Towards Dynamic Population Management of Components in Model Oriented Specifications*

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 the mechanisms for structuring and modularising specifications in model oriented specifications, choosing to study the formal specification language B as an example. We propose an extension of the language that allows one to specify machines whose composing modules (other abstract machines) may change dynamically, i.e., at run time.
The extensions were made without causing considerable changes in the semantics of standard B. We provide some examples to show the increased expressive power.

1 Introduction

The main advantage of formal methods is that, in addition to eliminating ambiguities in specification, they allow for analysis and verification of system properties. Formal methods support precise and rigorous specifications of those aspects of a computer system capable of being expressed in a formal language. Since defining what a system should do, and understanding the implications of these decisions, are amongst the most troublesome problems in software engineering, this 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 and verification tasks. In the formal specification process (and even more so in the formal analysis process), appropriate tool support is a necessity. Simple but sufficiently expressive semantics, tool support, structure and relevance to the current systems engineering practice are all important for the take-up of a formal method.

* 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.

** On leave from Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Río Cuarto, Córdoba, Argentina.

Model based formal methods such as B[1], VDM[9] and Z[10] 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 a information system is about: clearing all ambiguity straight from the interpretation of the need, constructing a specification both coherent and conform to the need (the model), elaborating the software system which realises the specification, in successive stages. The coherence of the model and then the conformity of the final program in relation to this model are guaranteed by mathematical proofs.

These languages 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 [6][4][7], due to their simpler semantics.

One of these languages, the B language, has an associated method, described in [1], and commercial tool support [6][4]. One of the restrictions of the B language (and its associated method) is its lack of a useful feature present in object-oriented languages, namely dynamic creation/deletion of modules or components (objects in object-oriented formalisms). However dynamic object management has now become a common task in the systems design practice. Having this feature in the specification language of the B method would be equivalent to being able to dynamically create or delete abstract machines. 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 machines population. 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 to introduce the complexity that is often associated with the semantics of fully-fledged object oriented languages, including the object oriented variants of 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 several 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, 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 it nor the number of included machines change), so abstract machines cannot be considered as *objects*, but instead they have to be considered 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 machines population. The notation of single, non-interacting 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 consists of an extension of a variant of the primitive machine *Scalar*, found in pages 320 and 321 of [1]:

MACHINE
    Scalar
VARIABLES
    var
INVARIANT
    $var \in \mathsf{INT}$
INITIALIZATION
    $x :\in \mathsf{INT}$
OPERATIONS
    $chg(v) \;\widehat{=}\; \text{PRE}\;\; v \in \mathsf{INT}\;\; \text{THEN}\;\; var := v\;\; \text{END}$
    $v \longleftarrow val \;\widehat{=}\; \text{BEGIN}\;\; v := var\;\; \text{END}$
END

This machine 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*:

MACHINE
    TwoScalars
EXTENDS

$xx.Scalar$, $yy.Scalar$
OPERATIONS
$swap \;\widehat{=}\; \text{BEGIN} \quad xx.chg(yy.var) \;\|\; yy.chg(xx.var) \quad \text{END}$
END

As seen here, 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.

Now, suppose we decide 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 [8], 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* would look as follows, under this approach:

MACHINE
SeveralScalars
SETS
SCALARSET
VARIABLES
$var, scalars$
INVARIANT
$(var \in scalars \rightarrow \mathsf{INT}) \wedge (scalars \subseteq SCALARSET)$
INITIALIZATION
$var, scalars := \emptyset, \emptyset$
OPERATIONS
$chg(v,p) \;\widehat{=}$
PRE $v \in \mathsf{INT} \wedge p \in scalars$
THEN $var := (\mathsf{dom}(var) - \{p\}) \lhd var \cup \{(p,v)\}$
END

$v \longleftarrow val(p) \;\widehat{=}\; \text{PRE} \quad p \in scalars \quad \text{THEN} \quad v := var(p) \quad \text{END}$

$swap(p,q) \;\widehat{=}$
PRE $p \in scalars \wedge q \in scalars$
THEN $var := (\mathsf{dom}(var) - \{p,q\}) \lhd var \cup \{(p, var(q)), (q, var(p))\}$
END

$add\_sc(p) \;\widehat{=}$
PRE $p \in (SCALARSET - scalars)$
THEN $(scalars := scalars \cup \{p\}) \;\|$
(ANY v WHERE $v \in \mathsf{INT}$ THEN $var := var \cup \{(p,v)\}$ END)
END

$rem\_sc(p) \;\widehat{=}$
PRE $p \in scalars$
THEN $scalars := scalars - \{p\} \;\|$
$var := (\mathsf{dom}(var) - \{p\}) \lhd var$
END

END

Machine definition *Scalar* had to be discarded, and all the operations corresponding to this machine 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 initilisation substitution of *Scalar* became an assigment (in fact, part of a parallel assignment) in *add_sc*, the operation that adds a new scalar in this machine.

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, 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

Because of the problems mentioned in the previous section, we suggest it is possible to provide B with a richer notation, that allows us to dynamically manage the population of abstract machines. 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 of EXTENDS. Included machines are declared to belong to an *instance set*, whose name is M_1Set (in our case *ScalarSet*). Instance sets are used to characterise live instances of machine types.

A machine equivalent to *SeveralScalars*, given in the previous section, written in our proposed extended notation is:

MACHINE
  SeveralScalars'
AGGREGATES
  Scalar
INITIALIZATION
  $ScalarSet := \emptyset$
OPERATIONS
  $swap(p, q) \;\widehat{=}$
    PRE $p \in ScalarSet \wedge q \in ScalarSet$
    THEN $p.chg(q.val) \parallel q.chg(p.val)$
    END
END

In contrast to machine *SeveralScalars*, machine *SeveralScalars'* is 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:

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

$del\_Scalar(p) \;\widehat{=}$
PRE $p \in scalars$
THEN $ScalarSet := ScalarSet - \{p\} \;\; ||$
$var := (\mathsf{dom}(var) - \{p\}) \lhd var$
END

The set NAME is assumed to be defined; 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[1]. 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*.

An extra operation, *init*, is available for instances of *Scalar*. This 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. 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.

[1] This restriction could be transformed into a type checking condition.

5 Providing Semantics to the Extension

The straightforward way to provide semantics to the syntax extension is by simply indicating that specifications like *SeveralScalars'* are syntax sugaring for an equivalent *flat* specification, like *SeveralScalars*; therefore, we could take advantage of the already well-defined semantics and consistency checkings of standard B for the syntax extension.

A *flat* version of specification M_1 which aggregates a primitive abstract machine M is constructed as follows:

- the declaration of set $MSet$, subset of NAME (specified in the INVARIANT section),
- all variables defined in M, relativised to names of instances, i.e., assuming they are of function type, we add set $MSet$ to the domain cartesian product of the corresponding operation (this is specified in the INVARIANT),
- all operations defined in M, relativised to names of instances, i.e., with an extra parameter , belonging to set $MSet$ (this is specified in the pre-condition of the corresponding operation),
- the declaration of sets, variables, invariants, operations, etc, defined explicitly in M_1; if an explicit operation of M_1 calls an operation *op* of M, then *op* must be unfolded.
- the definition of operations *add_M* and *del_M*, which are automatically generated from the definition of M (recall that *add_M* is rather complex, since it performs the "initialisation" of the added machine). We do not describe here the precise procedure employed to generate these operations, but it can be inferred by the way we defined them for specification *SeveralScalars'*.

There are just a few very basic differences between the flat *SeveralScalars'* obtained applying the above procedure and 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 big 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.

5.1 Proof Obligations for AGGREGATES

Using the approach to the semantics of the extension described above, we would be provided with a more suitable notation for dynamic population management of components, but to check for consistency we still would need to flaten specifications, and redo all checkings, instead of relying on properties of the more primitive machines. This is certainly not what we want, since we would like to treat AGGREGATES as a proper structuring mechanism, and not just as a shorthand for an unstructured flat specification. Therefore, we need to study de extra

proof obligations for the extension, on the basis of the assumption that the proof obligations of the compound machines were already discharged.

Let us consider then, a general abstract specification of a machine, which aggregates a primitive one, and with the simplifications implied by the fact we are not considering parameterised machines. An abstract primitive machine is defined below:

MACHINE
  M
SETS
  s
CONSTANTS
  c
PROPERTIES
  $PROP(s, c)$
VARIABLES
  v
INVARIANT
  $I(s, c, v)$
INITIALIZATION
  $INIT$
OPERATIONS
  $r \longleftarrow op(x) \quad \hat{=} \quad$ PRE PRE THEN S END
  $\vdots$
END

For this machine, the corresponding proof obligations are the following:

IC1 $\exists s, c : PROP(s, c)$
IC2 $PROP \rightarrow \exists v : I$
IC3 $PROP \rightarrow [INIT]I$
IC4 $(PROP \wedge I \wedge PRE) \rightarrow [S]I$

Now, suppose we have a machine specification M_1, which aggregates M, and which satisfies the visibility rules imposed by an EXTENDS to the linguistic elements from M:

MACHINE
  M_1
AGGREGATES
  M
SETS
  s_1
CONSTANTS
  c_1
PROPERTIES
  $PROP_1(s_1, c_1)$
VARIABLES
  v_1
INVARIANT

$I_1(s_1, c_1, v_1)$
INITIALIZATION
$INIT_1$
OPERATIONS
$r \longleftarrow op_1(x) \;\hat{=}\; \text{PRE}\;\; PRE_1 \;\; \text{THEN}\;\; S_1 \;\; \text{END}$
$\vdots$
END

In order to realise about which proof obligations we need to discharge to verify that M_1 is *consistent*, we generate a flat specification M_1', equivalent to M_1, in the way we explained in previous sections, and analyse its corresponding proof obligations. Machine M_1' generated from M_1 using the procedure described in previous sections is:

MACHINE
M_1'
SETS
s, s_1
CONSTANTS
c, c_1
PROPERTIES
$PROP(s, c) \wedge PROP_1(s_1, c_1)$
VARIABLES
v', v_1
INVARIANT
$I'(s, c, v') \wedge I_1(s_1, c_1, v_1)$
INITIALIZATION
$INIT_1$
OPERATIONS
$r \longleftarrow op(x, p) \;\hat{=}\; \text{PRE}\;\; PRE \wedge p \in MSet \;\; \text{THEN}\;\; S' \;\; \text{END}$
$r \longleftarrow op_1(x) \;\hat{=}\; \text{PRE}\;\; PRE_1 \;\; \text{THEN}\;\; S_1 \;\; \text{END}$
$add\_M(p) \;\hat{=}\; \ldots$
$del\_M(p) \;\hat{=}\; \ldots$
$\vdots$
END

Machine M_1' is a flat abstract machine, so its corresponding proof obligations are similar to those described above for M. Assuming that the proof obligations of M were discharged, let us study the proof obligations of M_1'

IC1 $\exists s, c, s_1, c_1 : PROP(s, c) \wedge PROP_1(s_1, c_1)$. Since we assume that proof obligations of machine M were already discharged, we know that

$$\exists s, c : PROP(s, c)$$

Therefore, this proof obligation is reduced to:

$$\exists s_1, c_1 : PROP_1(s_1, c_1)$$

IC2 $PROP \wedge PROP_1 \rightarrow \exists v', v_1 : I'(v') \wedge I_1(v_1)$. Since we know that

$$PROP \rightarrow \exists v : I(v)$$

then, and because of the way we generate I' (I' is equivalent to I, modulo the extra parameter added to certain linguistic elements of M), we know that

$$PROP \rightarrow \exists v' : I'(v')$$

is satisfied. Therefore, this proof obligation is reduced to:

$$PROP \wedge PROP_1 \rightarrow \exists v_1 : I_1(v_1)$$

IC3 $PROP \wedge PROP_1 \rightarrow [INIT_1]I' \wedge I_1$. This proof obligation cannot be reduced.

IC4 This corresponds to verifying that the operations preserve the invariant:

- $(PROP \wedge PROP_1 \wedge I' \wedge I_1 \wedge PRE \wedge p \in MSet) \rightarrow [S']I' \wedge I_1$. Part of this has been already considered in the proof obligations of machine M:

  $$(PROP \wedge I \wedge PRE) \rightarrow [S]I$$

  Therefore, and because of the way we generate S' (again, S' is equivalent to S, modulo the extra parameter added to certain linguistic elements of M), the proof obligation can be reduced to:

  $$(PROP \wedge PROP_1 \wedge I' \wedge I_1 \wedge PRE \wedge p \in MSet) \rightarrow [S']I_1$$

- $(PROP \wedge PROP_1 \wedge I' \wedge I_1 \wedge PRE_1) \rightarrow [S_1]I' \wedge I_1$. This proof obligation cannot be reduced.
- $(PROP \wedge PROP_1 \wedge I' \wedge I_1 \wedge p \in (\mathsf{NAME} - MSet)) \rightarrow [add\_body]I' \wedge I_1$. Because of the way the body of the add operation is defined, we know that, if

  $$PROP \rightarrow [INIT]I$$

  was proved, then the add operation will preserve I', again, due to the way we generate I' (recall that the add operation performs the initialisation of the aggregated machine). This proof obligation is then reduced to:

  $$(PROP \wedge PROP_1 \wedge I' \wedge I_1 \wedge p \in (\mathsf{NAME} - MSet)) \rightarrow [add\_body]I_1$$

- $(PROP \wedge PROP_1 \wedge I' \wedge I_1 \wedge p \in MSet) \rightarrow [del\_body]I' \wedge I_1$. This proof obligation cannot be reduced.

Further improvements or reductions in the proof obligations could be performed if we impose some extra (possibly type checking) conditions on specifications which make use of AGGREGATES. For instance, if I_1 cannot make any explicit reference to $MSet$, then operations add_M and del_M would trivially preserve the invariant.

6 Conclusions

We have argued about the benefits of extending the notation of the B language to support dynamic management of abstract machine population. 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 [5], 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 construst 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 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.

References

1. J.-R. Abrial, *The B-Book, Assigning Programs to Meanings*, Cambridge University Press, 1996.
2. N. Aguirre and T. Maibaum, *Reasoning about Reconfigurable Object-Based Systems in a Temporal Logic Setting*, in Proceedings of IDPT 2002, Society for Design and Process Science, 2002.
3. N. Aguirre and T. Maibaum, *A Temporal Logic Approach to the Specification of Reconfigurable Component-Based Systems*, to appear in Proceedings of the 17th International Conference Automated Software Engineering ASE 2002.
4. *The B-Toolkit User Manual*, B-Core (UK) Limited, 1996.
5. 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.
6. Digilog, *Atelier B - Générateur d'Obligation de Preuve, Spécifications*, Technical Report, RATP SNCF INRETS, 1994.
7. R.Elmstrøm, P.G.Larsen, P.B.Lassen, *The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications*, ACM Sigplan Notices, 1994.

8. K. Lano, *The B Language and Method, A Guide to Practical Formal Development*, Fundamental Approaches to Computing and Information Technology, Springer, 1996.
9. C. Jones, *Systematic Software Development Using VDM*, 2nd edition, Prentice Hall International,1990.
10. M. Spivey, *The Z Notation: A Reference Manual*, 2nd edition, Prentice Hall International, 1992.