Implementing Dynamic Aggregations of Abstract Machines in the B Method

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

[1] Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Enlace Rutas 8 y 36 Km. 601, Río Cuarto, Córdoba, Argentina
{naguirre, 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. We previously defined an extension to the B method to be able to dynamically aggregate components. The proposed extension allowed one to build specifications which can create and delete instances of machines at run time, a feature often associated with object oriented languages and not directly supported in the B method. In this paper, we study the refinement of specifications written using this extension.

We define a procedure that, given a valid implementation of an abstract machine M, systematically generates an implementation for a machine representing a dynamic aggregation of "instances" of M. Moreover, the generated implementation is guaranteed to be correct *by construction.*

Following the approach initiated in our previous work, the refinement process is defined in a way that is fully compatible with the standard B method.

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 [3][4]. As all formal methods, the B method provides a formal language to describe systems, allowing for analysis and verification of certain system properties prior to implementation. An important characteristic of B is that it covers the whole development process, from specification to implementation.

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 dynamic creation and deletion of modules or components. All structuring mechanisms of B are *static*; they allow one to define *abstract machines* (the specification components of B) whose architectural structure in terms of other components is fixed, i.e., it does not change at run time [5]. 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 [9]. Indeed, dynamic management of "objects" appears frequently and naturally when modelling software, perhaps due to the success of object oriented methodologies and programming languages. However, fully fledged object oriented extensions of B would imply a significant change to B's (rather neat) syntax and semantics, and would excessively complicate the tool support implementation (especially in relation to proof support).

In [2], an extension of the syntax of B is proposed for supporting dynamic population management of components. This extension, in contrast with other proposed extensions to model based specification languages with this feature, such as some of the object oriented extensions to Z [11] or VDM [7], is not object oriented. Moreover, it does not imply any changes to the standard semantics of B, since it can be mapped into the standard language constructs [2]. The extension is essentially the provision of an extra structuring mechanism, the AGGREGATES clause, which intuitively allows us to dynamically "link" a set of abstract machines to a certain including machine.

The aggregation relies on the generation of a *population manager* of the aggregated machines. 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 machine instances. However, these population managers are meant to be hidden from the specifier, who can use aggregation without worrying or knowing about the managers, via the "dot notation", borrowed from object orientation, and an extra combinator of substitutions, the *interleaving parallel composition*. In fact, AGGREGATES works as other conventional structuring mechanisms of B: the internal consistency[1] 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*, provided that M is internally consistent [2].

In this paper, we complement the work of [2] by showing how the new AGGREGATES clause can be refined. Since we want to keep the managers hidden from the software engineer, the refinement process consists of the systematic generation of implementations for the managers introduced by a specification using AGGREGATES. More precisely, if a machine M' "aggregates" a machine M, then, given a valid implementation of M, we generate an implementation

[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 abstract machines include conditions such as non-emptiness of the state space determined by the machine, or the preservation of the machine's invariant by the machine's operations [1].

```
MACHINE
    MinMax
VARIABLES
    y
INVARIANT
    $y \in \mathbb{F}(\mathsf{NAT}_1)$
INITIALIZATION
    $y := \emptyset$
OPERATIONS
    $ins(x)$  $\hat{=}$  PRE  $x \in \mathsf{NAT}_1$  THEN  $y := y \cup \{x\}$  END;
    $x \longleftarrow getMin$  $\hat{=}$  PRE  $y \neq \emptyset$  THEN  $x := \mathsf{min}(y)$  END;
    $x \longleftarrow getMax$  $\hat{=}$  PRE  $y \neq \emptyset$  THEN  $x := \mathsf{max}(y)$  END
END
```

Fig. 1. Abstract machine *MinMax*

of *MManager*, which is guaranteed to be internally consistent *by construction*. The generated implementation of *MManager* can then be used as part of an implementation for M'.

2 Aggregating Abstract Machines

Consider the abstract machine *MinMax* shown in Fig. 1. This is a simple variant of machine *Little_Example*, from page 508 of [1]. To illustrate the use of AGGREGATES, suppose we want a dynamic set of "instances" of machine *MinMax*, together with extra operations, to somehow "synchronise" the "min" or "max" values of two instances. This can be defined easily, by exploiting AGGREGATES and *interleaving parallel composition*, as shown in Fig. 2. The AGGREGATES M clause within a machine M' can be understood as the inclusion of a dynamic set of "instances" of M in M', in the style of EXTENDS, i.e., *promoting* the

```
MACHINE
    MultipleMinMax
AGGREGATES
    MinMax
OPERATIONS
    $syncMin(m_1, m_2)$  $\hat{=}$
        PRE  $m_1, m_2 \in MinMaxSet \wedge (m_1.y \cup m_2.y \neq \emptyset)$
        THEN
            ANY  $x$
            WHERE  $x = \mathsf{min}(m_1.y \cup m_2.y)$
            THEN  $m_1.ins(x) \parallel\!| \, m_2.ins(x)$
            END
        END;
    $syncMax(m_1, m_2)$  $\hat{=}$
        PRE  $m_1, m_2 \in MinMaxSet \wedge (m_1.y \cup m_2.y \neq \emptyset)$
        THEN
            ANY  $x$
            WHERE  $x = \mathsf{max}(m_1.y \cup m_2.y)$
            THEN  $m_1.ins(x) \parallel\!| \, m_2.ins(x)$
            END
        END
END
```

Fig. 2. Abstract machine *MultipleMinMax*, a dynamic aggregation of *MinMax* machines

operations of the included machine as part of the "interface" of the including machine.

MSet represents the set of "live instances" of M within an aggregating machine. In our case, *MinMaxSet* represents the live *MinMax* instances (see the preconditions of operations *syncMin* and *syncMax*).

Note that we employ the "dot notation" to access the variables and operations of the aggregated machines. For instance, the expression $m_1.y$ intuitively represents the value of variable y corresponding to instance m_1; analogously, the expression $m_1.ins(x)$ represents a "call" to operation *ins*, with argument x, corresponding to instance m_1 (see the definitions of operations *syncMin* and *syncMax*). As we said, the operations of the aggregated machines are promoted by the aggregating machine. For our example, this means that the operations *ins*, *getMin* and *getMax* of the live instances are accessible from the interface of *MultipleMinMax*.

Besides the operations explicitly defined in the aggregating machine M', we have available the operations originating in the aggregated machine M and two further (implicitly defined) operations. These further operations are *add_M* and *del_M*, and correspond to the population management of M, allowing for the creation and deletion of instances of machine M within M'.

2.1 The Population Managers

The AGGREGATES structuring mechanism relies on the generation of a *population manager* of the aggregated machines. A population manager of an abstract machine M puts together the relativisation of the operations of M with operations that mimic the creation and deletion of machine instances. The relativisation of the operations is a process that transforms the definitions of the operations of a basic machine so they work for multiple instances. An extra parameter is given for the relativised operations. This extra parameter represents the name of the "instance" on which the operation performs. When "calling" relativised operations, we simply put the extra parameter as a prefix, using the dot notation from object orientation.

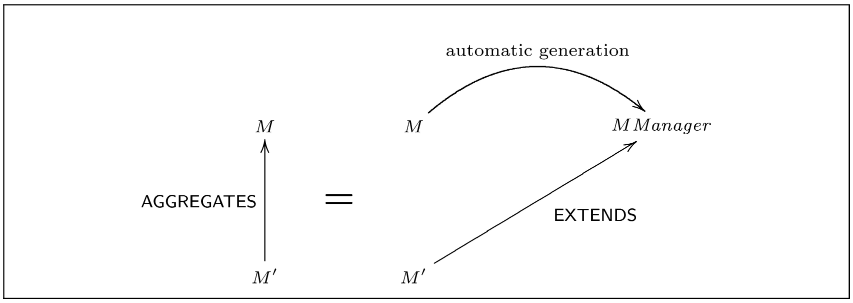

Fig. 3. The meaning of AGGREGATES in terms of EXTENDS

As the reader might imagine, AGGREGATES M within a machine definition is simply interpreted as EXTENDS *MManager*. Figure 3 illustrates this. The population manager for the *MinMax* machine is shown in Fig. 4. Note how the operations *ins*, *getMin* and *getMax* have changed.

```
MACHINE
    $MinMaxManager$
VARIABLES
    $y, MinMaxSet$
INVARIANT
    $(\forall n \cdot n \in MinMaxSet \Rightarrow y(n) \in \mathbb{F}(\mathsf{NAT}_1)) \wedge$
    $(MinMaxSet \subseteq \mathsf{NAME}) \wedge (y \in MinMaxSet \rightarrow \mathbb{F}(\mathsf{NAT}_1))$
INITIALIZATION
    $y, MinMaxSet := \emptyset, \emptyset$
OPERATIONS
    $ins(x, n) \ \hat{=}$
        PRE $n \in MinMaxSet \wedge x \in \mathsf{NAT}_1$
        THEN $y(n) := y(n) \cup \{x\}$
        END;
    $x \longleftarrow getMin(n) \ \hat{=}$
        PRE $n \in MinMaxSet \wedge y(n) \neq \emptyset$
        THEN $x := \mathsf{min}(y(n))$
        END;
    $x \longleftarrow getMax(n) \ \hat{=}$
        PRE $n \in MinMaxSet \wedge y(n) \neq \emptyset$
        THEN $x := \mathsf{max}(y(n))$
        END;
    $add\_MinMax(n) \ \hat{=}$
        PRE $n \in \mathsf{NAME} - MinMaxSet$
        THEN
            $MinMaxSet := MinMaxSet \cup \{n\} \ || \ y := y \cup \{n, \emptyset\}$
        END;
    $del\_MinMax(n) \ \hat{=}$
        PRE $n \in MinMaxSet$
        THEN
            $y := \{n\} \lhd y \ ||$
            $MinMaxSet := MinMaxSet - \{n\}$
        END;
END
```

Fig. 4. *MinMaxManager*: Population manager of abstract machine *MinMax*

The population manager of a basic machine M is *automatically constructed*. Moreover, the population manager of a machine M is internally consistent *by construction*, provided that M is internally consistent [2]. Thus, population managers can be hidden from the specifier, who can use AGGREGATES as other conventional structuring mechanisms of B.

2.2 Interleaving Parallel Composition

Note that, in machine *MultipleMinMax* (see Fig. 2), we have used a new combination of substitutions, denoted by $|||$. The reason for the use of this operator has to do with the conditions under which parallel composition is well formed. In B, a (reasonable) restriction on the parallel composition $S||T$ of two substitutions S and T is that these substitutions must not write on the same variables.

In order to understand the need for the triple bar, consider the operations $m_1.ins$ and $m_2.ins$ within a machine aggregating *MinMax* (*MultipleMinMax*, for instance). When m_1 and m_2 are different, $m_1.ins$ and $m_2.ins$ *seem* to update different variables, namely $m_1.y$ and $m_2.y$. Thus, we would like to be able to combine these substitutions in parallel, using $||$. However, what actually happens is that $m_1.ins$ and $m_2.ins$ are the same operation, called with different parameters; so, $m_1.ins$ and $m_2.ins$ modify the same variable, namely, the mapping y. For defining operations such as $syncMin(m_1, m_2)$ and $syncMax(m_1, m_2)$, we still want to somehow obtain the joint "behaviour" of $m_1.ins$ and $m_2.ins$; we could use sequential composition, but this is not allowed in B at the specification stage, for very good reasons. Hence, we decided to introduce a new combinator of substitutions, called *interleaving parallel composition*. This is a derived combinator of substitutions, whose semantics, in terms of weakest precondition, is:

$$[S|||T]P \equiv [S][T]P \wedge [T][S]P$$

Note that there are no syntactic restrictions on S and T for the well-formedness of $S|||T$. In particular, S and T can write on the same variables. Also, triple bar maintains the abstraction required at the specification stage, not allowing us to enforce sequentiality on substitutions.

We refer the reader to [2] for more details on this combinator, as well as its relationship with parallel composition.

3 The Refinement Process in the Presence of Aggregation

Our promise with aggregation during specification is that the specifier will not need to see or know about the managers, and should just use them abstractly, via the dot notation and interleaving parallel composition. However, AGGREGATES exists only at the specification stage. So, the specifier cannot, in principle, escape from the managers when considering refinements and implementations of machines with aggregation.

As explained in [1][3], the purposes of structuring a development at specification stage are different from the purposes of structuring it in the implementation. When decomposing a specification, one primarily seeks to alleviate the effort of proving consistency and to make specifications more understandable. On the other hand, the main motivations for structuring an implementation are to allow separate implementability and to promote reuse (of both library implementations and subsystems of the development). Thus, the architectural shape of a development at specification stage might be completely different from its architectural structure at implementation.

An extreme case of "architectural refactoring" during a development would be to flatten the top level machine of a specification (after proving its consistency) and then refine the resulting flat machine, introducing a new structure during the refinement process. However, in practice, both the purposes of structuring at specification and the purposes of structuring at implementation concern us throughout the whole development process (e.g., we factor implementations to make them more understandable, we decompose specifications to reuse common substructures, etc). So, one usually prefers to preserve, perhaps partially, the decomposition of the specification when refining.

Indeed, this last approach is an alternative available in the B method. In particular, the INCLUDES structure of a specification can be preserved when refining or implementing:

> "if a specification component **B** INCLUDES a machine **A** then we can use an IMPORTS **A** clause in the implementation **B_1** of **B** to carry out the functionality supplied by **A** in the specification" (cf. pp. 148-149 of [8]).

A diagrammatic presentation of the above situation is shown in Fig. 5. We refer the reader to [8] for details on the exact process needed for translating the INCLUDES into IMPORTS at implementation.

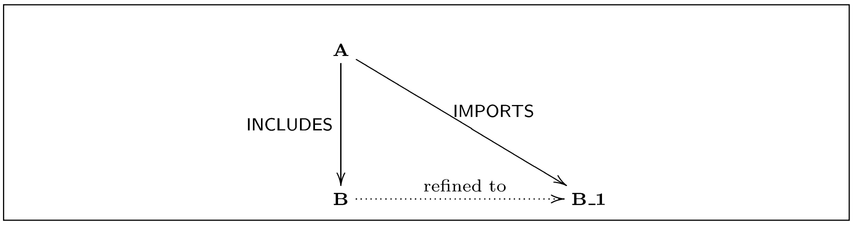

Fig. 5. The INCLUDES structure being preserved by implementations

We would like to provide the same result for our AGGREGATES clause, i.e., to somehow translate the AGGREGATES structure into a corresponding structure at implementation. For that purpose, we need to keep the managers hidden. Our intention is to make the process of refining the managers fully automated, making the use of aggregations completely transparent. Essentially, we propose the following procedure: Let M be a basic machine, and M' a machine aggregating M. M' then relies on a manager $MManager$ of M. If M is refined and eventually implemented by an implementation I_M, then we can automatically produce an implementation $I_{MManager}$ for the manager $MManager$ of M. We can then exploit the above cited result, relating INCLUDES to IMPORTS, to construct an implementation of M' based on $I_{MManager}$. Finally, what keeps the manager hidden is the fact that we generate the implementation $I_{MManager}$ of $MManager$ in such a way that its consistency is ensured, provided that I_M is a valid implementation of M. Therefore, the specifier will be able to use abstractly the dynamic aggregation of M within M', and will just need to deal with M and M' and their implementations.

3.1 Refining the Machine Managers

During a development in standard B, the whole state in implementations of abstract machines (i.e., the variables) must be imported from the state of other machines [1]. These imported (simpler) machines either *(i)* have already been refined and implemented, or *(ii)* will need to be subsequently refined and implemented, to obtain an implementation of the whole system. So, we can always assume that if a machine M is imported in some implementation, we count (or will eventually count) on an implementation of it. A set of *library machines*, for which there exist (proved correct) implementations, constitutes the *base case* of this recursive process of implementation.

Our main requirement for automatically implementing managers of arbitrary machines is the following: For every library machine M_l, there exists a provably correct implementation of its corresponding *manager* M_l*Manager*. Note that machine M_l*Manager* is systematically produced from M_l, and is consistent provided that M_l is consistent; but the implementation of M_l*Manager* will have to be provided. These managers, and their implementations, will constitute the base case for the implementations of managers for arbitrary machines.

For every machine M with a corresponding implementation I_M, an implementation of *MManager* is built via the following procedure:

```
/* ------------------------------------------------------------------ */
/* implManager(M, I_M): builds an implementation for the manager      */
/* MManager of M.                                                     */
/* Precondition: I_M is a valid implementation of M, and all machines */
/* imported in I_M are already implemented.                           */
/* Postcondition: A valid implementation of MManager is returned. The */
/* implementation of the manager is constructed from the              */
/* implementation I_M of M.                                           */
/* ------------------------------------------------------------------ */
implManager(Machine M, Implementation I_M) returns Implementation
begin
    create an "Empty Implementation" I_MManager
    add "REFINES MManager" to I_MManager
    let N1, ..., Nk be the machines imported by I_M
    for every Ni do
    begin
        let NiManager be the manager of Ni
        if (Ni is not a library machine and
            NiManager is not implemented) then
              create an impl. of NiManager via implManager(Ni, I_Ni)
        add "IMPORTS NiManager" to I_MManager
    end
    relativise the invariant in I_M and add it to I_MManager
    relativise the operations defs. in I_M and add them to I_MManager
    add operations for creation/deletion of instances, with standard
                                                           implementations
    return I_MManager
end
```

Note that the above process is recursive. Each "IMPORTS N_i" is translated into "IMPORTS $N_i Manager$", and incorporated as part of the implementation being built. In case these imported managers do not have corresponding implementations, we recursively employ the procedure to obtain implementations for them. The process is **guaranteed to terminate**, since the recursive calls will eventually reach library machines, for which, as we assume, there exist implemented managers. An important point to note is that the decision on how the deferred set NAME is implemented will have to be made only in the implementation of the managers of library machines (then it will be simply used by the rest of the implementations for managers).

The relativisation of the invariant and the operations is similar to the relativisation involved in the creation of a machine manager from an abstract machine. We will describe in detail the relativisation in implementations later on in this paper.

In order to get an intuition of the process for implementing managers, consider the following example.

Example 1. Let us consider the abstract machine *MinMax* (Fig. 1), previously introduced, and an implementation of it, shown in Fig. 6. This implementation is based on another machine, called *Pair* and shown in Fig. 7. Machine *Pair* is also implemented, as shown in Fig. 8, this time by using the library machine *Scalar*. Note that, strictly speaking, the implementation in Fig. 8 is not a proper B implementation: we are not allowed to use the suffix dot mechanism in order to import multiple instances of a machine in an implementation. However, this mechanism can be easily implemented in B, by replicating the machine specifications we need to import.

```
IMPLEMENTATION
    $I_{MinMax}$
REFINES
    $MinMax$
IMPORTS
    $Pair(\mathsf{maxint}, 0)$
INVARIANT
    $first = \mathsf{min}(\{\mathsf{maxint}\} \cup y) \;\wedge\; second = \mathsf{max}(\{0\} \cup y)$
OPERATIONS
    $ins(x) \;\hat{=}$
        VAR  $v, w$  IN
            $v \longleftarrow getFirst$;
            $w \longleftarrow getSecond$;
            IF  $x < v$  THEN  $setFirst(x)$  END;
            IF  $w < x$  THEN  $setSecond(x)$  END
        END;
    $x \longleftarrow getMin \;\hat{=}\;$ BEGIN  $x \longleftarrow getFirst$  END;
    $x \longleftarrow getMax \;\hat{=}\;$ BEGIN  $x \longleftarrow getSecond$  END
END
```

Fig. 6. An implementation of abstract machine *MinMax*

Let us apply the previous algorithm to obtain an implementation of the manager *MinMaxManager*. The resulting implementation is shown in Fig. 9.

Note how the operations and the invariant have been relativised. Also, standard implementations for the operations for creation and deletion of instances are incorporated. The exact process of relativisation will be described later on.

Since I_{MinMax} is defined in terms of *Pair*, the algorithm recursively generates an implementation for *PairManager*. Again, the generation of *PairManager* from *Pair* is systematic (and is left as an exercise for the interested reader). The obtained implementation of *PairManager*, generated from I_{Pair}, is shown in Fig. 10. The implementation I_{Pair} is based on *Scalar*, which is a library machine; $I_{PairManager}$ is then based on *ScalarManager*, for which, as we assumed, there exists a proved correct implementation.

3.2 Relativisation of Invariant and Operations of an Implementation

The algorithm for the generation of implementations for managers involves the relativisation of invariants and operations. We already had examples of these relativisations, for the implementations of *MinMaxManager* and *PairManager*. Here, we define precisely how operations and invariants are relativised.

Relativisation of Invariants. Let M be an abstract machine, with a valid implementation I_M. The invariant of I_M is an expression that relates the "state space" of M (i.e., its variables) with the "state space" of I_M (composed of the variables of the machines imported by I_M). Thus, if v are the variables of M, and $M_1, \ldots, M_k$ are machines imported in I_M, with variables $M_1.v, \ldots, M_k.v$ respectively, then the invariant of I_M has the form:

$$INV(v, M_1.v, \ldots, M_k.v)$$

The relativisation of this invariant has the purpose of relating the state space of *MManager* with the implementation for it that we generate from I_M. It has the following form:

$$(MSet = M_1Set \wedge \ldots \wedge MSet = M_kSet) \quad \wedge$$
$$[\forall n \cdot n \in MSet \Rightarrow INV(v(n), M_1.v(n), \ldots, M_k.v(n))]$$

```
MACHINE
    Pair(x, y)
VARIABLES
    first, second
INITIALIZATION
    first := x  ||  second := y
OPERATIONS
    setFirst(fst)  ≙  BEGIN  first := fst  END;
    setSecond(snd)  ≙  BEGIN  second := snd  END;
    fst ⟵ getFirst  ≙  BEGIN  fst := first  END;
    snd ⟵ getSecond  ≙  BEGIN  snd := second  END
END
```

Fig. 7. Abstract machine *Pair*

```
IMPLEMENTATION
    $I_{\mathit{Pair}}(x, y)$
REFINES
    $\mathit{Pair}$
IMPORTS
    $xx.\mathit{Scalar}(x), yy.\mathit{Scalar}(y)$
INVARIANT
    $xx.var = \mathit{first} \wedge yy.var = \mathit{second}$
OPERATIONS
    $\mathit{setFirst}(\mathit{fst}) \;\hat{=}\;$ BEGIN $xx.chg(\mathit{fst})$ END;
    $\mathit{setSecond}(\mathit{snd}) \;\hat{=}\;$ BEGIN $yy.chg(\mathit{snd})$ END;
    $\mathit{fst} \longleftarrow \mathit{getFirst} \;\hat{=}\;$ BEGIN $\mathit{fst} \longleftarrow xx.val$ END;
    $\mathit{snd} \longleftarrow \mathit{getSecond} \;\hat{=}\;$ BEGIN $\mathit{snd} \longleftarrow yy.val$ END
END
```

Fig. 8. An implementation of abstract machine *Pair*

```
IMPLEMENTATION
    $I_{\mathit{MinMaxManager}}$
REFINES
    $\mathit{MinMaxManager}$
IMPORTS
    $\mathit{PairManager}(\mathsf{maxint}, 0)$
INVARIANT
    $(\mathit{MinMaxSet} = \mathit{PairSet}) \quad \wedge$
    $[\forall n \cdot n \in \mathit{MinMaxSet} \Rightarrow$
        $(n.\mathit{first} = \mathsf{min}(\{\mathsf{maxint}\} \cup n.y) \wedge n.\mathit{second} = \mathsf{max}(\{0\} \cup n.y))]$
OPERATIONS
    $\mathit{ins}(x, n) \;\hat{=}$
        VAR $v, w$ IN
            $v \longleftarrow n.\mathit{getFirst}$;
            $w \longleftarrow n.\mathit{getSecond}$;
            IF $x < v$ THEN $n.\mathit{setFirst}(x)$ END;
            IF $w < x$ THEN $n.\mathit{setSecond}(x)$ END
        END;
    $x \longleftarrow \mathit{getMin}(n) \;\hat{=}\;$ BEGIN $x \longleftarrow n.\mathit{getFirst}$ END;
    $x \longleftarrow \mathit{getMax}(n) \;\hat{=}\;$ BEGIN $x \longleftarrow n.\mathit{getSecond}$ END;
    $\mathit{add\_MinMax}(n) \;\hat{=}\;$ BEGIN $\mathit{add\_Pair}(n)$ END;
    $\mathit{del\_MinMax}(n) \;\hat{=}\;$ BEGIN $\mathit{del\_Pair}(n)$ END
END
```

Fig. 9. The implementation of *MinMaxManager* obtained from I_{MinMax}

The first conjunct indicates how MSet is represented in the implementation of $\mathit{MManager}$, by identifying it with each of the instance sets of the imported managers. The second conjunct shows how the variables of $\mathit{MManager}$ are represented by using the representation of M's variables introduced by I_M.

Example 2. Consider the following expression:

$$\mathit{first} = \mathsf{min}(\{\mathsf{maxint}\} \cup y) \;\wedge\; \mathit{second} = \mathsf{max}(\{0\} \cup y)$$

This is the invariant of the implementation I_{MinMax}. The relativisation of this is the following:

$$\begin{aligned}
&(\mathit{MinMaxSet} = \mathit{PairSet}) \quad \wedge \\
&[\forall n \cdot n \in \mathit{MinMaxSet} \Rightarrow \\
&\qquad (\mathit{first}(n) = \mathsf{min}(\{\mathsf{maxint}\} \cup y(n)) \;\wedge\; \mathit{second}(n) = \mathsf{max}(\{0\} \cup y(n)))]
\end{aligned}$$

```
IMPLEMENTATION
    $I_{PairManager}(x, y)$
REFINES
    $PairManager$
IMPORTS
    $xx.ScalarManager(x), yy.ScalarManager(y)$
INVARIANT
    $(PairSet = xx.ScalarSet \cup yy.ScalarSet) \quad \wedge$
    $[\forall n \cdot n \in PairSet \Rightarrow (xx.var(n) = n.first \ \wedge \ yy.var(n) = n.second)]$
OPERATIONS
    $setFirst(fst, n)$ $\hat{=}$ BEGIN $xx.chg(fst, n)$ END;
    $setSecond(snd, n)$ $\hat{=}$ BEGIN $yy.chg(snd, n)$ END;
    $fst \longleftarrow getFirst(n)$ $\hat{=}$ BEGIN $fst \longleftarrow xx.val(n)$ END;
    $snd \longleftarrow getSecond(n)$ $\hat{=}$ BEGIN $snd \longleftarrow yy.val(n)$ END;
    $add\_Pair(n)$ $\hat{=}$ BEGIN $xx.add\_Scalar(n); yy.add\_Scalar(n)$ END;
    $del\_Pair(n)$ $\hat{=}$ BEGIN $xx.del\_Scalar(n); yy.del\_Scalar(n)$ END
END
```

Fig. 10. The implementation of machine *PairManager* generated from *I_Pair*

Relativisation of Operations. Let M be an abstract machine, with a valid implementation I_M. Let *op* be an operation of M, implemented in I_M. The implementation of *op* is a substitution with certain restrictions, that can be, as any substitution, reduced to the normal form as described in page 284 of [1]. So, we can say that the implementation of *op* in I_M has the following form:

$$r \leftarrow op(\overline{p}) \quad \hat{=} \quad P'(\overline{p}, M_i.v) \,|\, @\overline{x}'.(Q'(\overline{x}', \overline{p}, M_i.v) \Longrightarrow M_i.v, r := \overline{x}')$$

The relativisation of this operation has the following form:

$$\begin{aligned} & r \leftarrow op(\overline{p}, n) \quad \hat{=} \\ & \quad P'(\overline{p}, M_i Manager.v(n)) \,| \\ & \qquad\qquad @\overline{x}'.(Q'(\overline{x}', \overline{p}, M_i Manager.v(n)) \Longrightarrow M_i Manager.v(n), r := \overline{x}') \end{aligned}$$

Example 3. Consider the following operation:

```
$ins(x)$ $\hat{=}$
    VAR $v, w$ IN
        $v \longleftarrow getFirst$;
        $w \longleftarrow getSecond$;
        IF $x < v$ THEN $setFirst(x)$ END;
        IF $w < x$ THEN $setSecond(x)$ END
    END
```

This is the implementation of *ins* provided by I_{MinMax}. Its relativisation, which corresponds to an implementation of *ins* as an operation of *MinMaxManager* is the following:

```
$ins(x, n)$ $\hat{=}$
    VAR $v, w$ IN
        $v \longleftarrow getFirst(n)$;
        $w \longleftarrow getSecond(n)$;
        IF $x < v$ THEN $setFirst(x, n)$ END;
        IF $w < x$ THEN $setSecond(x, n)$ END
    END
```

Note that, in Fig. 9, we denoted this operation by using the dot notation, i.e., denoting the extra parameter n as a prefix of the corresponding expressions.

3.3 Refining Interleaving Parallel Composition

The implementation of interleaving parallel composition does not present any difficulties, since this combinator of substitutions is a *derived* one. In fact, refining $S|||T$ is equivalent to refining:

$$S;T \, [] \, T;S$$

4 Proving Consistency of Automatically Generated Implementations

Unfortunately, due to space restrictions we are unable to reproduce the proof of the fact that the implementations that we generate are correct by construction. We restrict ourselves to giving a sketch of the proof in this section.

The proof proceeds as follows. We first consider a generic abstract machine M which has a valid implementation I_M. Then, the following proof obligations indicating that I_M is a correct implementation of M are fulfilled:

a1 Correct instantiation of the parameters of imported machines in I_M,
a2 Non-emptiness of the joint state space of M and I_M,
a3 refinement of the initialisation of M by I_M,
a4 Refinement of the operations of M by I_M.

Assuming that M is internally consistent, we automatically generate the manager $MManager$ of M, which, as proved in [2], is consistent by construction; moreover, we apply the algorithm presented previously in this paper to produce an implementation for $MManager$. We then prove that the following proof obligations, indicating that $I\_MManager$ is a correct implementation of $MManager$, are satisfied as a consequence of our hypotheses:

b1 Correct instantiation of the parameters of imported machines in $I\_MManager$,
b2 Non-emptiness of the joint state space of $MManager$ and $I\_MManager$,
b3 refinement of the initialisation of $MManager$ by $I\_MManager$,
b4 Refinement of the operations of $MManager$ by $I\_MManager$.

Proof obligation b1 follows immediately from a1, since the constraints and properties of M and I_M are preserved by $MManager$ and $I\_MManager$, respectively. The non-emptiness of the joint state space of machine $MManager$ and its implementation $I\_MManager$, i.e. proof obligation b2, is easily proved by taking the instance sets $MSet, M_1Set, \ldots, M_kSet$ as empty sets. Proof obligation b3 is proved by relatively simple calculations, using the form of the initialisation of $MManager$ and its refinement in $I\_MManager$. Finally, proof obligation b4 is actually split into two parts: *(i)* refinement of the population management operations add_M and del_M, and *(ii)* refinement of the operations originating in

M. The correctness of the refinement of the population management operations requires some calculations, and concerns the general forms of these operations and their implementations. The correctness of the refinement of the operations originating in M requires more complex calculations, and uses the fact that the refinements of the operations of M by I_M are correct, i.e., proof obligation a4.

5 Conclusions

We have defined a procedure that, given an abstract machine M correctly implemented, automatically generates an implementation for a machine representing a dynamic aggregation of "instances" of M. The implementation is generated in such a way that its consistency is guaranteed by construction.

This work complements the work initiated in [2], where we proposed an extension to the B language to support dynamic creation or deletion of instances of abstract machines. The extension was based on the provision of a new structuring mechanism, AGGREGATES. Now the specifications written using the AGGREGATES clause can be implemented in a way that is fully transparent to the specifier.

The semantics of standard B is preserved by the defined extensions (including the conditions for the generation of implementations). Only some basic machinery has to be built on top of standard B.

There is some evidence of the need for complementing model oriented formal specification languages with support for some of the current systems engineering practices. The work in [12] and the various object oriented extensions to model oriented specification languages (e.g., [13], [10]), for instance, are cases in which this need becomes manifest. The proposed extension, although preliminary, builds into B support for some common activities of the current systems design practice (highly influenced by object orientation), and avoids the complexities often associated with the semantics of fully-fledged object oriented languages.

As work in progress, we are currently studying some further properties of the interleaving parallel composition operator, in particular, trying to find a stronger relationship between the standard parallel composition and triple bar. We are doing so in the context of Dunne's theory of generalised substitutions [6], which has so far simplified our proof efforts. We are also exploring ways of incorporating support for associations between dynamic instances of abstract machines, trying not to fall into the complexities of object orientation, i.e., keeping the structural organisation of machines hierarchical. Also, we are currently studying how our AGGREGATES structuring mechanism combines with other structuring mechanisms of B when the architectural organisation of a specification involves various layers of abstract machines. We believe there will be no difficulties in applying our aggregation approach to abstract machines with complex architectural organisations.

Acknowledgments

Nazareno Aguirre wishes to thank the Z User Group for supporting him and other students for their participation in the ZB2003 Conference, where some of the ideas presented in this paper evolved. The authors wish to thank Steve Dunne, Kevin Lano and the anonymous referees for their useful comments on a preliminary version of this paper.

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, Springer-Verlag, June 2003.
3. *The B-Toolkit User's Manual*, version 3.2, B-Core (UK) Limited, 1996.
4. Digilog, *Atelier B - Générateur d'Obligation de Preuve, Spécifications*, Technical Report, RATP SNCF INRETS, 1994.
5. 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.
6. 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.
7. C. Jones, *Systematic Software Development Using VDM*, 2nd edition, Prentice Hall International,1990.
8. K. Lano, *The B Language and Method, A Guide to Practical Formal Development*, Fundamental Approaches to Computing and Information Technology, Springer, 1996.
9. B. Meyer, *Object-Oriented Software Construction*, Second Edition, Prentice-Hall International, 2000.
10. G Smith, *The Object-Z Specification Language*, Advances in Formal Methods, Kluwer Academic Publishers, 2000.
11. M. Spivey, *The Z Notation: A Reference Manual*, 2nd edition, Prentice Hall International, 1992.
12. H. Treharne, *Supplementing a UML Development Process with B*, in Proceedings of FME 2002: Formal Methods– Getting IT Right, Denmark, LNCS 2391, Springer, 2002.
13. R. Holzapfel and G. Winterstein, *VDM++ – A Formal Specification Language for Object-oriented Designs*, in Proceedings of Ada-Europe Conference 1988, Ada in Industry, Cambridge University Press, 1989.