The B-Technologies : A system for computer aided programming

D.S. Neilson and I.H.Sørensen
©B-Core(UK) Limited

October 1994

Abstract

The paper introduces the B-Technologies, a mathematically based formal method and a toolset for computer aided software engineering.

The *B-Technologies* (comprising three components - the *B-Method*, the *B-Tool* and the *B-Toolkit*) have been designed to scale up formal methods for practical application. The *B-Method* and the *B-Toolkit* are described in this paper.

The *B-Method* is designed to provide a notation and a methodology for the formal specification, design, implementation and maintenance of industrial-scale software systems. The features of incremental construction of layered software as well as its incremental verification have been guiding principles in the development of the B-Method. The method uses J-R Abrial's *Abstract Machine Notation* (AMN) as the language for specification, design and implementation within the software process. AMN is is based on an extension of Dijkstra's guarded command notation, with built-in structuring mechanisms for the construction of large systems.

The *B-Toolkit* supports the method over the entire spectrum of activities from specification through design and implementation into maintenance. The B-Toolkit comprises automatic and interactive theorem-proving assistants, a proof printer and a set of software development tools: an AMN syntax & type checker, a specification animator and generators promoting an object oriented approach at all stages of development, and the re-use of specification models/software modules. All tools are integrated with the proof assistants into a window-based development environment.

0 Introduction

Formal methods of software engineering offer a number of benefits. Firstly, they enable the production of reliable software systems for which all possible behaviour can be accurately predicted and verified to meet their functional specifications. Secondly, they have the excellent potential for automation, which will reduce software production costs, and encourage standardisation. Thirdly, they encourage accountability, uniformity of approach and reusability from specification through to coding, all easing the task of software maintenance.

Existing formal methods have failed to make provision for all of these benefits, and existing applications of formal methods have thus failed to demonstrate the advantage of the approach convincingly. So far, the emphasis has been on formal specification, rather than on the entire software cycle of specification, coding and maintenance.

The B-Method [3],[5],[2],[4],[6] is a formal software development process which has been designed as a complete methodology for production of highly reliable, portable and maintainable software on an industrial scale; extremely powerful integrated tool support is provided at all stages of the development process in the form of the B-Toolkit.

This paper is based on an earlier paper written in collaboration with J-R Abrial ([1]).

1 The B-Method and the B-Toolkit

Several important principles have guided the design of the B-Method:

1. The B-Method is a mathematical method that belongs to the "model oriented" approach to software construction (VDM [7] [8] and Z [9] employ a similar approach).

2. Great attention has been paid in ensuring that the notational aspect of the method is as simple as possible. In particular, there is no real distinction between the specification notation and the programming notation; the latter appears as a restricted subset of the former.

3. The method is founded on set theory in a way which is made as solid as possible by reconstructing the original Zermelo set theory [10],[11] within the method itself. In particular, each construct of the specification (and programming) notation has a set-theoretic counterpart.

4. The method has been designed in parallel with the B-Toolkit that supports it, so that every aspect of the method is supported by providing significant help to the user.

5. Separate techniques are proposed within the method to support the design of (large) specification construction and that of (large) program construction.

6. The method is based on a series of embedded notations: the logical notation, the basic set notation, the relational notation, the mathematical object notation, the generalised substitution notation, and, at the highest level, that of the Abstract Machine.

The notation used by the B-Method relies on an extension to Dijkstra's calculus [12]. The extension allow specification of operations in terms of preconditions and postconditions, and permits object-oriented designs. Similar extensions are common within university research, eg. Morgan's Specification Statement [13].

This paper assumes a knowledge of the underlying logical and set notations, and describes some aspects of the Abstract Machine Notation (AMN) and the Generalised Substitution Language (GSL); a full description of AMN and GSL may be found in [14].

J-R Abrial has given a set theoretic model for the notation to ensure its soundness, and his book is due for publication [15].

The B-Toolkit which supports the B-Method is an integrated suite of computer programs which covers many aspects of software development, including:

1. Full configuration and dependency management of all source files, including abstract specification documents as well as binary code files.
2. Syntax and type-checking of specification documents as well as low level design documents, with comprehensive error reporting .
3. Verification condition generation (which generates the proof-obligations necessary in order to guarantee specification consistency and correctness of refinement).
4. Automatic & interactive provers for discharging the verification conditions.
5. Specification animation, enabling the specification to be 'run' and tested.
6. Code generation from declarative descriptions, facilitated by a re-usable library of code modules.
7. A Library of reusable code modules which integrate automatically with models from the reusable specification library.
8. A translator for translating low level design documents into C.
9. Rapid prototyping, facilitated by an interface generator, built on the re-usable library.
10. Automatic Markup and cross-referencing and indexing of documentation.
11. Remaking facilities when a source file is altered; the scope of the remake is determined by the toolkit, and the system is rebuilt (if possible automatically) to its former state, including code and documentation.
12. A multi-user facility for team software development.
13. All the individual tools are integrated and designed to run automatically.
14. All tools are activated from a simple to use window-based control panel; an X Motif Interface will be complete by the end of 1994.

The B-Toolkit is commercially available from B-Core(UK); a more detailed description of its components is given in section 6.

2 Abstract Machine Notation

In the B-Method, specifications, refinements and implementations are presented as Abstract Machines. Clauses within the Abstract Machine Notation declare a context of global constraints, given constants, abstract sets and their properties; further AMN clauses provide a list of abstract state variables and operations on those variables, the latter having optional preconditions, parameters and results. The operations provide the interface for the outside world to the state variables encapsulated within the Abstract Machine. Abstract Machines may be parameterised (through parameter constraints) so that instances of machines can be reused in the incremental construction of more complex machines.

2.1 The Generalised Substitution Language

In order to describe operations, a "before-after" predicate for each operation could be used, as done in VDM [8] and in Z [9]. However, the B-Method uses instead an extension of the notion of substitution (as used in mathematics) called the Generalised Substitution Language (GSL) for specifying an operation in a dynamic systems. AMN is an extended form of GSL, itself a generalisation of Dijkstra's guarded command notation [12], enhanced with preconditioning and unbounded choice, similar to the generalisations proposed in [13].

The statements in AMN (and GSL) are very concise and easy to read; there is no difference between the specification and programming notation; the generalised substitution notation, in common with all programming notations, does not refer to variables which do not change. Each generalised substitution S is defined as a predicate transformer which transforms a postcondition R into the weakest precondition for S to establish R. We write this condition as $[S]R$ and it is read : S establishes R. S may take one of the following forms: the simple substitution ($v \;\; := \;\; e$); the `skip` substitution; the *pre-conditioned* substitution ($\mid$); the *bounded choice* substitution ($[]$); the *guarded* substitution ($\Longrightarrow$); the *unbounded choice* substitution (@). These generalised substitutions are axiomatised:

GSL Syntax	Definition
$[vl := el]\, R$	substititions within R
$[\mathtt{skip}]\, R$	R
$[P \mid S]\, R$	$P \;\wedge\; [S]\, R$
$[P \Longrightarrow S]\, R$	$P \Rightarrow [S]\, R$
$[S [] T]\, R$	$[S]\, R \;\wedge\; [T]\, R$
$[@l.S]\, R$	$\forall\, l \,.\, [S]\, R$, if l is not free in R

Note that, thanks to the two choice substitutions, it is possible to specify the behaviour of a dynamic systems in a non-deterministic way: this is important, since a specification should not always give the precise and determinate details of a future systems. Such details will be added later in the refinement/implementation phases leading eventually to code.

Below, are examples of syntactic rewriting rules (sugars) allowing us a very rich "pseudo-programming" specification notation (AMN).

AMN Substitution	GSL Substitution
BEGIN S **END**	S
PRE P **THEN** S **END**	$P \mid S$
CHOICE S **OR** $\cdots$ **OR** T **END**	$S \; [] \; \cdots \; [] \; T$
IF P **THEN** S **ELSE** T **END**	$P \Longrightarrow S \; [] \; \neg P \Longrightarrow T$
IF P **THEN** S **END**	$P \Longrightarrow S \; [] \; \neg P \Longrightarrow \texttt{skip}$
ANY z **WHERE** P **THEN** S **END**	$@z \, . \, (P \Longrightarrow S)$
VAR z **IN** S **END**	$@z \, . \, S$
$x \; := \texttt{bool}(P)$	$P \Longrightarrow x := \texttt{TRUE} \; [] \; \neg P \Longrightarrow x := \texttt{FALSE}$

3 Abstract Machine Specification

Proof of internal consistency of an Abstract Machine specification requires demonstration that its context may exist - the formal parameters, constants and variables - and that within this context of the machine, the initialisation establishes the invariant, and each machine operation maintains that invariant. For example, the following Abstract Machine:

MACHINE
  $Machine\_name(x)$
CONSTRAINTS
  P
CONSTANTS
  y
PROPERTIES
  Q
VARIABLES
  z
INVARIANT
  R
INITIALISATION
  T
OPERATIONS
  $Operation\_name \;\; = \;$ **PRE** L **THEN** S **END ;**
  ...
END

gives rise to the following proof obligations:

$$\exists\, x \, . \, P$$

$$P \;\Rightarrow\; \exists\, y \, . \, Q$$

$$(P \;\wedge\; Q) \;\;\Rightarrow\;\; \exists\, z \, . \, R$$

$$(P \;\wedge\; Q) \;\;\Rightarrow\;\; [T]R$$

$$(P \;\wedge\; Q \;\wedge\; R \;\wedge\; L) \;\;\Rightarrow\;\; [S]R$$

In the above, $[S]R$ is read as "S establishes R"; the first three are concerned with consistency of contextual information (i.e if these conditions are not met then we have given a contradictory specification, which we should not attempt to implement); the fourth concerns the initialisation, and checks whether the invariant condition is satisfied initially; the last proof obligation concerns the operations (there will be one such proof obligation for each operation of the machine), and this checks whether an operation maintains the invariant condition.

3.1 Structuring and Constructing Specifications

In order to manage the complexity of the specification process it is essential that specifications can be constructed and verified in an incremental way, and that existing specifications can be used to build new specifications. Whilst in the construction of programs we are concerned with separate compilation, in the construction of large specifications from component parts we are concerned with separate verification and proof.

The key to the problem is a new generalised substitution construct, the multiple substitution ($\|$), allowing us to put together two substitutions working on distinct variables. Thus we extend the syntax of GSL to permit multiple substitution, and we may now derive such laws as:

GSL Syntax	Equivalent Syntax
$v := x \parallel w := y$	$v, w := x, y$
$S \parallel \texttt{skip}$	S
$S \parallel (P \mid T)$	$P \mid (S \parallel T)$
$S \parallel (T \; [] \; U)$	$(S \parallel T) \; [] \; (S \parallel U)$
$S \parallel (P \implies T)$	$P \implies (S \parallel T)$, *if S "terminates"*
$S \parallel @z \, . \, T$	$@z \, . \, (S \parallel T)$, *if z is not free in S*

From these properties we can show that multiple substitution, $\|$, can always can be eliminated in any formal reasoning about AMN operations.

We extend AMN with five Abstract Machine Clauses. These clauses will make operations from other machines visible, and these operations can be used in the construction of the specification of new operation (e.g. using $\|$). The clauses are described in the two sections that follow.

3.1.1 The INCLUDES, PROMOTES and EXTENDS Clauses

These concerns are provided for by employing what is known as the *semi-hiding principle*: a variable from one Abstract Machine can be included within another Abstract Machine but not modified directly from within that Abstract Machine (and so the invariant of an included machine does not add to the proof obligations of the machine in which it is included). The included variables are thus "read-only", but may appear in the invariant of the including machine. The only way in which the included variables may be changed is by "calling" one of the included operations from within the body of an operation of the including machine.

AMN allows for the inclusion of existing Abstract Machine specifications with distinct variables to be included to build new specifications, following the semi-hiding principle by the use of the INCLUDES clause. Machine parameterisation and renaming permit particular instances of different machines, as well as distinct copies of the same machine, to be included within a new machine. The variables of the new machine include the variables from each of the included machines; the invariant of the new machine includes the conjunction of the invariants of all included machines; the initialisations from the included machines are also inherited. New variables can be introduced and new invariant conditions (a "gluing invariant") can be imposed; each operation of an included machine is accessible to each operation of the new machine. The INCLUDES clause is transitive.

The PROMOTES clause contains the list of operation names of those operations of the included machines that become, without alteration, operations of the new machine. The EXTENDS clause has exactly the same form as the INCLUDES clause: all extended machines are included and, moreover, all operations of extended machines are automatically promoted to the new machine.

3.1.2 The USES and SEES Clauses

The INCLUDES mechanism does not allow for the "sharing" of one specification by several others, however, the USES and SEES constructs allow for one or more machines to have knowledge of another, provided that the used/seen and all using/seeing machines are eventually included in a single new machine. Hence neither USES nor SEES permits reference to formal parameters, and neither clause is transitive.

Machines may call operations of used/seen machines provided those operations do not change the state (i.e. they may call only "enquiry" operations). Both clauses allow for the renaming of machines. The difference between the two clauses concerns variable scope; variables of a used machine may appear in the invariant of a using machine, but variables of a seen machine may not appear in the invariant of the machine that sees it. This restriction means that a seen machine may be refined independently of a machine that sees it.

4 Abstract Machine Refinement

An Abstract Machine specification is not necessarily executable, and certainly the abstract model of the state of the machine as well as the formulation of the operations may not be directly translatable into executable code (for example, non-determinism plays an important role in many specifications). To facilitate translation into code, the Abstract Machine must be refined.

A refinement can be regarded as a step towards code; it is not the final step (see page 28), but permits a change of state in order to get closer to the data structures of the final implementation - *data refinement* - if required, and also allows for *algorithmic refinement*, and for a combination of the two.

Data refinement is achieved by a change of variables and a new set of operations which refine each of those in the original specification. The relationship between the abstract variables and the new variables is given through new invariant conditions. Algorithmic refinement permits the reformulation of operations without changing the state model.

An Abstract Machine specification does not permit sequencing: substitutions are either simple or multiple (parallel). In a refinement, however, sequencing is permitted, the GSL extension being provided by the following law:

GSL Syntax	Equivalent Syntax
$[S\ ;\ T]R$	$[S]([T]R)$

The refinement relation defined within the B-Method is based on that presented in [8],[13] and [16] and possesses some fundamental properties which are of great importance in performing incremental development and proof. The refinement relation is *transitive*, which means that the final implementation may be verified by verification of each refinement step, The refinement relation is also *monotonic* with respect to all constructs of GSL, which means that subcomponents of an operation can be refined independently.

The refinement of an Abstract Machine incurs a number of proof obligations. We consider the most general case of simultaneous data and algorithmic refinement.

MACHINE *Machine_name* **VARIABLES** x **INVARIANT** P **INITIALISATION** S **OPERATIONS** $z \longleftarrow Op\_Name =$ **PRE** Q **THEN** T **END ;** ... **END**	**REFINEMENT** *Refinement_name* **REFINES** *Machine_name* **VARIABLES** y **INVARIANT** R **INITIALISATION** U **OPERATIONS** $z \longleftarrow Op\_Name =$ **PRE** L **THEN** V **END ;** ... **END**

gives rise to the following proof obligations:

$$\exists(x, y) \cdot (P \ \wedge \ R)$$

$$[U] \neg ([S] \neg R)$$

$$\forall(x, y) \,.\, ((P \ \wedge \ R \ \wedge \ Q) \ \Rightarrow \mathrm{L})$$

$$\forall(x, y) \,.\, ((P \ \wedge \ R \ \wedge \ Q) \ \Rightarrow \ ([V'] \neg ([T] \neg (R \ \wedge \ z = z'))))$$

In the above, V' stands for substitution V within which the output variable z has been replaced by z'. The first proof obligation concerns the existence for the new variables of the refinement, and this checks whether the new invariant condition, R, contradicts the invariant condition already given - P; the second deals with the correctness of the initialisation, and it verifies that the new initialisation, U, establishes a situation where the specified initialisation, S, cannot fail to establish the condition - R. (R being the new invariant and the abstraction relationship); the third and fourth ensures the correctness of the operations (there will be a pair of such proof obligation for each operation), and it verifies that an operation, V, establishes a situation where the specified operation, T, cannot fail to maintain R, while preserving output.

5 Abstract Machine Implementation

An Abstract Machine implementation represents the last refinement and one which may be directly translated into code; thus an implementation may contain no non-determinism (unlike a refinement), complete concrete algorithms being provided for each operation. Loop constructs are permitted in an implementation and thus GSL is extended by the following syntactic rewriting rule for a while loop:

AMN Substitution	GSL Substitution
WHILE P **DO** S **END**	$(P \Longrightarrow S)\hat{}\ ; (\neg P \Longrightarrow \texttt{skip})$

For a description of ˆ, see [15].

Usually a loop is used together with an initialisation (substitution), an invariant (predicate) and, a variant (expression giving a natural number), giving the proof rule enabling us to verify loop constructs without using the definition above:

$$
\begin{array}{l}
\quad [T]Q \ \wedge \\
\quad Q \ \Rightarrow \ E \in \mathbb{N} \ \wedge \\
\quad \forall\, l \,.\, (\ (Q \wedge P) \ \Rightarrow \ [S]Q\ ) \ \wedge \\
\quad \forall\, l \,.\, (\ (Q \wedge P) \ \Rightarrow \ [n := E]([S]E) < n\ ) \ \wedge \\
\quad \forall\, l \,.\, (\ (Q \wedge \neg P) \ \Rightarrow \ R\ ) \\
\Rightarrow \\
\quad [T\ ;\ \textbf{WHILE}\ P\ \textbf{DO}\ S\ \textbf{INVARIANT}\ Q\ \textbf{VARIANT}\ E\ \textbf{END}]\,R
\end{array}
$$

The first antecedent expresses the fact that the initialisation, T, establishes the loop invariant - Q; the second that, the loop variant is a natural number expression; the third that, under the guard P, the body, S, maintains the loop invariant; the fourth that, under the guard, the body decreases the loop variant - E; the last that the post-condition, R, is established when the loop terminates, i.e. when the negation of the guard holds.

The proof obligations of an Abstract Machine implementation are exactly those give for an Abstract Machine refinement on page 27.

5.1 Structuring and Implementing Designs

An important difference between an Abstract Machine refinement and an Abstract Machine implementation is that in the latter all of the data is completely encapsulated in other machines; encapsulated variables may not be seen or used from within the algorithms of an implementation. This is known as the *full-hiding principle*, and means that an implementation need have access only to the specifications of the encapsulating machines; such machines may be independently verified, refined and implemented, so long as the specification is met. This object oriented approach has clear and considerable benefits in the construction of large software systems, where teams of designers may work in parallel on different aspects of the system. This also means that Implementations are machine independent descriptions.

Another important consideration in the design of software systems is that of reuse. A library of encapsulating machines is available to the designer, each having an Abstract Machine specification and matching implementation (for example a set machine, sequence machine and so on), to facilitate the process of reaching code from abstract specification quickly, and also to impose standards that are easily-understood and followed.

5.1.1 The IMPORTS and SEES Clauses

The IMPORTS clause implements the full-hiding principle, allowing for the introduction of encapsulating machines into an Abstract Machine implementation. As with the INCLUDES clause for specification construction, it permits machine parameterisation to permit particular instances of machines (including library machines) to be imported.

Since all data is imported, an Abstract Machine implementation may not have a variables clause, but the imported variables may be referenced in the INVARIANT clause (to enable the introduction of "gluing invariants") and also in the invariant part of a loop.

The SEES clause allows for sharing of encapsulating machines which are separately developed. An imported machine can be shared by other machines. These machines SEES the imported machine. While the importing machine has access to all the operations provided by the shared machine, the seeing machines have limited access, and cannot use the operations which modify the abstract state of the shared machine.

6 The B-Toolkit Components

The B-Toolkit provides powerful fully-integrated tool support for every aspect of the B-Method. These tools are implemented on the B-Platform (the successor to the B-Tool [17], [18], thus providing theorem-proving capabilities). The B-Toolkit and the B-Method underwent seven years of research and development at the PRG in Oxford and at BP Research in Sunbury, and development is now continuing at B-Core (UK) Ltd in Oxford, from whom the product is commercially available.

6.1 The B-Toolkit Managers

Integration of the toolkit is achieved by several cooperating managers; the two most important are the configuration manager, which incorporates a file manager and is responsible for the status of each single construct in a development, and the dependency manager, responsible for maintaining the inter-dependence between constructs.

These managers provide for the introduction (and removal) of constructs within a development, and ensure that when edits are to be committed, the extent of those edits is made known to the developer.

One of their most important functions is to ensure that all parts of the development are up-to-date, and reflect all changes that have been committed to the system (for example, it is impossible to produce code or documents that do not reflect the latest changes made in a specification). Equally important is the facility to automatically remake the system to some previous "high-water" mark once changes have been made (this process may be fine-tuned by the setting of appropriate flags).

Configuration Management is achieved through the Reset tool which, in cooperation with the B-Toolkit Managers, permits a development (at any stage) to be collapsed to its source files (which together with the toolkit dependency and configuration managers enable the complete system to be automatically rebuilt - including proof, documents and code). This facility provides for conformance with emerging standards such as BS 5750 and DEF STAN 00-55.

The B-Toolkit Managers also provide a full range of status and query functions.

6.2 Analysis

Processing of Abstract Machine specifications, refinements and implementations begin with the Analyser, providing syntax-checking and the "de-sugaring" of AMN into GSL. The former, of course, provides a rich "pseudo-programming" notation, very convenient for humans, the latter providing the concise substitution language which lends itself very well to machine processing and automation. All subsequent processing takes place on the GSL form of the construct.

The Analyser also ensures compliance with the semi-hiding and full-hiding principles.

The TypeChecker will report on the type correctness of each construct; each expression in a construct is assigned a type constructed on the given sets using the two operators $\mathbb{P}$ and $\times$ denoting respectively the power set constructor and the cartesian product constructor. Moreover, in a substitution the types of the elements situated on each side of the $:=$ operator must be the same.

6.3 Animation

It is of fundamental importance in any formal development of a computer system that the mathematics "captures" what is actually required; verification can ensure that a specification is internally consistent, or that refinement is correct with respect to its abstraction, but cannot demonstrate compliance with the (usually informal) requirements.

The B-Toolkit Animator provides an important validation facility, often overlooked in formal developments, that enables Abstract Machines to be "executed" and tested against expected behaviour. It may thus may be viewed as a very high-level prototyper (although no design need be provided by the developer).

It is envisaged that as well as a software house building computer systems on the B-Toolkit, the software procurer having the B-Toolkit could take delivery of specifications and refinements as they are written (rather than waiting for the final code to be delivered) and use validation tools such as the Animator to check progress.

6.4 Proof

One of the major advantages that a formal development method provides is the facility to provide verification conditions or proof obligations. A criticism of formal methods in the past has been the vast number of proof obligations provided by even the most modest of developments: thus automated machine support in this area is of fundamental importance.

The B-Toolkit POGenerator is an tool that automatically produces all proof obligations associated with Abstract Machine specifications, refinements and implementations. It encodes an efficient calculus implementing the rules presented on pages 21, 23, 24, 26, 27 and 28; a file of proof obligations is produced which may be subsequently presented to the B-Toolkit Provers (or, indeed, external provers if required).

The B-Toolkit provides for automated proof - the Autoprover - and for interactive proof - the InterProver. Both are built on a mathematical library containing a rich collection of mathematical laws for the underlying set-theoretic notation. The InterProver provides an environment in which the developer may supplement the library with further rules (such rules should, of course, be verified independently).

Both the POGenerator and AutoProver participate in the automatic Remake facility (if required); user-supplied theory is reused in an attempt to discharge as many of the new proof obligations as possible.

These tools also co-operate with the DocumentMarkUpTool [17] so that all proof obligations and proofs are automatically marked up, and may be either viewed independently or incorporated into the documentation of the development.

6.5 Specification/Module Library

The B-Toolkit Specification/Module Library provides a collection of off-the-shelf Abstract Machine specifications together with matching code, which may be employed in the construction of specifications (using the SEES, USES or INCLUDES facilities) or in the design of implementations (using the IMPORTS or SEES facility); such machines require no further development; in particular, they do not require analysis or proof (since they are guaranteed to be internally consistent) or refinement (since the code is provided).

The use of the library machines facilitate greatly the process of building an abstract specification and of its subsequent implementation.

Library machines provided include those for:

1. Manipulating types of objects: character string, bit string, scalar and boolean.
2. Encapsulation of a single object: variable, array, sequence, partial function and character string.
3. Encapsulation of a collection of objects: sets, sequences, strings and functions.
4. Simple i/o: terminal i/o for basic types of objects (numbers and strings etc), token i/o for entering and displaying elements of a specified type, persistent data facilities.

Each library machine provides a rich collection of operations for manipulating and accessing the encapsulated data.

A facility is also provided for the establishment of team libraries, so that specification/module pairs already developed are made accessible to other developers, as well as a facility for supplementing this system library.

All library machines may be submitted to the Animator and to the automatic InterfaceGenerator (see below).

6.6 Code Translators

The B-Toolkit Translator translates appropriate Abstract Machine implementations into object code; library modules participate fully in this process. When a development is complete, the Linker will produce executable code.

A C translator is provided; other code translators have been developed for IBM(UK) to produce PL/X and CICS code, and further translators are planned.

The Translator and Linker participate in the automatic Remake facility (if required).

6.7 Interface Generator

The InterfaceGenerator will automatically provide an interface with executable code for any Abstract Machine specification provided that it, and all subordinate constructs, have implementations built on the toolkit library; the i/o library machines will be used to provide appropriate functionality for each operation parameter.

The InterfaceGenerator thus provides a rapid-prototyping facility, and participates in the automatic Remake facility (if required).

6.8 Enumerator and Base Object Generators

The Enumerator generates the access functions associated with a given enumerated set, and an Abstract Machine specification and code modules are generated; thus the output from this generator may be viewed as a library machine.

The Base Object Generator is a powerful tool which interprets a declarative description of a database requirement (a syntactic form of entity relationship modelling) and automatically generates an Abstract Machine specification modelling that requirement, together with a rich set of operations to manipulate and access the data; an Abstract Machine implementation is also automatically generated, and built on the library object machines. The operations generated may be limited to those that are actually required in the development (this process is automatic) thus resulting in the provision of a tailor-made "instruction set".

All such generated Abstract Machines may be incorporated in the construction and design of other Abstract Machine components; they may be submitted to the Animator and to the InterfaceGenerator. The Enumerator and Base generator participate in the automatic Remake facility (if required).

6.9 Documentation

All Abstract Machine components (including interfaces, enumerations, bases, proof obligations, proofs and so on) may be submitted to the DocumentMarkUp tool which will automatically produce and compile LaTeX source code. Informal annotations may be included (which will, of course, be ignored by all other tools).

Components may be gathered together into a single Document, which provides full cross-referencing of variable, set, constant, operation and construct names at the clause and construct level, together with an index for the entire document. These features are automatic, but may be turned off.

As with all other tools, the DocumentMarkUp tool participates in the automatic Remake facility, if required.

7 Conclusion

The B-Method is designed to provide a homogeneous language, methodology and integrated tool support for the formal specification, design and implementation of real-life software systems. Therefore, the features of incremental construction and proof have been guiding principles in its development. It has now reached a mature and stable form, and a several books are in an advanced state of preparation providing a full account of the B-Method and its theoretical foundations. [15].

The B-Toolkit has been under alpha- and beta-test since January 1992, and was released in June 1994; The B-Method has been in use at GEC-Alsthom, which has for some years used an in-house tool environment (under licence from BP) to specify and develop railway control systems [19] [20],[21]. A prototype version of the B-Toolkit was used at BP International in building a fully operational graphics system [22] for visualising the flow of oil inside a refinery. MATRA Transport is using the current version of the B-Toolkit for software specification, and IBM (UK) is using it in the complete development of a new sub-system within CICS/ESA. The Parisian subway authority, RATP, and the French railways authorities, SNCF and INRETS, are funding the introduction of the B-Method into the French transport industries.

References

[1] Abrial et al, J-R., B-Technology Technical Overview, B-Core(UK) Ltd. 1993. (available from B-Core(UK) on request)

[2] Abrial, J-R., A Formal Approach to Large Software Construction, in van de Snepscheut, J.L.A (editor) Mathematics of Program Construction, Springer-Verlag, Berlin, 1989

[3] Abrial, J-R., On Constructing Large Software Systems, in van Leeuwen,J (editor), Algorithms, Software,Architecture. Information Processing 92, Vol. 1, Elsevier Science Publishers B.V., North Holland..

[4] Abrial, J-R., A Refinement Case Study (Using the Abstract Machine Notation), in *Proceedings of the Fourth Refinement Workshop*, Springer-Verlag, 1991.

[5] Lee, M.K.O., Scharbach, P.N., Sørensen, I.H., Engineering Real Software Using Formal Methods, in *Proceedings of the Fourth Refinement Workshop*, Springer-Verlag, 1991.

[6] Bicarregni, J., Richie, B., Invariants, Frames and Postconditions: a Comparison of the VDM and B Notations, in *FME'93: Industrial-Strength Formal Methods*, Springer-Verlag, 1993.

[7] Jones, C.B., *Software Development. A Rigorous Approach*, Prentice Hall, 1980.

[8] Jones, C.B., *Systematic Software Development Using VDM* Prentice Hall, Englewood Cliffs, NJ 1986.

[9] Spivey, J.M., *Understanding Z*, Cambridge University Press, 1988.

[10] Zermelo, E., *Untersuchung über die Grundlagen der Me genlehre*, 1908.

[11] Zermelo, E., *Neuer Beweis für die Möglichkeit ei er Wohlordnung der Mengenlehre*

[12] Dijkstra, E.W., *A Discipline of Programming*, Prentice Hall, 1976.

[13] Morgan, C.C., *Programming From Specifications*, Prentice Hall, 1990.

[14] B-Core (UK), *B-Method Abstract Machine Summary.*

[15] Abrial, J-R., *The B-Method*, book to appear

[16] He, J. et al, *Data Refinement Refined*, PRG Oxford,1985

[17] Abrial, J-R., *B-Tool Reference Manual*, Edinburgh Portable Compiler, 1991

[18] N. Brown, D. Mery, *A proof Environment for Concurrent Programs*, in *FME'93: Industrial-Strength Formal Methods*, Springer-Verlag, 1993.

[19] Chapront,P. Vital Coded Processor and Safety Related Software Design, in Frey,H.H. (editor). Safety of Computer Control Systems 1992 (SAFECOMP 92), Computer System in Safety Critical Applications, Proceedings of the IFAC Symposium, Switzerland. Pergamon Press, 1992.

[20] Carnot, M., DaSilva, C., Dehbonei, B., Mejia, F., Error-free Software Development for Critical Systems Using the B-Methodology, in *Proceedings of the Third IEEE International Conference on Software Reliability Engineering*, October 92, North Carolina, USA.

[21] DaSilva, C., Dehbonei, B., Mejia, F., Formal Specification in the Development of Industrial Applications: The Subway Speed Control Mechanism, in *Proceedings of the Fifth International Conference on Formal Description Techniques, FORTE'92*, October 92, Lannion, France.

[22] Blamey, W., McNeil, I., *Productivity Gains Through the Use of the B-Method*, ITRU/273, BP Internal Report, 1991.