# B-ASM: Specification of ASM *à la* B

David Michel[1]*, Frédéric Gervais[2], Pierre Valarcher[2]

[1] LIX, CNRS, Polytechnique School, 91128 Palaiseau, France
dmichel@lix.polytechnique.fr
[2] LACL, Université Paris-Est
IUT Sénart Fontainebleau, Dpt. informatique, 77300 Fontainebleau, France
{frederic.gervais,pierre.valarcher}@univ-paris-est.fr

**Abstract.** We try to recover the proof correctness strength of the B method and the simplicity of the *Abstract State Machine* model (*ASM*) by constructing a B-ASM language. The language inherits from the language of substitution and from *ASM* program. The process of refinement leads us to a program expressed in the *ASM* syntax only. As each step of refinement is correct towards the specification, we obtain an *ASM* that is proved to be correct towards the specification.

**Keywords.** B Method, ASM, refinement, correctness, fixed point.

## 1 Introduction

This paper aims at extending the B language [1] in order to build ASM programs which are correct with respect to B-like logical specifications. On the one hand, the main strengths of the B formal method are: i) the ability to express logical statements, and ii) the construction of a correct implementation by refinement. On the other hand, from our viewpoint, the striking aspects of ASM are the non-bounded outer loop that can reach the fixed point of a program and the power to express naturally any kind of (sequential) algorithms.

This paper introduces a new specification language, called B-ASM, attempting to bridge the gap between these two languages, by taking advantage of the strengths of each approach. Our leitmotiv is to build an ASM which is correct with respect to a B-like specification. In that aim, we have extended the syntax and the semantics of B to take the non-bounded iteration into account. Moreover, the reuse of the well-founded theoretical relation of refinement from the B method is then straightforward. Rather than directly writing a complex ASM program, one can first specify the required logical properties of the program in a B-ASM specification. Then, we are able to build from the latter a correct ASM program, by proving the proof obligations (PO) associated to each refinement step. For instance, if we can determine a variant in the B-ASM specification for

* This author has been supported by the ANR-09-JCJC-0098-01 MaGiX project together with the Digiteo 2009-36 HD grant and région Île-de-France.

the outer loop, then the ASM program obtained by refinement is guaranteed to terminate.

In the following paragraphs, we briefly describe ASM and B.

**Abstract State Machine.** Abstract State Machines ($ASMs$) are known as a powerful theoretical tool to model (sequential) algorithms and this, at any level of abstraction of data structures ([7]). An $ASM$ is a couple $(A, \pi)$ where $A$ is an algebra (that specifies data) and $\pi$ is a program that operates over the algebra $A$. The program is a finite set of conditional rules testing (essentially) equalities over terms of the algebra and then updating the state in parallel (in the following we don't restrict the syntax of $ASM$ to be in normal form, but we use a syntax closer to the Lipari Guide [6]).

The algebra $A$ is initialized by an initial algebra and a computation is then the execution of $\pi$ until a fixed point is reached or when a clashed-update occurs (a location is updated by two different values simultaneously).

This general model of computation has been used to model a large class of problems (see [8] for tools and general purpose and [9] for a large example).

More than a practical tool, the $ASM$ model is an attempt to formalize the widely used notion of *algorithms*. The most important theoretical result is given in [7] and states that *any algorithm may be simulated step-by-step (in strict lock step) by an appropriate ASM.*

**An Overview of B.** B is a formal method [1] that supports a large segment of the software development life cycle: specification, refinement and implementation. In B, specifications are organized into abstract machines (similar to classes or modules). State variables are modified only by means of substitutions. The initialization and the operations are specified in a generalization of Dijkstra's guarded command notation, called the Generalized Substitution Language (GSL), that allows the definition of non-deterministic substitutions. For instance, in an abstract machine, we can define an operation with guarded substitutions, which are of the form **any** $x$ **where** $A$ **then** $S$ **end**, where $x$ is a variable, $A$ a first-order predicate on $x$, and $S$ a substitution. Such a substitution is non-deterministic because $x$ can be any value that satisfies predicate $A$.

The abstract machine is then refined into concrete machines, by replacing non-deterministic substitutions with deterministic ones. At each refinement step, the operations are proven to satisfy their specification. Hence, through refinement steps and proofs, the final code is proven to be correct with respect to its specification. The B method is supported by several tools, like Atelier B [5], Click'n Prove [2] and the B-Toolkit [4].

**Contribution.** *Let $M$ be a B-ASM machine, then we can construct an ASM which is correct with respect to $M$.*

This contribution is detailed in the next sections. Language B-ASM, the extension of B integrating ASM constructs, is presented in Sect. 2. Then, Sect. 3

provides a formal semantics for B-ASM, based on the weakest preconditions. Section 4 shows how to prove that an ASM program built by refinement is correct with respect to its B-ASM specification. Finally, Sect. 5 concludes the paper with some remarks and perspectives.

## 2 B-ASM Programs

ASM programs are decomposed in two parts:

- an initialization algebra, *i.e.* an initial state;
- a one-step transition function.

For executing an ASM program, the transition function is iteratively applied to the current state, starting from the initial state. The program stops when a fixed point is reached, in other words, the transition function does not alter the current state anymore.

We define B-ASM programs in the same way as in ASM programs, but the language used to define transition functions is enriched with operations akin to some non-deterministic B substitutions.

**Definition 1.** B-ASM *transition functions, i.e. B-ASM transitions, are defined by induction as follows:*

*ASM operations:*

- $f(\overrightarrow{t}) := u$ *is an B-ASM transition, where* $f(\overrightarrow{t})$ *and* $u$ *are first-order terms;*
- **if** $A$ **then** $S$ **end** *is an B-ASM transition, where* $A$ *is a formula and* $S$ *is an B-ASM transition;*
- **par** $\overrightarrow{S}$ **end** *is an B-ASM transition, where* $\overrightarrow{S}$ *is a list of B-ASM transition;*
- **skip** *is an B-ASM transition;*

*non-deterministic operations:*

- $f(\overrightarrow{t}) :\in E$ *is an B-ASM transition, where* $f(\overrightarrow{t})$ *is a first-order term and* $E$ *a set;*
- **@**$x$**.**$S$ *is an B-ASM transition, where* $x$ *is a variable and* $S$ *is an B-ASM transition;*
- **choice** $S$ **or** $T$ **end** *is an B-ASM transition, where* $S$ *and* $T$ *are B-ASM transition;*
- **any** $x$ **where** $A$ **then** $S$ **end** *is an B-ASM transition, where* $x$ *is a variable,* $A$ *a formula and* $S$ *an B-ASM transition.*

Let us now focus on the B specification language for the purposes of this paper. In B, each abstract machine encapsulates state variables (introduced by keyword **VARIABLES**), an invariant typing the state variables (in **INVARIANT**), an initialization of all the state variables (**INITIALISATION**), and operations on the state variables (**OPERATIONS**). The invariant is a first-order predicate in a simplified version of the ZF-set theory, enriched by many relational operators.

We define the B-ASM specification language as a simple modification of the B language, in order to specify ASM programs. In B-ASM, the vocabulary of algebras (*i.e.* states) is introduced by keyword **VARIABLES**, the variables are typed in the clause **INVARIANT**, the **INITIALISATION** clause contains the definition of the initial states as parallel (non-deterministic) substitutions, and the ASM transition is described in the clause **OPERATION**. In this paper, we deal with terminating programs, so we add to the syntax an additional clause called **VARIANT**. The latter defines the integer value which strictly decreases at each iteration step.

To illustrate the B-ASM approach, we consider a machine specifying the maximum of an array of integer values.

MACHINE $Maximum(tab)$
CONSTRAINTS $tab \in seq_1(\mathbb{N})$
VARIABLES $maxi$ /* Vocabulary */
INVARIANT $maxi \in \mathsf{ran}(tab)$ /* Typing */
CONSTANTS $Maxi$
PROPERTIES
$Maxi \in seq_1(\mathbb{N}) \rightarrow \mathbb{N} \ \wedge$
$\forall t \in seq_1(\mathbb{N}).(Maxi(t) \in \mathsf{ran}(t) \ \wedge \ \forall e \in \mathsf{ran}(t).(e \leq Maxi(t)))$
INITIALISATION $maxi := tab(1)$ /* Initial state */
VARIANT $Maxi(tab) - maxi$ /* Halting condition */
OPERATION /* B-ASM transition function */
$maxi := Maxi(tab)$
END

In this abstract machine, we only specify the logical properties of the expected results, without defining an algorithm to compute them. At this stage, the OPERATION clause consists of a non-deterministic B-ASM transition. In order to obtain a formalized ASM (*i.e.* without non-deterministic operations), this abstract machine has to be *refined* into a deterministic specification. Our approach consists in adapting the B refinement relation to the B-ASM method. Since the semantics of the B language is based on weakest preconditions (WP), we have to provide the WP semantics of the B-ASM transition language defined in Def. 1. This semantics will be presented in Sect. 3. In our example, the following machine is one of the possible refinements of abstract machine $Maximum(tab)$.

REFINEMENT $MaximumASM(tab)$
REFINES $Maximum(tab)$
VARIABLES $length, i, maxi$ /* Vocabulary */
INVARIANT /* Typing */
$length = size(tab) \ \wedge$
$i \in 1..length \ \wedge$
$maxi = Maxi(tab \uparrow i)$
INITIALISATION /* Initial states */
$length := size(tab)$;

```
    i := 1;
    maxi := tab(1)
VARIANT length − i                    /* Halting condition */
OPERATION                             /* ASM transition function */
    if i < length then
        par
            i := i + 1
            if tab(i + 1) > maxi then
                maxi := tab(i + 1)
            end
        end
    end
END
```

Observe that we have formalized an ASM, since there are no non-deterministic operations used in the OPERATION clause. Intuitively, the algorithm specified by the ASM computes the maximum of array *tab*. By using the refinement relation inspired from B, we can prove that this ASM faithfully implements its abstract specification.

## 3 Weakest Precondition Semantics of Transitions

To define the weakest precondition semantics, we first introduce a function $m$ which associates an integer to each list of B-ASM transitions. It will be used to define the semantics by induction.

**Definition 2.** *For all B-ASM programs $S$, we define an integer $m(S)$ by induction on $S$, and for all lists $\overrightarrow{S}$ of B-ASM programs, we will write $m(\overrightarrow{S})$ for $\sum_{S\in\overrightarrow{S}} m(S)$:*

- $m(f(\overrightarrow{t}) := u) = 0$;
- $m(\textbf{if } A \textbf{ then } S \textbf{ end}) = 1 + m(S)$;
- $m(\textbf{par } \overrightarrow{S} \textbf{ end}) = 1 + m(\overrightarrow{S})$;
- $m(\textbf{skip}) = 1$;
- $m(f(\overrightarrow{t}) :\in E) = 1$;
- $m(@x.S) = 1 + m(S)$;
- $m(\textbf{choice } S \textbf{ or } T \textbf{ end}) = 1 + m(S) + m(T)$;
- $m(\textbf{any } x \textbf{ where } A \textbf{ then } S \textbf{ end}) = 1 + m(S)$.

We now define the weakest precondition predicate for each list of B-ASM transitions. Lists are here used to take the parallel B-ASM transitions into account.

**Definition 3.** *Let $\overrightarrow{f(\overrightarrow{t}) := u}$ denote the list of substitutions $(f_i(\overrightarrow{t_i}) := u_i)_{0\le i\le n}$. For all lists $\overrightarrow{S}$ of B-ASM programs, we define the formula $[\overrightarrow{S}]P$ by induction on $m(\overrightarrow{S})$; let $\overrightarrow{Z}$ be the list $\overrightarrow{f(\overrightarrow{t}) := u}$; we consider all cases according to definition 1:*

- $[\overrightarrow{Z}]P = \begin{cases} P(\overrightarrow{f \backslash f \Lhd \{\overrightarrow{t} \mapsto u\}}) & \text{if the substitution is consistent;} \\ P & \text{if not;} \end{cases}$
- $[\overrightarrow{Z}, \textbf{if } A \textbf{ then } S \textbf{ end}, \overrightarrow{T}]P = (A \Rightarrow [\overrightarrow{Z}, S, \overrightarrow{T}]P) \wedge (\neg A \Rightarrow [\overrightarrow{Z}, \overrightarrow{T}]P)$;
- $[\overrightarrow{Z}, \textbf{par } \overrightarrow{S} \textbf{ end}, \overrightarrow{T}]P = [\overrightarrow{Z}, \overrightarrow{S}, \overrightarrow{T}]P$;
- $[\overrightarrow{Z}, \textbf{skip}, \overrightarrow{T}]P = [\overrightarrow{Z}, \overrightarrow{T}]P$;
- $[\overrightarrow{Z}, f(\overrightarrow{t}) :\in E, \overrightarrow{T}]P = \forall x. x \in E \Rightarrow [\overrightarrow{Z}, f(\overrightarrow{t}) := x, \overrightarrow{T}]P$;
- $[\overrightarrow{Z}, @x.S, \overrightarrow{T}]P = \forall x. [\overrightarrow{Z}, S, \overrightarrow{T}]P$;
- $[\overrightarrow{Z}, \textbf{choice } S \textbf{ or } T \textbf{ end}, \overrightarrow{U}]P = [\overrightarrow{Z}, S, \overrightarrow{U}]P \wedge [\overrightarrow{Z}, T, \overrightarrow{U}]P$;
- $[\overrightarrow{Z}, \textbf{any } x \textbf{ where } A \textbf{ then } S \textbf{ end}, \overrightarrow{T}]P = \forall x. A \Rightarrow [\overrightarrow{Z}, S, \overrightarrow{T}]P$.

The following theorem allows us to prove that the semantics of parallel B-ASM transitions does not depend on the order of these transitions. Indeed, if we alter the order of transitions in a list, the resulting formulas are syntactically different. However, the theorem states that these formulas are semantically equivalent.

**Theorem 1.** *For all lists $\overrightarrow{S}$ of B-ASM programs and for all permutations $\sigma$, formula $[\overrightarrow{S}]P$ is equivalent to formula $[\sigma(\overrightarrow{S})]P$.*

*Proof.* We proceed by induction on $m(\overrightarrow{S})$; we remark that for all permutations $\overrightarrow{S'}$ of $\overrightarrow{S}$ we have $m(\overrightarrow{S}) = m(\overrightarrow{S'})$. Let us write $\overrightarrow{S} = \overrightarrow{Z}, T, \overrightarrow{U}$ where $\overrightarrow{Z} = \overrightarrow{f(\overrightarrow{t}) := u}$ and $T \neq f(\overrightarrow{t}) := u$; in the same way, we write $\overrightarrow{S'} = \overrightarrow{Z'}, T', \overrightarrow{U'}$ where $\overrightarrow{Z'} = \overrightarrow{f'(\overrightarrow{t'}) := u'}$ and $T' \neq f'(\overrightarrow{t'}) := u'$. We consider as an example $T = \textbf{if } A \textbf{ then } V \textbf{ end}$ and $T' = f(\overrightarrow{t}) :\in E$; since formulas of the form $[\overrightarrow{S}]P$ are in positive occurrences in definition 3, the other cases are quite similar. By definition we have:

$$[\overrightarrow{S}]P = (A \Rightarrow [\overrightarrow{Z}, V, \overrightarrow{U}]P) \wedge (\neg A \Rightarrow [\overrightarrow{Z}, \overrightarrow{U}]P)$$

There is a permutation $\mu$ such that $\mu(\overrightarrow{Z}, V, \overrightarrow{U}) = \overrightarrow{Z}, T', V, \overrightarrow{U''}$; by induction hypothesis, $[\overrightarrow{Z}, V, \overrightarrow{U}]P$ is equivalent to $[\overrightarrow{Z}, T', V, \overrightarrow{U''}]P$. In the same way, $[\overrightarrow{Z}, \overrightarrow{U}]P$ is equivalent to $[\overrightarrow{Z}, T', \overrightarrow{U''}]P$. Thus, we have:

$$[\overrightarrow{S}]P \Leftrightarrow (A \Rightarrow [\overrightarrow{Z}, T', V, \overrightarrow{U''}]P) \wedge (\neg A \Rightarrow [\overrightarrow{Z}, T', V, \overrightarrow{U''}]P)$$

By definition, we have:

$$[\overrightarrow{Z}, T', V, \overrightarrow{U''}]P = \forall x. x \in E \Rightarrow [\overrightarrow{Z}, f(\overrightarrow{t}) := x, V, \overrightarrow{U''}]P$$

and:

$$[\overrightarrow{Z}, T', \overrightarrow{U''}]P = \forall x. x \in E \Rightarrow [\overrightarrow{Z}, f(\overrightarrow{t}) := x, \overrightarrow{U''}]P$$

Thus, according to usual boolean tautologies, we have:

$$[\overrightarrow{S}]P \Leftrightarrow \forall x. x \in E \Rightarrow (A \Rightarrow [\overrightarrow{Z}, f(\overrightarrow{t}) := x, V, \overrightarrow{U''}]P) \wedge (\neg A \Rightarrow [\overrightarrow{Z}, f(\overrightarrow{t}) := x, \overrightarrow{U''}]P)$$

By induction hypothesis, $[\overrightarrow{Z}, f(\overrightarrow{t}) := x, V, \overrightarrow{U''}]P$ is equivalent to $[\overrightarrow{Z}, V, f(\overrightarrow{t}) := x, \overrightarrow{U''}]P$ ; thus, we have:

$$[\overrightarrow{S}]P \Leftrightarrow \forall\, x.x \in E \Rightarrow [\overrightarrow{Z}, T, f(\overrightarrow{t}) := x, \overrightarrow{U''}]P$$

There is a permutation $\rho$ such that $\rho(\overrightarrow{Z}, T, f(\overrightarrow{t}) := x, \overrightarrow{U''}) = \overrightarrow{Z'}, f(\overrightarrow{t}) := x, \overrightarrow{U'}$; by induction hypothesis, $[\overrightarrow{Z}, T, f(\overrightarrow{t}) := x, \overrightarrow{U''}]P$ is equivalent to $[\overrightarrow{Z'}, f(\overrightarrow{t}) := x, \overrightarrow{U'}]P$. Hence we have $[\overrightarrow{S}]P \Leftrightarrow [\overrightarrow{Z'}, T', \overrightarrow{U'}]P$; thus $[\overrightarrow{S}]P$ is equivalent to $[\sigma(\overrightarrow{S})]P$.

## 4 Refinement of Programs

In Def. 3, we have defined the semantics for B-ASM transitions. Now, we have to define the semantics for the associated program. The latter has the same semantics as the program of the following form:

**while** $\overrightarrow{f' \neq f}$ **do**
  $\overrightarrow{f' := f}$;
  $S$;
  **if** $\overrightarrow{f' = f}$ **then** $terminate := 0$ **end**
  **invariant** $I$
  **variant** $V + terminate$
**end**

In this program, we have introduced several notations:

- $\overrightarrow{f}$ denotes the list of variables;
- $\overrightarrow{f'}$ is a list of fresh variables which are used to save the values of $\overrightarrow{f}$ from the previous state; they are of the same type as $\overrightarrow{f}$ augmented with special values that denote undefinedness. For instance, in the machine $MaximumASM(tab)$, some clauses are implicitly extended:
  - variables $length'$, $i'$, $maxi'$ and $terminate$ are added to clause VARIABLES;
  - the following formulas are in clause INVARIANT:
    $$length' = size(tab) \cup \{\bot\} \ \wedge$$
    $$i' \in 1..length' \cup \{\bot\} \ \wedge$$
    $$maxi' = Maxi(tab \uparrow i') \cup \{\bot\} \ \wedge$$
    $$terminate \in \{0, 1\}$$
  - in clause INITIALISATION, $length'$, $i'$, and $maxi'$ are initialized to $\bot$, and $terminate$ is initialized to 1;
- $I$ denotes the body of the INVARIANT clause;
- $V$ denotes the body of the VARIANT clause;
- $S$ denotes the body of the OPERATION clause.

The proof obligation associated to the above-mentioned program is derived from the classical proof obligations associated to WHILE substitutions in the B method. Let us write $B$ the loop body:

$$B \triangleq \overrightarrow{f' := f}\ ;\ S\ ;\ \textbf{if}\ \overrightarrow{f' = f}\ \textbf{then}\ terminate := 0\ \textbf{end}$$

In order to prove that the program establishes predicate $P$, we have to prove that:

1. the loop body preserves the invariant:

$$I\ \wedge\ \overrightarrow{f' \neq f} \Rightarrow [B]I \quad (PO1)$$

2. the variant is well-typed:

$$I \Rightarrow V + terminate \in \mathbb{N} \quad (PO2)$$

3. the variant strictly decreases at each iteration step:

$$I\ \wedge\ \overrightarrow{f' \neq f} \Rightarrow [n := V + terminate][B](V + terminate < n) \quad (PO3)$$

4. when the loop terminates, the program establishes predicate $P$:

$$I\ \wedge\ \overrightarrow{f' = f} \Rightarrow P \quad (PO4)$$

**Refinement Proof.** The proof obligations (PO) associated to refinement are of the following form:

1. $[Init']\neg[Init]\neg J$ (PO init)
2. $I \wedge J \Rightarrow [Subst']\neg[Subst]\neg J$ (PO op)

where $Init$ represents the initialization substitutions, $Subst$ the operation substitutions, and $I$ the invariant in the abstract machine. $Init'$, $Subst'$, and $J$ denote the counterparts of $Init$, $Subst$, and $I$, respectively, in the refinement machine. The use of negation allows non-determinism to be taken into account. These two POs guarantee that the execution of the concrete initialization (the concrete operation, respectively) is not in contradiction with the effects of the abstract initialization (the asbtract operation, resp.).

For instance, in our example dealing with the maximum of an array of integer values, the proof of (PO init) is straightforward. Double negation is not needed, because no substitution is non-deterministic in this example. Since B-ASM programs mainly consist of a WHILE loop, (PO op) requires the decomposition of $[Subst']$ into the four above-mentioned POs associated to programs.

Proof obligation $(PO1)$ can be proved by a case analysis. Either the new element in $tab$ is a new maximum, in that case, invariant $maxi = Maxi(tab \uparrow i)$ is preserved by substitution $maxi := tab(i + 1)$, or the new element is not a maximum, consequently the invariant is also preserved.

Proof obligation $(PO2)$ is straightforward.

For $(PO3)$, the proof consists in applying $i := i+1$ at each step of iteration; hence, the variant strictly decreases. Variable *terminate* allows us to decrease the variant even in the last iteration step, when $i = length$, but just before $\overrightarrow{f' = f}$.

In this example, predicate $P$ in $(PO4)$ is:

$$[maxi := Maxi(tab)](length = size(tab) \ \wedge \ i \in 1..length \\ \wedge \ maxi = Maxi(tab \uparrow i) \ )$$

The latter can be rewritten into:

$$length = size(tab) \ \wedge \ i \in 1..length \ \wedge \ Maxi(tab) = Maxi(tab \uparrow i)$$

(PO4) is straightforward, since at each iteration step, we guarantee by the invariant clause that $Maxi$ restricted to the $i$ first elements is effectively the maximum. Once all the elements are analysed, $Maxi(tab) = Maxi(tab \uparrow length)$.

## 5 Conclusion

The B method and the *Abstract State Machine* model have their own strength: proof correctness during the software development life cycle for the B method and algorithmic completeness for $ASM$ model.

By mixing the two models, we expect to conserve both the qualities of the B method and the usability of $ASMs$. For this, we add the $ASMs$ syntax to the B language of substitution and give a semantics for the weakest precondition and a semantics for the program obtained by refinement.

At the end of the process a new $B_0$ program is obtained following strictly the syntax of a $\pi$ program of an $ASM$, moreover the process has followed the proof correctness of B method refinement.

The challenge is now, to verify the efficiency of the new method in a real case study and of course, to develop tools.

## References

1. J.R. Abrial. *The B-Book: Assigning programs to meanings.* CUP, 1996.
2. Abrial, J.R., Cansell, D.: Click'n Prove: Interactive proofs within set theory. In *TPHOLs 2003, Rome, Italy*, LNCS 2758, Springer-Verlag, September 2003.
3. J.R. Abrial and L. Mussat. Introducing dynamic constraints in B. In *B'98*, volume 1393 of *LNCS*, pages 83–128, Montpellier, France, April 1998. Springer-Verlag.
4. B-Core (UK) Ltd.: B-Toolkit, `http://www.b-core.com/btoolkit.html`
5. Clearsy: Atelier B, `http://www.atelierb-societe.com`
6. Yuri Gurevich, *Evolving Algebras 1993: Lipari Guide.* Specification and Validation Methods, 1993, Oxford University Press.
7. Yuri Gurevich, *Sequential Abstract State Machines capture Sequential Algorithms.* ACM Transactions on Computational Logic, 1(1) (July 2000), 77-111.
8. E. Boerger and R. Staerk, *Abstract State Machines: A Method for High-Level System Design and Analysis*, Springer-Verlag, 2003.
9. R. Staerk, J. Schmid and E. Boerger, *Java and the Java Virtual Machine: Definition, Verification, Validation*, Springer-Verlag, 2001.