Extending B with Control Flow Breaks

[Published in D. Bert, J.P. Bowen, S. King, and M. Waldén, Eds., *ZB 2003: Formal Specification and Development in Z and B*, vol. 2651 of *Lecture Notes in Computer Science*, pp. 513-527, Springer-Verlag, 2003.]

Lilian Burdy and Antoine Requet

Gemplus Research Lab
La Vigie, Avenue du Jujubier - ZI Athelia IV
13705 La Ciotat CEDEX - France
{lilian.burdy,antoine.requet}@gemplus.com

Abstract. This paper describes extensions of the B language concerning control flow breaks in implementations and specification of operations with exceptional behaviors. It does not claim to define those extensions in a pure formal and complete way. It is rather a presentation of what could be done and how it could be done. A syntax is proposed and proof obligations are defined using a weakest precondition calculus extended to deal with abrupt termination. Examples emphasizing the advantages of these extensions are also given.

1 Introduction

The B method [1], as a specification method, notably with its new event based features, begins to be very well known and used, in academic and research world but also in industry. Nevertheless one can obviously remark that B, as a development method, does not seem to follow the same successful story. Except for the railway transport industry, which is the historical user domain, it is really difficult to find industrials that are ready to use it to develop their software from specification to code.

We do not claim that extending B with control flow breaks is the way to change this situation, but we consider it as a little step that can bring B to be more attractive as a development method.

If we focus on the developer point of view, concerning the difficulty to adopt B, we can often remark that some reservations are expressed. Those reservations usually concern the B language itself and not the methodology. Some of them could be rubbed out if B could be presented as a real development language on which correctness proof can be done. But, for the moment, developers do not find in B all the classical language features they are used to appreciate. In this way, introducing return, break, continue and exceptions is a way to bring B0 closer to other development languages such as Ada, Java, CAML, C or C++ and to make it more attractive.

On the other hand, introducing exceptions and moreover allowing to specify exceptional behaviors can improve development clarity. This feature really provides a new way to specify operations, by clearly distinguishing normal behaviors from exceptional ones.

The remainder of this paper is organized as follows. Section 2 describes a possible syntax that allows to talk more formally and to give examples of the introduced features. Section 3 describes more in details the motivations to introduce control flow breaks in B. The semantics, consisting on a new proof obligations calculus, is described in Section 4. Section 5 gives some examples and Section 6 concludes.

2 Syntax

Before arguing on their usefulness, the syntax of the new features is introduced. We do not claim that it is the best one, but we need to define one in order to define its semantics and to give examples. The syntax is presented distinguishing two parts, the first one concerns the specification language and the second one the implementation language (B0). Associating to the syntax, some syntactical rules are given. Most of them are obvious and all can be checked statically.

2.1 Specifying exceptional behaviors

Exceptional behaviors can be specified in machines and refinements. Exceptions are declared in machines in a specific `EXCEPTIONS` clause, this clause behaves the same as the `SETS` clause, that is, all declared exceptions seen from a machine, should have different names.

In operations body, a new clause is introduced. The body of an operation should follow the rules described in the definition 1.

Definition 1. *An operation's body syntax in machine or refinement*

$$
\begin{array}{rl}
\mathit{Operation\_body} ::= & \text{PRE } \mathit{Predicate} \text{ THEN } \mathit{Substitution} \text{ END} \\
& \mid \text{BEGIN } \mathit{Substitution} \text{ END} \\
& \mid \text{PRE } \mathit{Predicate} \\
& \quad \text{THEN } \mathit{Substitution} \\
& \quad \text{EXCEPTION } \mathit{Exceptional\_behavior} \text{ END} \\
& \mid \text{BEGIN } \mathit{Substitution} \\
& \quad \text{EXCEPTION } \mathit{Exceptional\_behavior} \text{ END}
\end{array}
$$

$$
\begin{array}{rl}
\mathit{Exceptional\_behavior} ::= & \mathit{exception} \text{ WHEN } \mathit{Predicate} \text{ THEN } \mathit{Substitution} \\
& \mid \mathit{Exceptional\_behavior} \text{ ALSO } \mathit{Exceptional\_behavior}
\end{array}
$$

An operation can describe different exceptional behaviors but thrown exceptions should be distinct. Moreover all thrown exceptions in an operation should be declared in a visible `EXCEPTIONS` clause. There is no way to catch or to throw exceptions in machine, since this corresponds to implementation and not to specification matter. This implies that operations defining exceptional behavior in included or seen machines cannot be called from other machines.

2.2 Abrupt termination

The B0 implementation language is extended (see definition 2) to allow abrupt termination with a `RETURN`. One can also define labelled blocks and allow to abruptly exit them with a `BREAK`. `BREAK` and `CONTINUE` can also be used in labelled loops. Considering exceptions, they can be raised and caught in implementations.

Definition 2. *Substitution syntax extension in implementation*

Extended_Substitution ::= *Substitution*
| BREAK [*label*]
| CONTINUE [*label*]
| LABEL *label* THEN *Extended_Substitution* END
| RETURN
| RAISE *exception*
| BEGIN *Extended_Substitution*
CATCH *Catch_clause* END

Catch_clause ::= *exception* THEN *Extended_Substitution*
| *Catch_clause* WHEN *Catch_clause*

A `BREAK` corresponds to a jump to the end of the corresponding labelled block or loop. A `CONTINUE` corresponds to a return to the beginning of the loop on the test condition. `BREAK` and `CONTINUE` with label should be used in a block where this label is defined. Labels should be distinct within a method and are local to methods. Raised and caught exceptions should be declared in a visible `EXCEPTIONS` clause. Catched exceptions in a `CATCH` clause should be distinct.

3 Technical Vision

After having introduced the syntax with some syntactic rules, we describe in this section the reasons that have guided us to propose those extensions of the B language. Those reasons are multiple, nevertheless, one can focus on two points. The first one, coming from our experience in formalizing and developing with B convinces us that those extension could be useful. The second one, coming from our experience in proving Java application correctness convinces us that it was possible to generate proof obligations considering this kind of substitution.

3.1 It could be useful

From our experience developing with B : mainly the Java byte-code verifier [3, 4], other smart card applications [7] and also Meteor for one of the authors, we consider that those features are a way to ease the use of B. Those features exist in classical languages : CAML, Ada, Java and C++. They are usually good ways to reduce code size. Moreover, using exceptions usually provides more readable and maintainable code. Other control flow breaks have not this advantage as break usage for instance often gives less readable code. Nevertheless, as implementation correction is proved, there is not any reason to reduce B0 capabilities.

On the other hand, we consider that specifying exceptional behaviors can also be useful. To emphasize this point, we have rewritten the first example of the B Book concerning seat reservation. Figure 1 shows the specification of this example using exceptional behaviors. The main part of the specification is described as the normal behavior: the free seats are decreased from the booked number. And, in the appropriate clause, the exceptional behavior condition and action are also described: when the number of free seats is too small, the free seats number remains unchanged. Moreover, the operation does not need to return a parameter anymore. In the implementation, the test is done in a classical way

```
MACHINE
    booking
EXCEPTIONS
    book_failed
ABSTRACT_VARIABLES
    seat
INVARIANT
    seat ∈ ℕ
OPERATIONS
    book(nbr) ≜
    PRE
        nbr ∈ NAT
    THEN
        seat := seat - nbr
    EXCEPTION
        book_failed
            WHEN nbr > seat
            THEN skip
    END
END
```

```
IMPLEMENTATION
    booking_i
REFINES
    booking
OPERATIONS
    book(nbr) ≜
    IF nbr ≤ seat
    THEN
        seat := seat - nbr
    ELSE
        RAISE book_failed
    END
END
```

Fig. 1. Specifying and implementing with exception

and the exception is raised when it fails. Clearly identifying normal behaviors from exceptional ones can really be another way to obtain clearer and easier to read specifications.

3.2 It is possible

Working on Java applet correctness [2] have brought us to develop a tool that applies a weakest precondition calculus to Java statements. This calculus [5, 6] has been firstly used in the LOOP tool [8]. In fact, the classical Hoare logic is extended to deal with control flow breaks. Regarding this, it is possible to generate automatically proof obligations for the Java language which contains exceptions, returns, breaks and continue.

From this point of view, we have considered that it was possible to use this same calculus within the B method. Since B uses the classical Hoare logic to generate proof obligations, we have used the extended one to apply it to extended B substitutions. The next section describes the calculus applied to this previously defined syntax.

4 Semantics

After defining some notations, this section describes the extended weakest precondition calculus applied to the classical B substitutions and the newly introduced one. In the last part, proof obligations are defined dealing with this new calculus. The semantics is given in term of proof obligations and not by giving new definitions for trm and prd, since we consider that it is the more simpler to understand. For instance, $trm(S)$ and $prd_x(S)$ are in effect no more predicates since S can have different termination.

4.1 Notations

Given the set of formulas $\mathcal{F}$, the set of labels $\mathcal{L}$ and the set of exceptions $\mathcal{E}$:

- φ^{norm}, with $\varphi^{norm} \in \mathcal{F}$ corresponds to a formula that must hold in case of normal termination.
- φ^{ret}, with $\varphi^{ret} \in \mathcal{F}$ represents a formula that must hold in case of abrupt termination on RETURN.
- φ^{brk} and φ^{cont}, where $\varphi^{brk} \in \mathcal{L} \nrightarrow \mathcal{F}$ and $\varphi^{cont} \in \mathcal{L} \nrightarrow \mathcal{F}$ are partial functions mapping labels to formulas that must hold after abrupt termination, respectively on BREAK and CONTINUE.
- φ^{ex}, with $\varphi^{ex} \in \mathcal{E} \nrightarrow \mathcal{F}$ corresponds to a partial function mapping exceptions to formulas. It represents the formula that must hold when the corresponding exception occurs.

We assume the existence of a special label $\bar{l}$, the unnamed label, that is different from all the declared labels in the program. This special label is used for handling non-labelled BREAK and CONTINUE statements. In the following, φ^{brk}_{label} and φ^{cont}_{label} will be used as shortcuts to $\varphi^{brk}(label)$ and $\varphi^{cont}(label)$. In a similar way, φ^{ex}_{e} will correspond to $\varphi^{ex}(e)$ if $e \in dom(\varphi^{ex})$ and $false$ otherwise.

Using those definitions, $[S]^i(\varphi^{norm}, \varphi^{ret}, \varphi^{brk}, \varphi^{cont}, \varphi^{ex})$, where S is an extended substitution, corresponds to the necessary precondition that must hold to ensure that:

- φ^{norm} holds after S if S terminates normally;
- φ^{ret} holds after S if it terminates abruptly on a RETURN;

- for all label l defined in the context, φ_l^{brk} holds after S if it terminates abruptly on a BREAK l.
- for all label l defined in the context, φ_l^{cont} holds after S if it terminates abruptly on a CONTINUE l.
- for all exception e defined in the context, φ_e^{ex} holds after S if it terminates abruptly on a RAISE e.

In the following, we will use $X = (\varphi^{norm}, \varphi^{ret}, \varphi^{brk}, \varphi^{cont}, \varphi^{ex})$ in order to ease the notation.

4.2 Weakest precondition calculus

This section presents the extended weakest precondition calculus that is used to generate proof obligations. In order to handle exceptions and control flow breaks, we have to differentiate the cases where the program terminates normally from the cases where it terminates abruptly. We define the $[]^i$ operator (where i means implementation as opposed to machine) on extended generalized substitutions.

Definition for classical substitutions Definition 3 defines $[]^i$ for the classical substitutions. Those definitions are close to the classical ones. The definition for the sequencing operator is a bit different: it can be explained by the fact that the result of $[S_2]^i X$ is relevant only if S_1 terminates normally. Otherwise, as S_2 will never be executed, S_1 should establish the formulas concerning the abrupt terminations.

Definition 3. *Definition of $[]^i$ for the classical substitutions*

$$
\begin{array}{lcl}
[x := E]^i X & \Leftrightarrow & [x := E]\varphi^{norm} \\
[P \Longrightarrow S]^i X & \Leftrightarrow & P \Rightarrow [S]^i X \\
[S_1 [\!] S_2]^i X & \Leftrightarrow & [S_1]^i X \wedge [S_2]^i X \\
[@x.S]^i X & \Leftrightarrow & \forall x.([S]^i X) \\
[S_1; S_2]^i X & \Leftrightarrow & [S_1]^i([S_2]^i X, \varphi^{ret}, \varphi^{brk}, \varphi^{cont}, \varphi^{ex})
\end{array}
$$

Definition for operation calls We consider the operation as defined on Figure 2. When calling such an operation, the cases where the operation terminates normally must be distinguished from the cases where the operation raises an exception. This is handled by the definition of $[]^i$ for operation call given in definition 4.

Definition 4. *Definition of $[]^i$ for operation calls*

$$
[c_1, \ldots, c_n \leftarrow op(p_1, \ldots, p_m)]^i X
$$

$$
\Leftrightarrow \begin{bmatrix} arg_1, & & p_1, \\ \ldots, & := & \ldots, \\ arg_m & & p_m \end{bmatrix} \left(\begin{array}{c} P \wedge \\ (P \wedge \bigwedge\limits_{1 \leq i \leq p} (\neg W_i) \Rightarrow [[r_1, \ldots, r_n := c_1, \ldots, c_n]S]\varphi^{norm}) \wedge \\ \bigwedge\limits_{1 \leq i \leq p} (P \wedge W_i \Rightarrow [[r_1, \ldots, r_n := c_1, \ldots, c_n]T_i]\varphi_{e_i}^{ex}) \end{array} \right)
$$

$r_1, \ldots, r_n \leftarrow \mathrm{op}(arg_1, \ldots, arg_m) \triangleq$
PRE
    P
THEN
    S
EXCEPTION
    e_1 WHEN W_1 THEN T_1
...
ALSO e_p WHEN W_p THEN T_p
END

Fig. 2. Operation specification

Note that the classical definition for [] is used within the definition of operation calls, since the substitutions allowed for specification do not allow abrupt termination or raising exceptions. Note, also, that when the called operation does not declare exceptional behaviors, the definition remains valid (with p equals 0). Moreover, since the guards associated to exceptions can overlapped, each exception case leads to an independant proof obligation.

Definition for loops The definition for loops is updated to handle abrupt termination from the loop as shown on definition 5.

Definition 5. *Definition of* $[]^i$ *for loops*
[WHILE P DO S INVARIANT I VARIANT V END]$^i X$

$$\Leftrightarrow \begin{cases} I \wedge \\ \forall x.(I \wedge P \Rightarrow [S]^i \begin{pmatrix} I, \\ \varphi^{ret}, \\ \varphi^{brk} \lhd\!\!\!- \{\bar{l} \mapsto \varphi^{norm}\}, \\ \varphi^{cont} \lhd\!\!\!- \{\bar{l} \mapsto I\}, \\ \varphi^{ex} \end{pmatrix}) \wedge \\ \forall x.(I \Rightarrow V \in \mathbb{N}) \wedge \\ \forall x.(I \wedge P \Rightarrow [n := V][S]^i \begin{pmatrix} V < n, \\ \mathcal{L} \times \{true\}, \\ \mathcal{L} \times \{true\}, \\ V < n, \\ \mathcal{E} \times \{true\} \end{pmatrix}) \wedge \\ \forall x.(I \wedge \neg P \Rightarrow \varphi^{norm}) \end{cases}$$

The invariance property of the loop ensures that a terminating iteration of the loop, as well as an abrupt termination on a CONTINUE statement will establish the loop invariant. The same applies for the variant property, that must be ensured both by the normal termination and the continuing abrupt termination.

In the case of an abrupt termination of the iteration leaving the loop (that is, an operation return, an exception or a BREAK statement), proving the loop invariant is not required, but the formula corresponding to the normal behavior has to. Moreover, the variant property does not need to be proved.

Accordingly, the finalization part of the loop only requires proving the φ^{norm} formula, since it is reached only in the case of a normal termination of the loop.

Finally, the typing of the variant is left unchanged.

Definition for new substitutions The definition of $[]^i$ for the new substitutions is quite straightforward. In the case of label declaration, the definition is given in definition 6. This corresponds to adding the label to the set of known labels, and ensuring that the current φ^{norm} holds when the block is exited abruptly.

Definition 6. *Definition of* $[]^i$ *for labels*

$$\begin{bmatrix} \text{LABEL } l \\ \text{THEN } S \\ \text{END} \end{bmatrix}^i X \Leftrightarrow [S]^i \begin{pmatrix} \varphi^{norm}, \\ \varphi^{ret}, \\ \varphi^{brk} \lhd\!\!\!+ \{l \mapsto \varphi^{norm}\}, \\ \varphi^{cont}, \\ \varphi^{ex} \end{pmatrix}$$

Labeled loops Labels enclosing a loop are treated as special cases since the execution of the loop can be resumed using the CONTINUE *label* keyword. The definition of $[]^i$ for those loops is given in definition 7. This definition is very close to the one used for classical loops (Definition 5), except that it uses both the unnamed label $\bar{l}$ and the defined label l.

Definition 7. *Definition of* $[]^i$ *for labeled loops*

$$[\text{ LABEL } l \text{ THEN WHILE } P \text{ DO } S \text{ INVARIANT } I \text{ VARIANT } V \text{ END END}]^i X$$

$$\Leftrightarrow \begin{cases} I \land \\ \forall x.(I \land P \Rightarrow [S]^i \begin{pmatrix} I, \\ \varphi^{ret}, \\ \varphi^{brk} \lhd\!\!\!+ \{l \mapsto \varphi^{norm}, \bar{l} \mapsto \varphi^{norm}\}, \\ \varphi^{cont} \lhd\!\!\!+ \{l \mapsto I, \bar{l} \mapsto I\}, \\ \varphi^{ex} \end{pmatrix}) \land \\ \forall x.(I \Rightarrow V \in \mathbb{N}) \land \\ \forall x.(I \land P \Rightarrow [n := V][S]^i \begin{pmatrix} V < n, \\ \mathcal{L} \times \{true\}, \\ \mathcal{L} \times \{true\}, \\ V < n, \\ \mathcal{E} \times \{true\} \end{pmatrix}) \land \\ \forall x.(I \land \neg P \Rightarrow \varphi^{norm}) \end{cases}$$

The case definition for control-flow breaks is given by the definition 8. It simply selects the relevant formula to prove. The definition for the BEGIN CATCH END substitution corresponds to the precondition of S with the exception handlers added to φ^{ex}.

Definition 8. *Definition of* $[]^i$ *for control-flow breaks*

$$[\text{ CONTINUE label }]^i X \Leftrightarrow \varphi^{cont}_{label}$$
$$[\text{ BREAK label }]^i X \Leftrightarrow \varphi^{brk}_{label}$$
$$[\text{ RETURN }]^i X \Leftrightarrow \varphi^{ret}$$
$$[\text{ RAISE } e]^i X \Leftrightarrow \varphi^{ex}_{e}$$

$$\left[\begin{array}{l} \text{BEGIN } S \\ \text{CATCH } e_1 \text{ THEN } C_1 \\ \ldots \\ \text{WHEN } e_n \text{ THEN } C_n \\ \text{END} \end{array}\right]^i X \Leftrightarrow [S]^i \left(\begin{array}{l} \varphi^{norm}, \\ \varphi^{ret}, \\ \varphi^{brk}, \\ \varphi^{cont}, \\ \varphi^{ex} \lhd\!\!- \bigcup_{1 \le i \le n} \{e_i \mapsto [C_i]^i X\} \end{array}\right)$$

4.3 Proof obligations

We have defined in the previous section the new operator $[]^i$. We are now defining the new proof obligations calculus in machines, refinements and implementations.

Machine proof obligations As substitutions with control flow breaks are only usable in implementation, one can keep the classical B operator [] to generate machine proof obligations. We only have to extend it with the new introduced machine substitution that corresponds to an exceptional behavior description (see definition 9).

Definition 9. *Extending* [] *definition*

$$\left[\begin{array}{l} \text{BEGIN} \\ \quad S \\ \text{EXCEPTION} \\ \quad e_1 \text{ WHEN } W_1 \text{ THEN } T_1 \\ \ \ldots \\ \ \text{ALSO} \\ \quad e_n \text{ WHEN } W_n \text{ THEN } T_n \\ \text{END} \\ \end{array}\right] P \Leftrightarrow \left\{\begin{array}{l} \bigwedge_{1 \le i \le n} (W_i \Rightarrow [T_i]P) \ \wedge \\ \bigwedge_{1 \le i \le n} (\neg W_i) \Rightarrow [S]P \end{array}\right.$$

This definition looks like the definition for a `SELECT` clause where the normal substitution S corresponds to the `ELSE` branch and each exception description to a `WHEN` branch. So, a way of understanding it is that if an exception condition W_i is valid, then the corresponding exceptional behavior will occur, otherwise if no exception condition is valid, then the normal behavior will occur. Moreover, one can notice that exceptional conditions do not have to be distinct and obviously should not cover all the cases. For the other substitutions, the proof obligations calculus remains unchanged.

Refinement proof obligations To generate proof obligations for refinement, one has to take into account exceptional behaviours. Exceptional behaviours can be refined but an operation that declares exceptions can only be refined by an operation that declares the same exceptions or fewer exceptions. This means that one can suppress exceptional behaviours during the refinement but not create new ones.

BEGIN
    S'
EXCEPTION
    e_1 WHEN W'_1 THEN T'_1
...
ALSO e_p WHEN W'_p THEN T'_p
END

Fig. 3. Operation specification

Definition 10. *Exceptional behaviour refinement proof obligations*

When an operation with body as described figure 3 refines an operation in a machine or a refinement that declares exceptional behaviours (see a description figure 2), the proof obligation is

$$\begin{cases} \bigwedge_{1 \leq i \leq p} W'_i \Rightarrow W_i \\ \bigwedge_{1 \leq i \leq p} (\neg W_i) \wedge \bigwedge_{1 \leq i \leq p} (\neg W'_i) \Rightarrow [S']\neg[S]\neg I \\ \bigwedge_{1 \leq i \leq p} (W_i \wedge W'_i \Rightarrow [T'_i]\neg[T_i]\neg I) \end{cases}$$

Note that the exceptions that are not refined can be considered as refined with $false$ as guard. Note also that the guards can get stronger during the refinement.

Implementation proof obligations To generate refinement proof obligations for implementations, one has to take into account control flow breaks. This means that the classical refinement proof obligation ($[T]\neg[S]\neg I$ when T refines S under invariant I) is no longer valid, since the $[]^i$ operator should be used. The definitions 11 and 12 define refinement proof obligations depending on the refined substitution.

Definition 11. *Normal behaviour implementation proof obligations.*
When an operation with body T in an implementation refines an operation with body S that does not declare exceptional behaviours, the proof obligation is

$$[T]^i \begin{pmatrix} \neg[S]\neg I, \\ \neg[S]\neg I, \\ \emptyset, \\ \emptyset, \\ \emptyset \end{pmatrix}$$

If the refined substitution does not declare an exceptional behaviour, then the implementation has to ensure that it can only terminates normally or on a `RETURN`. In those two cases, one has to ensure classical refinement correctness; all other cases are initialized with the empty set corresponding to the fact that no formula has to be proved. It will lead to false if such abrupt termination occurs.

Definition 12. *Exceptional behaviour implementation proof obligations*

When an operation with body T in an implementation refines an operation that declares exceptional behaviours (see a description figure 2), the proof obligation is

$$[T]^i \begin{pmatrix} \bigwedge_{1 \leq i \leq p} (\neg W_i) \wedge \neg[S]\neg I, \\ \bigwedge_{1 \leq i \leq p} (\neg W_i) \wedge \neg[S]\neg I, \\ \emptyset, \\ \emptyset, \\ \bigcup_{1 \leq i \leq p} \{e_i \mapsto W_i \wedge \neg[S_i]\neg I\} \end{pmatrix}$$

If the refined substitution declares exceptional behaviours, the implementation should respect them. This means, that if it terminates normally or abruptly on a `RETURN`, one should prove that no exceptional behaviour condition is valid and that the refinement is correct in the classical sense. One also has to prove that if the refinement terminates abruptly on an exception, then the condition of this exception is valid and the refinement refines correctly the exceptional behaviour corresponding to this exception. Moreover, abrupt termination on `BREAK` or `CONTINUE` is still not valid for a method.

Proof obligations calculus becomes quite more complex than it was previously. But our experience generating proof obligations for Java has shown that after the calculation, generated proof obligations remain with the same complexity.

5 Examples

This section provides little examples that highlight the advantages of having control flow break substitutions.

5.1 Loop termination

The example given figure 4 describes two implementations of a loop. This loop implements the search of an element a in the range of an integer array t with domain 0..10. On the left side, a classical B operation is shown, the right side shows an implementation with a return inside the body of the loop.

```
res ← search(a) ≜
BEGIN
    res := FALSE;
    i := 0;
    WHILE i ≤ 10 ∧
            res = FALSE DO
        IF t(i) = a
        THEN
            res := TRUE
        END;
        i := i + 1
    INVARIANT
        i ∈ 0..11 ∧
        res = bool(a ∈ t[0..i-1])
    VARIANT
        11 - i
    END
END
```

```
res ← search(a) ≜
BEGIN
    i = 0;
    WHILE i ≤ 10 DO
        IF t(i) = a
        THEN
            res := TRUE;
            RETURN
        END;
        i := i + 1
    INVARIANT
        i ∈ 0..11 ∧
        a ∉ t[0..i-1]
    VARIANT
        11 - i
    END;
    res := FALSE
END
```

Fig. 4. Loop termination

One can consider two improvements when using abrupt exiting from a loop:

- the loop stop condition can be simplified, one does not need to test if the value has been found anymore. This allows to reduce code size and execution time.
- the invariant is simplified too, since we can consider that if we are in the loop, it is only if the searched for value has not been found. While the classical invariant should deal with the two cases : already found or not, the new invariant only deals with the second one.

One can consider that those advantages are not so important but in big development, it can really save code size, execution time and development time.

5.2 Exceptional behaviors

The examples (figures 5 and 6) show the advantages to have the possibly of specifying a method with exceptional behaviors. The first example shows the specification of a read operation on a partial function. On the left side, the classical specification is given, the right side describes the specification with an exceptional behavior. The second specification can be understood as: the operation returns the value of the function for this index and exceptionally if the index does not belong to the function domain, an exception will be raised and the returned value is not relevant.

The figure 6 describes the use of the previous read method. The implemented specification is given on top and two implementations are given. The left one has to test the validity parameter before using the returned value. The right one implements the test in normal way and in the catch part treats the exceptional behavior by returning false. Even in this simple example where two values are read sequentially, it is clear that the code with exception is more readable than the classical code. If one takes an example with more calls to the read function, the classical B code would be even harder to follow whereas the code with exception would not lose its readability.

Moreover, in such cases, tests are performed twice: once by the read function, in order to initialize the result value, and a second one by the caller of the read operation, in order to check wether the read operation succeeded. When using exception treatment, in the normal case, that can be considered as the usual one, the test has only to be performed in the read function.

It is really obvious to demonstrate that programming with exception provides clearer code. The new point on which we argue here is that specifying with exceptional behaviors allows also to obtain clearer specifications.

6 Conclusion

In this paper, an extension of the B language corresponding to the control-flow breaks in implementation and the specification of operations with exceptional behavior is presented.

It shows that those extensions could be useful for writing implementations, but also for differentiating the normal behavior from the exceptional one in specifications. It can also be a way to obtain more efficient code when translating B0, even if the introduced features do not exist in all targeted language.

Moreover, a surprising result is that those extensions also allow writing simpler loop invariants and can ease the proof process. This was unexpected, since control-flow breaks require more complicated handling.

Introducing those features has the side effect that B0 is not a subset of the languages supported by the converter anymore. More exactly, C does not have exception, and Ada has more restricted control flow breaks. For those languages, it could be easily possible to handle those features in the converter: for example, converting control flow breaks into Ada could use exceptions, and the `setjmp/longjmp` feature of C could be used to handle exceptions. However, this complexifies the converter, so if this complexity increase is an issue, it could also be possible to add additional B0 checks depending on the target language used.

```
is_valid, value ← read(i) ≙
PRE
    i ∈ INT
THEN
    IF i ∈ dom(t)
    THEN
        is_valid := TRUE ||
        value := t(i)
    ELSE
        is_valid := FALSE ||
        value :∈ INT
    END
END
```

```
value ← read(i) ≙
PRE
    i ∈ INT
THEN
    value := t(i)
EXCEPTION
    outofdomain
    WHEN
        i ∉ dom(t)
    THEN
        value :∈ INT
END
```

Fig. 5. Exceptional behaviors

```
res ← test(i,j) ≙
PRE
    i ∈ INT ∧
    j ∈ INT
THEN
    res := bool(i ∈ dom(t) ∧
                j ∈ dom(t) ∧
                t(i) = t(j))
END
```

```
res ← test(i,j) ≙
BEGIN
    res, l1 ← read(i);
    IF res = TRUE
    THEN
        res, l2 ← read(j);
        IF res = TRUE
        THEN
            res := bool(l1 = l2)
        END
    END
END
```

```
res ← test(i,j) ≙
BEGIN
    l1 ← read(i);
    l2 ← read(j);
    res := bool(l1 = l2)
CATCH
    outofdomain
    THEN
        res := FALSE
END
```

Fig. 6. Catching exceptions

So B0 using control flow breaks could not be converted to Ada and B0 with exceptions could not be converted to C.

Finally, although the new event B can reduce the need to directly specify exceptional behavior at the start, we consider that exceptional behavior and control-flow breaks are a complementary feature, that would prove useful for providing the link between event B and the classical B required for implementation.

References

1. Jean-Raymond Abrial. *The B Book, Assigning Programs to Meanings.* Cambridge University Press, 1996.
2. Lilian Burdy and Antoine Requet. Jack : Java Applet Correctness Kit. In *GDC 2002, Singapore*, November 2002.
3. Ludovic Casset. Development of an Embedded Verifier for Java Card Byte Code using Formal Methods. In Lars-Henrik Eriksson and Peter Alexander Lindsay, editors, *Formal Methods – Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 290–309. Springer-Verlag, July 22-24 2002.
4. Ludovic Casset, Lilian Burdy, and Antoine Requet. Formal Development of an Embedded Verifier for Java Card Byte Code. In *DSN 2002, International Conference on Dependable Systems & Networks*, pages 51–56, Washington, D.C., USA, June 2002.
5. Marieke Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle.* PhD thesis, University of Nijmegen, The Netherlands, 2001.
6. Marieke Huisman and Bart Jacobs. Java Program Verification via a Hoare Logic with Abrupt Termination. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE)*, volume 1783, pages 284–303. Springer-Verlag, 2000.
7. Pierre Lartigue and Denis Sabatier. The use of the B formal method for the design and the validation of the transaction mechanism for smart card applications. In *Formal Methods in System Design, Special Issue on FM'99*, November 1999.
8. Joachim van den Berg and Bart Jacobs. The LOOP Compiler for Java and JML. *Lecture Notes in Computer Science*, 2031:299–312, 2001.