \{p\} \quad || \quad p.init$
  END

$$
\begin{array}{l}
del\_Scalar(p) \quad \hat{=} \\
\quad \text{PRE} \quad p \in ScalarSet \\
\quad \text{THEN} \quad ScalarSet := ScalarSet - \{p\} \quad || \\
\qquad\qquad var := (\mathsf{dom}(var) - \{p\}) \lhd var \\
\quad \text{END}
\end{array}
$$

The set NAME is assumed to be predefined in some way in B (see the section regarding the semantics of the extension). It denotes the set of all names of machines (more than one machine type might be aggregated by a particular machine). It is assumed that the graph corresponding to the AGGREGATES dependency between machines is *acyclic*; in other words, no recursive (either direct or indirect) aggregation is allowed. The notation $p.chg(x)$ is in fact just a convenient more readable way (borrowed from object orientation) of writing $chg(x, p)$, i.e., p is simply an extra argument of chg.

The substitution *init* used in the definition of operation *add_Scalar* is not explicitly declared in the aggregated machine as an operation, but corresponds to the substitution defined in the INITIALIZATION clause, now relativised to an instance. For example, for the case of scalars, the initialisation was:

$$var :\in \mathsf{INT}$$

Then, $p.init$ is defined as:

$$\text{ANY } v \text{ WHERE } v \in \mathsf{INT} \text{ THEN } var := var \cup \{(p, v)\} \text{ END}$$

This expression is rather complicated, because we need to maintain the nondeterminism in the original substitution. We can use the syntax sugaring defined in pages 266 and 267 of [1] to express the above substitution in the following more readable form:

$$var(p) :\in \mathsf{INT}$$

To better understand the meaning of $p.init$, consider a simpler initialisation assignment, such as:

$$var := 0$$

Then, $p.init$ would be simply defined as:

$$var := var \cup \{(p, 0)\}$$

In case any of the automatically generated operations of the aggregating abstract machine should not be exported, a wrapper machine promoting the interface operations could be declared, as is usual in the B method.

Note that a new combination of substitutions, that we call *interleaving parallel composition* (denoted by the triple bar), is used in the above machine. We describe below the semantics of this operator in detail, and our need for it.

5 Providing Semantics to the Extension

A straightforward way to provide semantics to the proposed syntax extension to B would be to simply indicate that specifications like *SeveralScalars'* are syntax sugaring for an equivalent *flat* specification, like *SeveralScalars*. We could in this way take advantage of the already well-defined semantics and consistency checking of standard B for the syntax extension.

However, we wish to treat AGGREGATES as a proper structuring mechanism. Certainly, the above straightforward way of giving semantics to the extension does not treat AGGREGATES as a proper structuring mechanism. For instance, it would be necessary to flatten the specification in order to perform the consistency checking of a specification structured using AGGREGATES, and therefore there would not be, as for the other structuring mechanisms, a way of checking consistency of the structured machine in terms of the consistency of the simpler composing submachines.

On the contrary, the way we provide semantics for the AGGREGATES clause relies on the generation of a *population manager* for each aggregated machine. Given a basic (or flat) machine M, we construct a machine $MManager$, in such a way that its internal consistency is guaranteed, provided that M is internally consistent. We define the clause AGGREGATES M to mean simply EXTENDS $MManager$.

The generation of the population manager is described below.

5.1 Generating Population Managers

Let M be a generic basic abstract machine, of the form:

```
MACHINE
    M
SETS
    s
CONSTANTS
    c
PROPERTIES
    PROP(s, c)
VARIABLES
    v
INVARIANT
    I(s, c, v)
INITIALIZATION
    INIT(v) = P_INIT(v) | @x' · (Q_INIT(x', v) ==> v := x')
OPERATIONS
    r <-- op(p̄)  ≙  P(p̄, v) | @x̄' · (Q(x̄', p̄, v) ==> v, r := x̄')
    ⋮
END
```

Note that we have written the substitutions corresponding to the initialisation and the operations in the most general form (according to Theorem 6.1.1 in page 284 of [1], all substitutions are reducible to this *normal form*). Also, for the sake of simplicity, we have considered a generic machine without parameters, although it will be clear that our techniques can be straightforwardly extended to cope with parameterised machines.

Let us assume that M is internally consistent, i.e., it satisfies its proof obligations, and that $\overline{T}$ are the types assigned to its variables v [1]. The population manager $MManager$ for the abstract machine specification M has the following form:

MACHINE
  $MManager$
SETS
  s
CONSTANTS
  c
PROPERTIES
  $PROP(s, c)$
VARIABLES
  $MSet,\ v$
INVARIANT
  $(\forall n \cdot n \in MSet \Rightarrow I(s, c, v(n))) \wedge (MSet \subseteq \mathsf{NAME}) \wedge (v \in MSet \rightarrow \overline{T})$
INITIALIZATION
  $MSet, v := \emptyset, \overline{\emptyset}$
OPERATIONS
  $add\_M(n) \quad \hat{=}$
  PRE $\ n \in (\mathsf{NAME} - MSet)$
  THEN $\ MSet := MSet \cup \{n\} \quad || \quad INIT(v(n))$
  END
  $del\_M(n) \quad \hat{=}$
  PRE $\ n \in MSet$
  THEN $\ MSet := MSet - \{n\} \quad || \quad v := (\mathsf{dom}(v) - \{n\}) \lhd v$
  END
  $r \longleftarrow op(\overline{p}, n) \quad \hat{=}$
  $P(\overline{p}, v(n)) \wedge (n \in MSet) \mid @x' \cdot (Q(x', \overline{p}, v(n)) \Longrightarrow v(n), r := x')$
  $\vdots$
END

We can describe then the construction of the population manager of M as follows:

- The sets, constants and properties defined in M are included without any change in $MManager$,
- An extra variable $MSet$, representing the set of live instances of M, is declared,
- all variables defined in M are included as variables of $MManager$, relativised to names of live instances, i.e., each variable V of type T becomes a mapping from $MSet$ to T,

[1] Recall that in the B method, it is a requeriment for the invariant to imply that all variables are assigned, directly or indirectly, a corresponding type.

– the invariant M is relativised to names of instances, and incorporated as a conjunct of the invariant of $MManager$,
– all operations defined in M are included as operations of $MManager$, adding an extra parameter of type $MSet$, which indicates in which instance the operation should be executed,
– population management operations add_M and del_M, which are automatically generated from the definition of M, are defined.

To clarify the generation of population managers, consider the machine in Figure 4. It is the result of the generation of a population manager for machine *Scalar*, given in Figure 1.

MACHINE
  ScalarManager
VARIABLES
  $var, ScalarSet$
INVARIANT
  $(\forall n \cdot n \in ScalarSet \Rightarrow var(n) \in \mathsf{INT}) \wedge (ScalarSet \subseteq \mathsf{NAME}) \wedge$
  $(var \in ScalarSet \rightarrow \mathsf{INT})$
INITIALIZATION
  $ScalarSet, var := \emptyset, \emptyset$
OPERATIONS
  $add\_Scalar(n) \ \hat{=}$
  PRE $n \in (\mathsf{NAME} - ScalarSet)$
  THEN $(ScalarSet := ScalarSet \cup \{n\}) \ \ || \ \ var(n) :\in \mathsf{INT}$
  END

  $del\_Scalar(n) \ \hat{=}$
  PRE $n \in ScalarSet$
  THEN $ScalarSet := ScalarSet - \{n\} \ \ ||$
  $var := (\mathsf{dom}(var) - \{n\}) \lhd var$
  END
  $chg(v, n) \ \hat{=}$
  PRE $v \in \mathsf{INT} \wedge n \in ScalarSet$
  THEN $var(n) := v$
  END

  $v \longleftarrow val(n) \ \hat{=}$ PRE $n \in ScalarSet$ THEN $v := var(n)$ END

END

Fig. 4. Abstract machine *ScalarManager*.

There are just a few very basic differences between the meaning of machine *SeveralScalars'* (as an extension of *ScalarManager*) and the meaning of machine *SeveralScalars*; we use a general sort, called NAME, as the domain of names for machine instances (recall that in the flat specification *SeveralScalars*, a special local set named *SCALARSET* is used). It is easy to extend the core of B with a definition of a set NAME, and a sufficiently large number of constants of this sort; in fact, it is even not necessary to incorporate this to B's core, but instead a stateless abstract machine containing the definition might be declared, and implicitly used in all other machine declarations.

5.2 Consistency of Generated Population Managers

As we mentioned before, if the abstract machine M is internally consistent, i.e., it satisfies its proof obligations, then we guarantee that the generated *MManager* is also internally consistent, by construction. We justify this claim here.

Let us consider then the generic abstract machine specification M described above. Adapting the style used in [10] to the more general form of substitutions, we express the proof obligations corresponding to M as follows:

IC1 $\exists s_0, c_0 : PROP(s_0, c_0)$
IC2 $PROP \Rightarrow \exists v_0 : I(v_0)$
IC3 $PROP \Rightarrow [INIT]I(v)$
IC4 $(PROP \wedge I(v) \wedge P(\overline{p}, v) \wedge Q(x', \overline{p}, v)) \Rightarrow [v, r := x']I(v)$

The proof obligations for machine *MManager* would then be:

M-IC1 $\exists s_0, c_0 : PROP(s_0, c_0)$
M-IC2 $PROP \Rightarrow \exists v_0', MSet_0 : I_{Man}(v_0', MSet_0)$
M-IC3 $PROP \Rightarrow [MSet, v := \emptyset, \overline{\emptyset}]I_{Man}(v, MSet)$
M-IC4a $(PROP \wedge I_{Man}(v, MSet) \wedge (n \in (\mathsf{NAME} - MSet)) \Rightarrow$
$[INIT(v(n)) \,||\, MSet := MSet \cup \{n\}]I_{Man}(v, MSet)$
M-IC4b $(PROP \wedge I_{Man}(v, MSet) \wedge (n \in MSet) \Rightarrow$
$[MSet := MSet - \{n\} \,||\, v := (\mathsf{dom}(v) - \{n\}) \lhd v]I_{Man}(v, MSet)$
M-IC4c $(PROP \wedge I_{Man}(v, MSet) \wedge P(\overline{p}, v(n)) \wedge (n \in MSet) \wedge Q(x', \overline{p}, v(n))) \Rightarrow$
$[v(n), r := x']I_{Man}(v, MSet)$

where $I_{Man}(X, Y)$ represents the invariant of *MManager* for X and Y, i.e., the formula:

$$(\forall n \cdot n \in Y \Rightarrow I(X(n))) \wedge (Y \subseteq \mathsf{NAME}) \wedge (X \in Y \to T)$$

Let us assume that the proof obligations of machine M have already been *discharged*. We prove that this implies the satisfaction of each of the proof obligations of *MManager*:

M-IC1: Trivial, due to the validity of IC1.
M-IC2: Let us consider $MSet_0 = \emptyset$ and $v_0' = \overline{\emptyset}$. We have to prove that $I(v_0', MSet_0)$ is satisfied:
- $\forall n \cdot n \in \emptyset \Rightarrow I(v_0(n))$: Trivially true (the antecedent of the implication is false).
- $\emptyset \subseteq \mathsf{NAME}$: Trivially true, since NAME is defined to be a set.

– $\overline{\emptyset} \in \emptyset \rightarrow \overline{T}$: Trivially true, since $(\emptyset \rightarrow \overline{T}) = \overline{\{\emptyset\}}$.

M-IC3: valid (see proof for M-IC2).

M-IC4a: This proof obligation indicates that operation *add_M* preserves the invariant. So, under the hypothesis:

$PROP \wedge$
$(\forall n \cdot n \in MSet \Rightarrow I(s, c, v(n))) \wedge (MSet \subseteq \mathsf{NAME}) \wedge (v \in MSet \rightarrow \overline{T}) \wedge$
$(n \in (\mathsf{NAME} - MSet))$

we have to prove that the invariant is re-established after the assignment:

$$INIT(v(n)) \,||\, MSet := MSet \cup \{n\}$$

Let us assume $P_{INIT}(v(n))$ (the precondition of $INIT$), and let x' be an arbitrary expression such that $Q_{INIT}(x', v(n))$. We prove that each of the conjuncts of the invariant is preserved:

- $\forall n_0 \cdot n_0 \in MSet \cup \{n\} \Rightarrow I(s, c, (v <\!\!+ \{n \mapsto x'\})(n_0))$: If n_0 is distinct from n, then this holds due to the hypothesis. It remains to be proved then that this also holds when $n_0 = n$, i.e., that $I(s, c, x')$ holds. we know this is true, because the initialisation of M preserves I (IC3).
- $MSet \cup \{n\} \subseteq \mathsf{NAME}$: Trivially true, due to the hypothesis $MSet \subseteq \mathsf{NAME}$ and $n \in (\mathsf{NAME} - MSet)$.
- $(v <\!\!+ \{n \mapsto x'\}) \in (MSet \cup \{n\}) \rightarrow \overline{T}$: Holds trivially, due to our hypothesis
$$v \in MSet \rightarrow \overline{T}$$

and x' being of type $\overline{T}$ (enforced because $INIT$ is well-formed).

M-IC4b: This proof obligation indicates that operation *del_M* preserves the invariant. So, under the hypothesis:

$PROP \wedge$
$(\forall n \cdot n \in MSet \Rightarrow I(s, c, v(n))) \wedge (MSet \subseteq \mathsf{NAME}) \wedge (v \in MSet \rightarrow \overline{T}) \wedge$
$(n \in MSet)$

we have to prove that the invariant is re-established after the assignment

$$MSet := MSet - \{n\} \quad || \quad v := (\mathsf{dom}(v) - \{n\}) \lhd v$$

i.e., that the following holds:

$$\begin{array}{l}(\forall n_0 \cdot n_0 \in MSet - \{n\} \Rightarrow I(s, c, ((\mathsf{dom}(v) - \{n\}) \lhd v)(n_0))) \wedge \\ (MSet - \{n\} \subseteq \mathsf{NAME}) \wedge \\ ((\mathsf{dom}(v) - \{n\}) \lhd v) \in (MSet - \{n\}) \rightarrow \overline{T}\end{array}$$

The first conjunct reduces to

$$\forall n_0 \cdot n_0 \in MSet - \{n\} \Rightarrow I(s, c, v(n_0))$$

because n_0 is distinct from n. Due to our hypothesis, the above holds. The second and third conjuncts follow immediately from the hypothesis.

M-IC4c: This proof obligation indicates that the operations that were originally defined in M preserve the invariant when adapted and incorporated into the manager of M. So, under the hypothesis:

$$(PROP \wedge I_{Man}(v, MSet) \wedge P(\overline{p}, v(n)) \wedge (n \in MSet) \wedge Q(x', \overline{p}, v(n)))$$

the invariant is re-established, i.e., that $I_{Man}(x', MSet)$ holds. The second conjunct of the invariant is trivially preserved, since the substitution does not write on $MSet$. The first conjunct of the invariant is also preserved, since according to IC4, an assignment based on relation Q preserves I under hypothesis P. The third conjunct of the hypothesis is trivially preserved, according to the definition of assignments of the form $f(x) := E$ (page 267 of [1]).

5.3 Proof Obligations for AGGREGATES

Using the extension described above, we would be provided with a more suitable notation for dynamic population management of components in B. To check for consistency of a machine M_1 aggregating an internally consistent machine M, we just need to check the proof obligations corresponding to EXTENDS $MManager$, according to the semantics described above for the proposed extension. In this way, we are treating AGGREGATES as a proper structuring mechanism, and not just as a short-hand for an unstructured flat specification; in other words, we do not need to flatten specifications involving AGGREGATES in order to check for consistency.

6 Using Aggregated Machines: Interleaving Parallel Composition

In the abstract machine *TwoScalars* that we described in Figure 2, an operation *swap* was defined. This operation simultaneously *called* two other operations, namely $xx.chg$ and $yy.chg$. This is allowed because these two operations belong to different extended machines. If two (or more) operations belong to the same abstract machine, then they cannot be called in parallel. Several researchers noticed this restriction, and proposed different extensions to B and languages with similar characteristics, incorporating write frames, modifying the semantics of parallel composition, etc [6][5][11]. A common restriction on the parallel composition of statements, which is reasonable, is that composed statements should not write on the same variables.

In the case of our machine *SeveralScalars*', which aggregates *Scalar*, we would like to be able to call in parallel operations $p.chg(q.var)$ and $q.chg(p.var)$, which, at least when p and q are different, naturally *seem* to belong to different machines. However, because of the way we provide meaning to the AGGREGATES clause, even when p are q are different, they will be writing on the same variable, namely the mapping var. Therefore, according to the definitions of parallel composition we know of, we cannot define an operation similar to the *swap* operation in *SeveralScalars* when, instead of redefining scalar in a flat specification, we aggregate scalars.

Due to this problem, we are forced to introduce an extra operator for combining substitutions, in order to be able to use combinations of operations of aggregated machines. The operation we provide is different from the extensions or alternatives to parallel substitution we are aware of. For all substitutions S and T, and formula P, we define the *interleaving parallel composition* of S and T as follows:

$$[S \,|||\, T]P \quad \hat{=} \quad ([S][T]P) \wedge ([T][S]P)$$

In the presence of a sequencing operator (not present in the set of operations at the specification stage in the B method), $S \,|||\, T$ can be described as $(S;T)[](T;S)$. There are good reasons for the absence of sequencing at the specification stage in B; methodologically, it forces the specifier to describe behaviour in an abstract way, without allowing one to enforce a particular order in the substitutions that define an operation. We believe our interleaving parallel composition respects this philosophy, since the order in which the substitutions are applied in unknown. In fact, the interesting case is when $[S][T] = [T][S]$, which, intuitively, leads to a definition of non-interference between S and T [7][2]. Note that the definition of *swap* in *SeveralScalars*' is a case of a use of $|||$ with non-interfering substitutions: $(p.chg(q.var); q.chg(p.var))$ and $(q.chg(p.var); p.chg(q.var))$ both give the same result, i.e., they are non-interferent.

There are no side conditions for the well-formedness of $(S \,|||\, T)$. In particular, as we wanted, two substitutions S and T can be combined using $|||$ even when they write on the same variables.

It is important to say that it is not our aim to provide an alternative to parallel composition. As we indicated before, we need to introduce this extra substitution operator to be able to express in a natural way operations defined in terms of other operations in aggregated machines. In fact, substitutions of the form

$$x := y \,||\, y := x$$

for instance, are not equivalently defined using $|||$ instead. Note that

$$x := y \,|||\, y := x$$

does not swap the values of x and y, but instead lets variables x and y with the same value (either the original value of x or the original value of y).

An interesting special case of the use of interleaving parallel composition is the one in which the composing substitutions write on the same mapping variable, as, for instance, in the following substitution:

$$f(x) := E_1 \,|||\, f(y) := E_2.$$

[2] Our definition of interleaving parallel composition is related but not equivalent to the interleaving semantics of parallel composition: in our interleaving parallel composition, non-interference of the composed statements is not a requirement for the well-formedness of the composite statement.

Here, if x and y are distinct elements of the domain of f, the substitution changes the image of x by E_1 and the image of y by E_2. This clearly is not possible if we use $||$ instead of $|||$, since both composing substitutions write on the same variable (the mapping f).

The reader might argue that an equivalent substitution can be expressed, without the use of parallel composition (for our case, $f := f \lessdot\!\!+ \{(x, E_1), (y, E_2)\}$ would be an equivalent substitution without the use of parallel composition). However, this is just an alternative when defining substitutions in a flat specification. On the other hand, when the composing substitutions correspond to operations defined in "submachines", this alternative is no longer possible. So, for instance, we cannot define the *swap* operation using this approach in a machine extending *ScalarManager*.

We need to justify that the introduction of this new combination of substitutions does not affect the standard semantics of B. The following Theorem complements the result of Theorem 6.1.1 in page 284 of [1]. It proves that substitutions built using interleaving parallel composition reduce to the normal form, thus indicating that our extension is within B's standard semantics.

Theorem 1. *Let S and T be two substitutions which reduce to the normal form defined in 284 of [1]. Then, the substitution*

$$S \mid\mid\mid T$$

also reduces to the normal form.

Proof. Let S and T be two substitutions, which reduce to the normal form. If we prove that both $S;T$ and $T;S$ reduce to normal form, then $S|||T$ will also reduce to normal form, due to Theorem 6.1.1, since nondeterministic choice of reducible substitutions reduces to normal form.

Since S and T are reducible to normal form, they can be expressed respectively as follows:

$$\begin{array}{l} P_S \mid @x' \cdot (Q_S \Longrightarrow x := x') \\ P_T \mid @x'' \cdot (Q_T \Longrightarrow x := x'') \end{array}$$

We prove that $S;T$ also reduces to normal form. The proof for $T;S$ is similar. We refer to the basic properties of ";", given in pages 375 and 376 of [1], as "BP ;". We also refer to the laws of substitutions given in pages 284 and 285 of [1].

$$\begin{array}{l}
(P_S \mid @x' \cdot (Q_S \Longrightarrow x := x')) \,;\, (P_T \mid @x'' \cdot (Q_T \Longrightarrow x := x'')) = \{\text{BP} \,;\, (2)\} \\
P_S \mid (@x' \cdot (Q_S \Longrightarrow x := x')) \,;\, (P_T \mid @x'' \cdot (Q_T \Longrightarrow x := x'')) = \{\text{BP} \,;\, (8)\} \\
P_S \mid ([@x' \cdot (Q_S \Longrightarrow x := x')]P_T \mid \\
\qquad\qquad\qquad @x' \cdot (Q_S \Longrightarrow x := x') \,;\, @x'' \cdot (Q_T \Longrightarrow x := x'')) = \{\text{Law } 3\} \\
P_S \wedge ([@x' \cdot (Q_S \Longrightarrow x := x')]P_T) \mid (@x' \cdot (Q_S \Longrightarrow x := x') \,;\, @x'' \cdot (Q_T \Longrightarrow x := x''))
\end{array}$$

We now go on reducing $(@x' \cdot (Q_S \Longrightarrow x := x') \,;\, @x'' \cdot (Q_T \Longrightarrow x := x''))$.

$$
\begin{array}{l}
(@x' \cdot (Q_S \Longrightarrow x := x') \,;\, @x'' \cdot (Q_T \Longrightarrow x := x'')) = \{\text{BP} \,;\, (11)\} \\
@x'' \cdot (@x' \cdot (Q_S \Longrightarrow x := x') \,;\, (Q_T \Longrightarrow x := x'')) = \{\text{BP} \,;\, (5)\} \\
@x'' \cdot @x' \cdot ((Q_S \Longrightarrow x := x') \,;\, (Q_T \Longrightarrow x := x'')) = \{\text{BP} \,;\, (3)\} \\
@x'' \cdot @x' \cdot Q_S \Longrightarrow ((x := x') \,;\, (Q_T \Longrightarrow x := x'')) = \{\text{BP} \,;\, (9)\} \\
@x'' \cdot @x' \cdot Q_S \Longrightarrow ([x := x']Q_T \Longrightarrow x := x'; x := x'') = \{\text{Law } 6\} \\
@x'' \cdot @x' \cdot (Q_S \wedge [x := x']Q_T) \Longrightarrow (x := x'; x := x'') = \{\text{Def. };\} \\
@x'' \cdot @x' \cdot (Q_S \wedge [x := x']Q_T) \Longrightarrow x := [x := x']x''
\end{array}
$$

So, as we wanted to prove, $S;T$ reduces to normal form, which implies (together with $T;S$ reducing to normal form) that $S|||T$ also reduces to normal form.

6.1 Relating Parallel Composition and Interleaving Parallel Composition

It is interesting to compare the use of parallel substitution and interleaving parallel substitution. In [11], S. Dunne compares

$$(\mathit{skip} \,||\, x := x + 1) \text{ and } (x := x \,||\, x := x + 1),$$

illustrating that *skip* and $x := x$ are not *absolutely* equivalent. Surprisingly, substitutions

$$(\mathit{skip} \,|||\, x := x + 1) \text{ and } (x := x \,|||\, x := x + 1)$$

are well-formed, and indeed equivalent, since both reduce to $x := x + 1$. In fact, they are also equivalent to $(\mathit{skip} \,||\, x := x + 1)$.

To finish our introduction to the interleaving parallel composition, we state a Proposition, which gives a sufficient condition for the equivalence between $||$ and $|||$.

Proposition 1. *Let S and T be two substitutions, of the form:*

$$
\begin{array}{l}
P_S \mid @x' \cdot (Q_S \Longrightarrow x := x') \\
P_T \mid @x'' \cdot (Q_T \Longrightarrow y := x'')
\end{array}
$$

respectively. If $x \cap y = \emptyset$, $x \backslash P_T, Q_T$ and $y \backslash P_S, Q_S$, then

$$(S||T) = (S|||T)$$

7 Conclusions

We have argued for the benefits of extending the notation of the B language to support dynamic management of abstract machine populations. We proposed a preliminary notation, in which we generalise the EXTENDS clause (by defining a new clause AGGREGATES) to support dynamic creation and deletion of machines. The semantics of standard B is preserved by the extension, and just very simple machinery had to be built on top of B's core.

The mechanisms via which we can extend the B language have been used in [8], in the context of object-oriented modelling languages, and in [2][3], in the context of axiomatic specifications of reconfigurable architectures. Other concepts introduced in this previous work, such as the use of associations and inheritance, remain to be studied in the context of model-oriented specifications.

Among our priorities for future research in this direction are:

– to generalise the concept of *aggregate* to support associations between dynamic sets of instance machines;
– to study similar concepts to *aggregate* supporting specification structuring within the IMPLEMENTATION construct of the B method;
– to study the generalisation of the REFINEMENT construct to accomodate refinement between aggregates of dynamically managed instances;
– to provide a mechanism that allows instances and associations to be composed into a *subsystem* instance.

All the above are necessary for acheiving a general theory of dynamic management of component populations within specifications in the B language. A similar approach should also be possible for a larger group of similar model-oriented specifications such as Z and the module version of VDM.

An interesting new combination of substitutions, the interleaving parallel composition, emerged as a consequence of the use of machine aggregations. We plan to explore in more detail the implications of introducing this new operation in the B method.

Acknowledgements

We would like to thank the anonymous referees for their useful comments and suggestions.

References

1. J.-R. Abrial, *The B-Book, Assigning Programs to Meanings*, Cambridge University Press, 1996.
2. N. Aguirre and T. Maibaum, *A Temporal Logic Approach to the Specification of Reconfigurable Component-Based Systems*, in Proceedings of the 17th International Conference Automated Software Engineering ASE 2002, IEEE Press, 2002.
3. N. Aguirre and T. Maibaum, *A Logical Basis for the Specification of Reconfigurable Component-Based Systems*, to appear in Proceedings of Fundamental Aspects of Software Engineering FASE 2003, Poland, LNCS, Springer, 2003.
4. *The B-Toolkit User Manual*, B-Core (UK) Limited, 1996.
5. R.-J. Back and M. Butler, *Fusion and Simultaneous Execution in the Refinement Calculus*, Acta Informatica 35, vol 11, 1998.

6. D. Bert, M.-L. Potet and Y. Rouzaud, *A Study on Components and Assembly in B*, in Proceedings of the First B Conference, IRIN, Nantes, 1996.
7. J.C. Bicarregui, *Do Not Read This*, in Proceedings of FME 2002: Formal Methods– Getting IT Right, Denmark, LNCS 2391, Springer, 2002.
8. 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.
9. Digilog, *Atelier B - Générateur d'Obligation de Preuve, Spécifications*, Technical Report, RATP SNCF INRETS, 1994.
10. 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.
11. S. Dunne, *A Theory of Generalised Substitutions*, in Proceedings of the International Conference of B and Z Users ZB2002, Grenoble, France, LNCS, Springer-Verlag, 2002.
12. R.Elmstrøm, P.G.Larsen, P.B.Lassen, *The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications*, ACM Sigplan Notices, 1994.
13. K. Lano, *The B Language and Method, A Guide to Practical Formal Development*, Fundamental Approaches to Computing and Information Technology, Springer, 1996.
14. C. Jones, *Systematic Software Development Using VDM*, 2nd edition, Prentice Hall International,1990.
15. M. Spivey, *The Z Notation: A Reference Manual*, 2nd edition, Prentice Hall International, 1992.
16. H. Treharne, *Supplementing a UML Development Process with B*, in Proceedings of FME 2002: Formal Methods– Getting IT Right, Denmark, LNCS 2391, Springer, 2002.