Transforming EVENT B Models into Verified C# Implementations

Dominique Méry[1] and Rosemary Monahan[2]

[1] Université de Lorraine
LORIA, BP 239, 54506 Vandœuvre lès Nancy, France
mery@loria.fr

[2] Department of Computer Science,
National University of Ireland, Maynooth, Co. Kildare, Ireland
rosemary.monahan@nuim.ie

Abstract

The refinement-based approach to developing software is based on the *correct-by-construction* paradigm where software systems are constructed via the step-by-step refinement of an initial high-level specification into a final concrete specification. Proof obligations, generated during this process are discharged to ensure the consistency between refinement levels and hence the system's overall correctness.

Here, we are concerned with the refinement of specifications using the EVENT B modelling language and its associated toolset, the RODIN platform. In particular, we focus on the final steps of the process where the final concrete specification is transformed into an executable algorithm. The transformations involved are (a) the transformation from an EVENT B specification into a concrete recursive algorithm and (b) the transformation from the recursive algorithm into its equivalent iterative version. We prove both transformations correct and verify the correctness of the final code in a static program verification environment for C# programs, namely the Spec# programming system.

1 Introduction

EVENT B is a formal modelling language developed by Abrial [1]. Key features of EVENT B are the use of set theory as a modelling notation, the use of refinement to represent software systems at different abstraction levels and the use of mathematical proof to verify consistency between refinement levels. This mathematical proof is typically achieved in a semi-automated way, with the user interacting with theorem proving tools using the RODIN platform. The final concrete representation of the system results from discharging accumulated proof obligations, which are recorded as invariants of the system under development.

In this paper, we focus on the transformation of the final concrete specification into an executable algorithm. We present the transformations for (a) transforming an EVENT B specification into a recursive algorithm and (b) transforming from that recursive program to an iterative version of the same program. We prove both transformations correct and verify the correctness of the final code in a static program verification environment for C# programs, namely the Spec# programming system. This work is a component of our general framework for integrating two popular approaches to formal software development. In this framework we combine the efforts of program refinement as supported by EVENT B and program verification as supported by the Spec# programming system. The architecture induces a methodology [12], which improves the usability of formal verification tools for the specification, the construction and the verification of correct sequential algorithms.

In the sections that follow, we provide an overview of EVENT B, our integrated development framework and the transformations that form an essential component of the transformation of concrete specifications into an executable recursive program. Finally, we present the iterative program verified in the automatic program verification environment of the Spec# programming system.This verification ensures that the generated program is correct with respect to the initial EVENT B abstract model.

2 The EVENT B Modelling Framework

EVENT B [1] is a formal method for system-level modelling and analysis. An EVENT B model is defined via *contexts* and *machines*. As shown in Figure 1, *machines* express dynamic information about the model via *events*, which modify state variables that are defined in the *contexts*. Machines may also express other properties, such as invariant and safety properties of the model.

An event is equivalent to a *reactive action* waiting for a condition (called a guard) to hold in order to trigger an action. It has three main parts: a list of local parameters, a guard G and a relation R over values of state variables denoted *pre*-values (x) and *post*-values (x'). When the guard holds the actions in the event body modify the state variables according to the relation R. The *before–after* predicate $BA(e)(x, x')$ associated with each event describes the event as a logical predicate for expressing the relationship linking values of the state variables just before, and just after, the *execution* of event e. We indicate the ith action in each event using the prefix $acti$. The most common event representation has the form

$$\text{ANY } t \text{ WHERE } G(t,x) \text{ THEN } x : |(R(x,x',t)) \text{ END}$$

where t is a local parameter and the event actions establish $x : |(R(x,x',t))$. The form is semantically equivalent to $\exists\, t\cdot\ (G(t,x)\ \wedge\ R(x,x',t))$.

```
MACHINE specquare
SEES square0
VARIABLES
  r
INVARIANTS
  inv1 : r ∈ ℕ
EVENTS
EVENT INITIALISATION
  BEGIN
    act1 : r := 0
  END
EVENT square_computing
  BEGIN
    act1 : r := n ∗ n
  END
END
```

```
CONTEXT square0
CONSTANTS
  n
AXIOMS
  axm1 : n ∈ ℕ
END
```

- *square0* is a context defining properties of a natural number n
- *specsquare* is a machine with an event square_computing computing the square function for n and assigning the value to r.
- The SEES clause related the context and the machine.

Figure 1: EVENT B structure: context and machine

These basic structures are extended by the refinement process, which provides a mechanism for relating an abstract model and a concrete model by adding new events or by adding new variables. This mechanism allows the gradual development of EVENT B models and the validation of each decision step. The refinement of a formal model allows us to enrich our formal reactive models via a *step-by-step* approach and is the foundation of our correct-by-construction approach [7]. Refinement provides a way to strengthen invariants and to add details to a model. It is also used to transform an abstract model to a more concrete version by modifying the state description.

Refinement is achieved by extending the list of state variables (and possibly suppressing some of them), by refining each abstract event to a set of possible concrete versions, and by adding new events. The abstract (x) and concrete (y) state variables are linked by means of a *gluing invariant* $J(x, y)$, which must be maintained throughout the system modelling. A number of proof obligations ensure that each abstract event is correctly refined by its corresponding concrete version, each new event refines $skip$, no new event takes control forever and relative deadlock freedom is preserved. The refinement relationship is expressed as follows: a model M is refined by a model P, when P simulates M. The final concrete model is close to the behaviour of the final software system that executes events using real source code. In this paper we present the translation of these concrete models to recursive and iterative algorithms that can be directly mapped to code.

The EVENT B modelling language is supported by the Atelier B [3] environment and by the RODIN platform [14] . Atelier B and the RODIN platform both provide facilities for editing contents and machines, refinements, contexts and projects, for generating proof obligations corresponding to a given property, for proving proof obligations in an automatic or/and interactive process and for animating models.

3 Implementing EVENT B models

Our integrated development framework for implementing abstract EVENT B models brings together the strengths of the refinement based approaches and verification based approaches to software development. In particular, our framework supports:

1. Splitting the abstract specification to be solved into its component specifications.
2. Refining these specifications into a concrete model using EVENT B and the RODIN platform.
3. Transforming the concrete model into recursive and iterative algorithms that can be directly implemented as real source code.
4. Verifying the iterative algorithm in the automatic program verification environment of Spec#.

In this paper we focus on the transformations involved in item number three above. First we provide an overview of our integrated development framework to help set the context of our work.

3.1 An overview of our integrated development framework

Figure 2 provides an overview of our framework for refinement based program verification. The problem to be solved is stated as a collection of method contracts, in the form of a Spec# program. Spec# is a formal language for API contracts (influenced by JML, AsmL, and Eiffel), which extends C# through a rich assertion language that allows the specification of objects through class invariants, field annotations, and method specifications [8, 2]. Method preconditions, annotated with the keyword *requires*, express the constraints under which the method will execute correctly. Method postconditions, annotated with the keyword *ensures*, express what should happen as a result of the methods proper execution. The post-condition of methods may refer to the return value of a method using the keyword *result*. The type of the value stored in result must be a subtype of the method's return type. Note also that variables in post-conditions can be prefixed with the keyword *old* e.g., $x = old(x) + 1$ indicates that the new value of x is the old value incremented by 1.

Spec# comes with a sound programming methodology that permits the extended static verification of specifications and their implementations. This process is represented by the arrow labelled *checking* in Figure 2. Dynamic analysis allows the compiler to emit run-time checks at compile time, recording the assertions in the specification as meta-data for consumption by downstream tools. This allows the analysis of program correctness before allowing the program to be run. Internally, it uses an automatic SMT solver (such as Simplify [6] or Z3 [5]) that analyses the verification conditions to prove the correctness of the program or find errors in it.

Note that in the traditional verification approach, the programmer provides both the specification and its implementation. In our integrated development framework we use model refinement in Event B to construct the Spec# implementation from its specification. This refinement also generates the proof obligations that must be discharged as part of the verification. We add these as invariants and assertions in the program so that its verification is completely automatic with the Spec# programming system. The result is a program, from which we can obtain a *cross-proof*, which verifies that the refinement process generates a program, which correctly implements its contract.

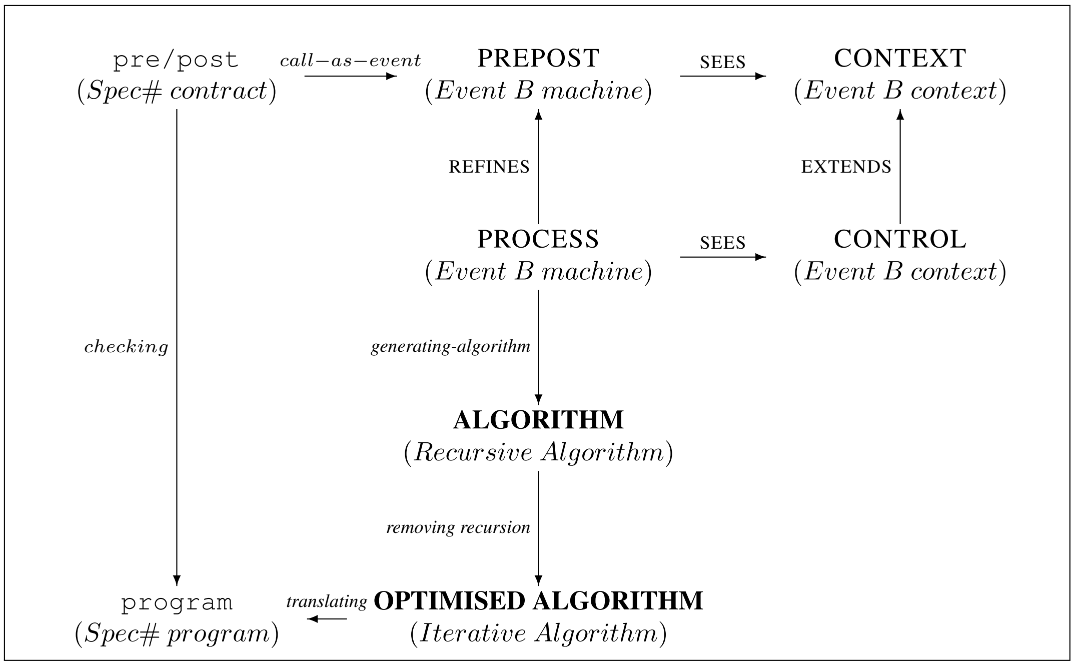

Figure 2: The specification and implementation of an algorithm containing a loop.

The refinement square (with nodes PREPOST, CONTEXT, PROCESS and CONTROL) in Figure 2, provides the mechanism for deriving annotations via refinement. It can be explained briefly as follows:

- The EVENT B machine PREPOST contains events, which have the same contract as that expressed in the original pre/post contract. This machine SEES the EVENT B CONTEXT, which expresses static information about the machine.
- The EVENT B machine PROCESS refines PREPOST generating a concrete specification that satisfies the contract. This machine SEES the EVENT B context CONTROL, which adds control information for the new machine.

- The labelled actions REFINES, SEES and EXTENDS, are supported by the RODIN platform and are checked *completely* using the proof assistant provided by RODIN.

The result of the refinement is the EVENT B machine PROCESS, which contains the refined events and the proof obligations that must be discharged in order to prove that the refinement is correct. The transformation of this EVENT B machine PROCESS into a concrete iterative OPTIMISED ALGORITHM is achieved via two transformations which we present in the sections that follow:

1. Transformation of an EVENT B machine into a concrete recursive algorithm (represented by the arrow labelled *generating-algorithm*).

2. Transformation of this recursive algorithm into its equivalent partially annotated and iterative algorithm (represented by the arrow labelled *removing recursion*).

4 Generating a recursive algorithm from the EVENT B machine

As seen in Figure 2 the result of the refinement is a concrete machine which contains events and their associated proof obligations. Our approach to generating the recursive algorithm depends on the format of these events and is determined by using the pre/post contract of the calling procedure. The event's format may be deterministic, may contain a recursive call of the *procedure under development*, or may contain a call to another procedure which may (or may not) be already developed. The translation of each event into a computable structure is based on a systematic transformation using control labels. Each event is characterised by a current label and a next label. These control labels are added as *annotations* in the event, their purpose being to *simulate* the different steps of the computation. The computation of the recursive algorithm is described by the acyclic graph of labels describing the set of events used in the computation.

4.1 Generating the machines computation graph

Each event e annotates one link in the computation graph, joining nodes that represent the event's pre and post labels. If e annotates the link $\ell_1 \xrightarrow{e} \ell_2$, then the guard of e contains a predicate $\ell = \ell_1$ and the action of e contains $\ell := \ell_2$. From a label ℓ_1, the set of possible *events* that can be observed is denoted by $\mathcal{E}(\ell_1)$. The set of *target labels*, $\mathcal{L}(\ell_1)$, are those labels that can be directly reached from ℓ_1 by an event of $\mathcal{E}(\ell_1)$. The graph of labels annotated by events, denoted $(\mathcal{L}, \mathcal{E}, \longrightarrow)$, is built in a way such that the label $start$ (representing the initial event) has no incoming labels and the label end has no outgoing labels. A further property of the graph is that, for any label $\ell \in \mathcal{L}$, there is a path from label $start$ to end via label ℓ. Moreover, the graph has no cycles since we use recursive calls. This acyclic nature of the graph leads to a recursive version of the algorithm that implements the specification.

For every label in the graph there exists, by construction, a bottom label. The bottom label satisfies the following property: If there is a path from ℓ_1 to ℓ_2 via each $\ell_3 \in \mathcal{L}(\ell_1)$, and a path that leads directly from ℓ_1 to ℓ_2 then the bottom label ℓ_2 is unique. This bottom label is denoted by $\perp(\ell_1)$.

4.2 Deriving the Recursive Program

The next step is to derive a programming structure from the graph. As the graph is acyclic, we derive a program that initially consists only of if statements as illustrated. If we consider the label ℓ_1 we have the following general pattern:

- The set of labels in $\mathcal{L}(\ell_1)$ is $\{\ell_{31}, \ldots, \ell_{3n}\}$.
- The guard of the event labelling the link from ℓ_1 to ℓ_{3i} is denoted by $\ell = \ell_1 \wedge g_{\ell_1,\ell_{3i}}(x)$ where x is a variable parameter of the guard.
- The sentence $\ell = \ell_1$ is removed in the translation.
- $\mathsf{comp}_{\ell_{3i}}$ denotes the result of the translation from ℓ_{3i}.

```
/ * $\ell = \ell_1$
IF   $\mathsf{act}_{\ell_{31}}$
   / * $\ell = \ell_{31}$
   $\mathsf{comp}_{\ell_{31}}$
ELSIF   $\mathsf{act}_{\ell_{32}}$
   / * $\ell = \ell_{32}$
   $\mathsf{comp}_{\ell_{32}}$
ELSIF   $\mathsf{act}_{\ell_{3i}}$
   / * $\ell = \ell_{3i}$
   $\mathsf{comp}_{\ell_{3i}}$
   ...
ELSE   $\mathsf{act}_{\ell_{3n}}$
   / * $\ell = \ell_{3n}$
   $\mathsf{comp}_{\ell_{3n}}$
FI
/ * $\ell = \perp(\ell_1)$
```

The translation process uses the labelled graph of events for translating events into *programming* structures (hence defining what the statements $\mathsf{act}_{\ell_{3i}}$ are). It achieves this by applying a rule for each label ℓ, in each of the three following scenarios: basic events, recursive calls and non recursive calls. We discuss these three scenarios below.

4.2.1 Case 1:Basic Events

If the event e is a basic event controlling the state of the variable x, guarded by $g_{\ell_1,\ell_2}(x)$ and modified by the assignment $x := f_{\ell_1,\ell_2}$ where f is a function, the event e takes the form below.

```
EVENT e
  WHEN
    $\ell = \ell_1$
    $g_{\ell_1,\ell_2}(x)$
  THEN
    $\ell := \ell_2$
    $x := f_{\ell_1,\ell_2}(x)$
  END1
```

The translation omits the control variable ℓ, introduces an IF statement using the guard as the condition and translates the assignment to the target programming language as long as the function f_{ℓ_1,ℓ_2} is implementable. If the event e labels the link $\ell_1 \xrightarrow{e} \ell_2$ then the statement act_{ℓ_2} is defined as WHEN $g_{\ell_1,\ell_2}(x)$ THEN $x := f_{\ell_1,\ell_2}(x)$.
The function f_{ℓ_1,ℓ_2} must be deterministic and translated by an expression definable in some programming language. The EVENT B models are designed to satisfy this hypothesis. If the writer of the models can not remove the non-determinism, the event falls into the two possible next categories.

4.2.2 Case 2: Recursive Call of the Procedure

```
EVENT rec%PROC(h(x),y)%P(y)
ANY $y$
WHEN
  $\ell = \ell_1$
  $g_{\ell_1,\ell_2}(x, y)$
THEN
  $\ell := \ell_2$
  $x := f_{\ell_1,\ell_2}(x, y)$
END1
```

The definition of the event e is not executable and the translation is driven by instances of the control variable ℓ in the guard (as $\ell = \ell_1$) and in the assignment ($\ell := \ell_2$). The statement act_{ℓ_2} is therefore defined as: $\mathsf{PROC}(h(x), y)$. The choice of the event name is the responsibility of the writer of the EVENT B models, who must identify the case corresponding to a recursive call. RODIN authorizes any string and we choose to indicate as much as possible the category of the event (using the keyword *rec*) to facilitate the translation into the programming language. The name is meaningful and annotates the EVENT B models.

Note that it is possible that other occurrences of $rec\%\mathsf{PROC}(h(x), y)\%$ start from the same label and lead to the same post-label. For instance, if the postcondition $P(y)$ holds in one possible event, then another event, with the same pre-label and post-label, may occur with $\neg P(y)$. In this case, the two events are translated into one call.

4.2.3 Case 3: Non Recursive Call

In the third and final case, the event e can be transformed into a call of another procedure.

```
EVENT call%APROC(h(x),y)%P(y)
ANY $y$
WHEN
   $\ell = \ell_1$
   $g_{\ell_1,\ell_2}(x,y)$
THEN
   $\ell := \ell_2$
   $x := f_{\ell_1,\ell_2}(x,y)$
END1
```

The call is expressed by an event e, which we name $call\%\mathsf{APROC}(h(x),y)\%P(y)$ and the statement act_{ℓ_2} is defined as $\mathsf{APROC}(h(x),y)$.
However, the procedure APROC should already be defined (or at least specified) by an EVENT B machine PREPOST. We consider that there is a tree-like structure of sub-procedures under development and we develop the identified procedure APROC in the same way. This last case provides a way to define a hierarchical structure of procedures, which are developed using the same methodology.

In summary, the annotated, and possibly recursive algorithm ALGORITHM is derived from the PROCESS machine by a systematic transformation using the control labels to *simulate* the different steps of the computation. The next step is the transformation of ALGORITHM, into a partially annotated and non recursive OPTIMISED ALGORITHM. This transformation will be presented in the section that follows.

5 Transforming the recursive algorithm into an iterative one

A recursive procedure named APROC can be transformed into a non-recursive procedure named BPROC via a transformation $\mathcal{T}$ as follows $\mathcal{T}(\mathsf{APROC}) = \mathsf{BPROC}$. The source and target of the transformation are stated below in Figure 3. The transformation $\mathcal{T}$ produces a new procedure without *recursive calls* and preserves the partial correctness with respect to the pre and post specification. It must preserve the *operational* semantics of the algorithms. This is, if $[\![A]\!]$ is the function denoting the procedure A then $[\![\mathcal{T}(A)]\!] = [\![A]\!]$. The permitted states are expressed as the set $\Sigma = V \to \mathsf{VALUES}$, where V is the set of variables of the procedure and VALUES are their values.

Theorem 1. *The transformation is sound with respect to the pre and post specification.*

We consider the macro-expansion corresponding to the call $APROC(x,y)$ leading to the set of variables as $V = x \cup z \cup y$ where the initial values of x, y, and z are x_0, y_0, and z_0. We prove that $[\![APROC]\!] = [\![BPROC]\!]$ by considering two cases.

CASE 1 No iteration: Consider that $C(x_0)$ is true. Then $[\![BPROC]\!](x_0, y_0, z_0) = g(x_0)$, since the WHILE loop is not possible and the post processing leads to $y = g(x_0)$.

CASE 2 Iteration: Consider that a sequence of values of x, namely $x_0 \dots x_n$, lead to a value such that either $C(x_n)$ is true or $D(x_n)$ is true, causing the loop to stop iterating. The relation between these values are defined by two sub-cases as follows:

Sub-case 2.1: The sequence is terminated by $C(x_n)$ and no value of (x_i, z_i) satisfies $D(x_i, z_i)$. Hence the following properties hold:

1. $\forall i \in 0..n-1. z_{i+1} = h(x_i, z_i)$
2. $\forall i \in 0..n-1. x_{i+1} = f_{k_i}(x_i)$ with $k_i = 1$ if $E(x_{i-1}, z_{i-1})$

```
PROCEDURE APROC($x$; VAR $y$)
PRECONDITION $P(x)$
POSTCONDITION $Q(x,y)$
BEGIN
LOCAL VARIABLES $z$
IF  $C(x)$  THEN
  $y := g(x);$
ELSE
  $z := h(x,z);$
  IF  $D(x,z)$  THEN
    $y := f(x,z)$
  ELSEIF  $E(x,z)$  THEN
    $APROC(f_1(x),y)$
  ELSE
    $APROC(f_2(x),y)$
  ENDIF
END
```

```
PROCEDURE BPROC($x$; VAR $y$)
PRECONDITION $P(x)$
POSTCONDITION $Q(x,y)$
BEGIN
LOCAL VARIABLES  $z$
WHILE  $not\ C(x) \wedge not\ D(x,z)$  DO
  $z := h(x,z);$
  IF  $E(x,z)$  THEN
    $x := f_1(x);$
  ELSE
    $x := f_2(x);$
  ENDIF
ENDDO
IF  $C(x)$  THEN
  $y := g(x);$
ELSEIF  $D(x,z)$  THEN
  $y := f(x,z);$
ELSEIF  $E(x,z)$  THEN
  $y := f_1(x);$
ELSE
  $y := f_2(x);$
  ENDIF
END
```

Figure 3: Transformation

3. $\forall i \in 0..n-1.x_{i+1} = f_{k_i}(x_i)$ with $k_i = 2$ if $\neg E(x_{i-1}, z_{i-1})$

4. $\forall i \in 0..n.\neg D(x_i, z_i)$

The result y is therefore set by the statement $y = g(x_n)$. Hence $[\![APROC]\!](x_0) = g(x_n)$.

Next, we consider the procedure BPROC executed under the same conditions: We build the same sequence of values for x and z, which ensures that the loop terminates when $C(x_n) \wedge \neg D(x_n, z_n)$. Since the value x_n satisfies $C(x_n)$, the next statement executed is $y := g(x)$ giving y the final value $g(x_n)$. Therefore, $[\![APROC]\!](x_0) = [\![BPROC]\!](x_0)$.

Sub-case 2.2: The sequence is terminated by $D(x_n, z_n)$ and no value of x_i satisfies $C(x_i)$. Hence the following properties hold:

1. $\forall i \in 0..n-1.z_{i+1} = h(x_i, z_i)$

2. $\forall i \in 0..n-1.x_{i+1} = f_{k_i}(x_i)$ with $k_i = 1$ if $E(x_{i-1}, z_{i-1})$

3. $\forall i \in 0..n-1.x_{i+1} = f_{k_i}(x_i)$ with $k_i = 2$ if $\neg E(x_{i-1}, z_{i-1})$

4. $\forall i \in 0..n.\neg C(x_i)$

5. $\forall i \in 0..n-1.\neg D(x_i, z_i)$

The value x_n satisfies $D(x_n, z_n)$ and the result y is therefore set by the statement $y = f(x_n, z_{n+1})$. Hence, $[\![APROC]\!](x_0) = f(x_n, z_{n+1})$. Similar reasoning on BPROC leads to termination of the loop when $D(x_n, z_n)$ and $\neg C(x_n)$. The next statement executed is the assignment $y = f(x_n, z_{n+1})$. Hence, $[\![APROC]\!](x_0) = [\![BPROC]\!](x_0)$.

By the reasoning applied to both cases above, the overall transformation of APROC into BPROC is sound. We illustrate our approach by using a transformation for removing recursion in a given case. In the next section, we illustrate the approach on the binary search problem.

6 Case Study: Binary Search Problem

The binary search problem is a classic algorithmic problem. We reformulate the development of a solution and illustrate the use of the transformation rules for removing recursive calls from the algorithm generated from the EVENT B machine.

6.1 Specifying the binary search problem

The input parameters of the *binsearch* procedure are: a sorted array t; the bounds of the array within which the algorithm should search (lo and hi); and the value for which the algorithm should search (val). Output parameters are $result$ and a boolean flag ok that indicates if $t(result) = val$. The procedure pre and post conditions are presented below in Algorithm 1.

Algorithm 1: $binsearch(t, val, lo, hi, ok, result)$

$$\textbf{precondition} : \left(\begin{array}{l} t \in 0..t.Length \longrightarrow \mathbb{N} \\ \forall k.k \in lo..hi - 1 \Rightarrow t(k) \leq t(k+1) \\ val \in \mathbb{N} \\ l, h \in 0..t.Length \\ lo \leq hi \end{array} \right)$$

$$\textbf{postcondition} : \left(\begin{array}{l} ok = TRUE \Rightarrow \ t(result) = val \\ ok = FALSE \Rightarrow (\forall i.i \in lo..hi \Rightarrow t(i) \neq val \end{array} \right)$$

The array t is sorted with respect to the ordering over integers and a simple inductive analysis is applied leading to a binary search strategy. The specification is first expressed by two events corresponding to the two possible cases: either a key exists in the array t containing the value val, or there is no such key. These two events correspond to the two possible resulting *calls* to the procedure $binsearch(t, val, lo, hi; ok, result)$:

- EVENT find is $binsearch(t, val, lo, hi; ok, result)$ with $ok = TRUE$
- EVENT fail is $binsearch(t, val, lo, hi; ok, result)$: with $ok = FALSE$

```
EVENT find
  ANY   $j$
  WHERE
    $grd1 : j \in lo\,..\,hi$
    $grd2 : t(j) = val$
  THEN
    $act1 : ok := TRUE$
    $act2 : i := j$
  END
```

```
EVENT fail
  WHEN
    $grd1 : \forall k \cdot k \in lo\,..\,hi \Rightarrow t(k) \neq val$
  THEN
    $act1 : ok := FALSE$
  END
```

These two events form the machine called *binsearch1* (which corresponds to the PREPOST machine of Figure 2). This machine is refined to obtain *binsearch2* (which corresponds to PROCESS of Figure 2). This refined machine contains a new control variable, l, which *simulates* how the binary search is achieved.

6.2 Refinement for Computation

The two events EVENT find and EVENT fail are refined according to the following diagram. Note that computations are controlled by the new control variable l, which takes on the values $start$, $middle$ and end to define the possible computation paths of the algorithm. We consider eight possible scenarios within this refinement diagram:

1. $$\begin{pmatrix} l = start \\ lo = hi \\ t(lo) = val \end{pmatrix} \xrightarrow{m_1} \begin{pmatrix} l = end \\ lo = hi \\ ok = TRUE \ \wedge \ result = lo \end{pmatrix}$$
2. $$\begin{pmatrix} l = start \\ lo = hi \\ t(lo) \neq val \end{pmatrix} \xrightarrow{m_2} \begin{pmatrix} l = end \\ lo = hi \\ ok = FALSE \end{pmatrix}$$
3. $$\begin{pmatrix} l = start \\ lo < hi \end{pmatrix} \xrightarrow{split} \begin{pmatrix} l = middle \\ lo < hi \\ mi = (lo + hi)/2 \end{pmatrix}$$
4. $$\begin{pmatrix} l = middle \\ lo < hi \\ mi = (lo + hi)/2 \\ val < t(mi) \end{pmatrix} \xrightarrow{rec(lo,mi-1,val,ok,result)} \begin{pmatrix} l = end \\ ok = TRUE \wedge t(result) = val \end{pmatrix}$$
5. $$\begin{pmatrix} l = middle \\ lo < hi \\ val < t(mi) \end{pmatrix} \xrightarrow{rec(lo,mi-1,val,ok,result)} \begin{pmatrix} l = end \\ \wedge \ ok = FALSE \wedge (\forall i.i \in lo..hi \Rightarrow t(i) \neq val \end{pmatrix}$$
6. $$\begin{pmatrix} l = middle \\ lo < hi \\ mi = (lo + hi)/2 \\ val = t(mi) \end{pmatrix} \xrightarrow{m_3} \begin{pmatrix} l = end \\ ok = TRUE \ \wedge \ result = mi \end{pmatrix}$$
7. $$\begin{pmatrix} l = middle \\ lo < hi \\ mi = (lo + hi)/2 \\ val > t(mi) \end{pmatrix} \xrightarrow{rec(mi+1,hi,val,ok,result)} \begin{pmatrix} l = end \\ ok = TRUE \wedge \ t(result) = val \end{pmatrix}$$
8. $$\begin{pmatrix} l = middle \\ lo < hi \\ mi = (lo + hi)/2 \\ val > t(mi) \end{pmatrix} \xrightarrow{rec(mi+1,hi,val,ok,result)} \begin{pmatrix} l = end \\ \wedge \ ok = FALSE \wedge (\forall i.i \in lo..hi \Rightarrow t(i) \neq val \end{pmatrix}$$

Each of these scenarios are used to generate the refined events in the concrete machine *binsearch2* with event names corresponding to the labels on the arrows in each scenario.

6.3 Generating the algorithm from events

The events of the machine called *binsearch2* are listed below. Note that events $m1$, $m2$, $split$ and $m3$ correspond directly with scenarios 1, 2, 3 and 6.

```
EVENT m1   REFINES find
  WHEN
    $grd1 : l = start$
    $grd2 : lo = hi$
    $grd3 : t(lo) = val$
  WITNESSES
    $j : j = lo$
  THEN
    $act1 : l := end$
    $act2 : ok := TRUE$
    $act3 : i := lo$
  END
```

```
EVENT m3   REFINES find
  WHEN
    $grd1 : l = middle$
    $grd3 : t(mi) = val$
  WITNESSES
    $j : j = mi$
  THEN
    $act1 : l := end$
    $act2 : ok := TRUE$
    $act3 : i := mi$
  END
```

```
EVENT m2   REFINES fail
  WHEN
    $grd1 : l = start$
    $grd2 : lo = hi$
    $grd3 : t(lo) \neq val$
  THEN
    $act1 : l := end$
    $act2 : ok := FALSE$
  END
```

```
EVENT split
  WHEN
    $grd1 : l = start$
    $grd2 : lo < hi$
  THEN
    $act1 : l := middle$
    $act2 : mi := (lo + hi)/2$
  END
```

Scenarios 4 and 5 correspond to the case where $val < t(mi)$ and hence the search continues on the left part of the array. Two events will be generated for this case: one where the value is found ($OK = true$), and one where the value is not found ($OK = false$). Similarly scenarios 7 and 8 correspond to the case where $val > t(mi)$ and the search continues on the right part of the array. Again, two events will be generated for this case: one where the value is found ($OK = true$), and one where the value is not found ($OK = false$). We illustrate two of the four events below.

```
EVENT rightsearchOK   REFINES find
  ANY  $j$
  WHERE
    $grd1 : l = middle$
    $grd2 : val > t(mi)$
    $grd3 : j \in mi + 1 \,..\, hi$
    $grd4 : t(j) = val$
    $grd5 : mi + 1 \leq hi$
  THEN
    $act1 : i := j$
    $act2 : ok := TRUE$
  END
```

```
EVENT rightsearchKO   REFINES fail
  WHEN
    $grd1 : l = middle$
    $grd2 : val > t(mi)$
    $grd4 : \forall j \cdot j \in mi + 1 \,..\, hi \Rightarrow t(j) \neq val$
    $grd5 : mi + 1 \leq hi$
  THEN
    $act2 : ok := FALSE$
  END
```

We identify that the translation from EVENT B into an algorithmic notation introduces new proof obligations. These proof obligations state that the call is correct [15] i.e. the current state implies that the precondition is true. In the case of our example, we prove that each guard of rightsearchOK and rightsearchKO implies the precondition of the algorithm: the theorems labelled th_{call1} and th_{call2}, in the invariant below, express the discharged conditions.

Using the control variable l, we can apply our **generating-algorithm** transformation to produce the recursive algorithm. The result is Algorithm 2 below which is derived from the refined EVENT B model *binsearch2*. The control variable l is removed by introducing control via if statements.

Algorithm 2: Recursive Algorithm binsearch(t,val,lo,hi,ok,result).

```
**precondition** : $\begin{pmatrix} n \in \mathbb{N}1 \land lo, hi \in dom(t) \land lo \leq hi \\ t \in 0 \,..\, n-1 \rightarrow \mathbb{N} \land \forall i.i \in 0..n-2 \Rightarrow t(i) \leq t(i+1) \end{pmatrix}$
**postcondition** : $\begin{pmatrix} ok = true \Rightarrow t(result) = val \\ ok = false \Rightarrow (\forall i.i \in lo..hi \Rightarrow t(i) \neq val \end{pmatrix}$
**local variables**: $mi \in \mathbb{Z}$
$start : \begin{pmatrix} n \in \mathbb{N}1 \land lo, hi \in dom(t) \land lo \leq hi \\ t \in 0 \,..\, n-1 \rightarrow \mathbb{N} \land \forall i.i \in 0..n-2 \Rightarrow t(i) \leq t(i+1) \end{pmatrix}$
**if** $lo = hi \land t(lo) = val$ **then**
    $result := lo; ok := true;$
**else**
    **if** $lo = hi \land t(lo) \neq val$ **then**
        $ok := false;$
    **else**
        $mi := (lo + hi)/2;$
        $middle : \left\{ \begin{pmatrix} mi = (lo + hi)/2 \\ val < t(mi) \Rightarrow \forall k.k \in mi..hi \Rightarrow t(k) \neq val \\ val > t(mi) \Rightarrow \forall k.k \in lo..mi \Rightarrow t(k) \neq val \end{pmatrix} \right\}$
        **if** $t(mi) = val$ **then**
            $result := mi; ok := true;$
        **else**
            **if** $mi + 1 \leq hi \land t(mi) < val$ **then**
                binsearch(t,val,mi+1,hi,ok,result);
            **else**
                **if** $lo \leq mi - 1 \land t(mi) < val$ **then**
                    binsearch(t,val,lo,mi-1,ok,result);
                **else**
                    ok:=FALSE
;
$end : \left\{ \begin{pmatrix} ok = true \Rightarrow t(result) = val \\ ok = false \Rightarrow (\forall i.i \in lo..hi \Rightarrow t(i) \neq val \end{pmatrix} \right\};$
```

Finally, using RODIN we prove that the following assertion is an invariant for our model and hence it can be used for inferring the invariant of the generated code.

$$\begin{array}{l}
inv1 : i \in 1 \mathinner{..} n \\
inv2 : l \in LOC \\
inv3 : dom(t) = 1 \mathinner{..} n \\
inv4 : mi \in 1 \mathinner{..} n \\
inv5 : l = middle \Rightarrow lo < hi \land mi \in lo \mathinner{..} hi \\
inv6 : l = middle \land val < t(mi) \Rightarrow (\forall k \cdot k \in mi \mathinner{..} hi \Rightarrow t(k) \neq val) \\
inv7 : l = middle \land val > t(mi) \Rightarrow (\forall k \cdot k \in lo \mathinner{..} mi \Rightarrow t(k) \neq val) \\
inv8 : l = end \land ok = TRUE \Rightarrow i \in lo \mathinner{..} hi \land t(i) = val \\
inv9 : l = end \land ok = FALSE \Rightarrow (\forall k \cdot k \in lo \mathinner{..} hi \Rightarrow t(k) \neq val) \\
inv10 : lo \mathinner{..} hi \subseteq 1 \mathinner{..} n \\
th_{call1} : (\exists j \cdot l = middle \land j \in mi+1 \mathinner{..} hi \land key > t(mi) \land t(j) = key \land mi+1 \leq hi) \\
\qquad\qquad \Rightarrow \quad mi + 1 \leq hi \\
th_{call2} : (\exists j \cdot l = middle \land j \in lo \mathinner{..} mi-1 \land key < t(mi) \land t(j) = key \land lo \leq mi-1) \\
\qquad\qquad \Rightarrow \quad lo \leq mi - 1
\end{array}$$

6.4 Transforming the Binary Search Recursive Procedure

In order to generate the iterative version of our algorithm, we apply the **removing recursion** transformation. We identify the rules condition C as $\left(\begin{array}{l} lo = hi \land t(lo) = val \\ \lor\ lo = hi \land t(lo) \neq val \\ \lor\ lo < hi \land mi = (lo+hi)/2 \land t(mi) = val \end{array}\right)$ and obtain PROCEDURE $binsearch(t, val, lo, hi, ok, result)$ as presented below in Figure 4.

6.5 Interpreting the algorithms within Spec#

In order to fully utilize our integrated development framework for refinement based program verification, we have translated the resulting iterative algorithm into Spec#. As shown in Figure 5 this is almost a one-to-one mapping. The main difference in the algorithms is that we simply return a value of -1 when our iterative algorithm sets OK to $false$ and return the index where the value is found when our iterative algorithm sets OK to $true$. The algorithm verified as correct, in less than 2 seconds using the Spec# programming system (version 2011-10-03). No user interaction is required in the verification as all assertions required (preconditions, postconditions and loop invariants) have been generated as part of the refinement and transformation of the initial abstract specification into the final iterative algorithm. It is interesting to note that, prior to formalising our transformation rules, our initial attempt at writing this iterative C# program contained an error. This error in the loop body, was due to our omission to check that the values of $mi + 1$ and $mi - 1$ were within the array bounds before narrowing the search space. This error was immediately detected by the Spec# programming system. The automatic verification of the final program is available online at `http://www.rise4fun.com/SpecSharp/psP4`.

This verification step acts as an insurance check for the *correct-by-construction* approach in two ways. Firstly, while the Event B framework provides for the automatic verification of some proof obligations, many proofs require the user to manually interact with the tools to provide guidance.This often leads to error, typically introduced by incorrect assumptions made by the user while proving a proof obligation. Having an alternative verification tool that automatically verifies that the final implementation is correct with respect to its specification re-assures the developer that their interactions were correct at each stage of the development. Secondly, Event B is a modelling language where data types, event guards and actions are *logical* structures based on set theory. While the *programming* structures of Spec# have logical features, they also have programming constraints that must be taken into account

```
PROCEDURE $binsearch(t, val, lo, hi, ok, result)$
PRECONDITION $\begin{pmatrix} t \in 0..t.Length \longrightarrow \mathbb{N} \\ \forall k.k \in lo..hi-1 \Rightarrow t(k) \leq t(k+1) \\ val \in \mathbb{N} \wedge lo, hi \in 0..t.Length \wedge lo \leq hi \end{pmatrix}$
POSTCONDITION $\begin{pmatrix} ok = TRUE \Rightarrow \ t(result) = val \\ ok = FALSE \Rightarrow (\forall i.i \in lo..hi \Rightarrow t(i) \neq val \end{pmatrix}$
BEGIN
WHILE  $not \begin{pmatrix} lo = hi \wedge t(lo) = val \\ \vee\ lo = hi \wedge t(lo) \neq val \\ \vee\ lo < hi \wedge mi = (lo+hi)/2 \wedge t(mi) = val \end{pmatrix}$  DO
  $mi := (lo + hi)/2;$
$middle : \left\{ \begin{pmatrix} mi = (lo+hi)/2 \\ val < t(mi) \Rightarrow \forall k.k \in mi..hi \Rightarrow t(k) \neq val \\ val > t(mi) \Rightarrow \forall k.k \in lo..mi \Rightarrow t(k) \neq val \end{pmatrix} \right\}$
  IF  $mi + 1 \leq hi \wedge val > t(mi)$  THEN
    $lo := mi + 1$
  ELSEIF  $lo \leq mi - 1 \wedge val < t(mi)$  THEN
    $hi := mi - 1$
ENDDO
  IF  $lo = hi \wedge t(lo) = val$  THEN
    $result := lo; ok := true$
  ELSEIF  $lo = hi \wedge t(lo) \neq val$  THEN
    $ok := false$
  ELSEIF  $lo < hi \wedge t(mi) = val$  THEN
    $result := mi; ok := true$
  ELSE  $ok := false$
ENDIF
END
```

Figure 4: PROCEDURE $binsearch(t, val, lo, hi, ok, result)$

when translation from the resulting iterative algorithm to a programming langauge. The *cross* verification increases trust in the final product ensuring that the semantics of the original specification is maintained.

7 Related Work

The topic of program transformation, and in particular, the transformation of recursive programs to iterative ones is not new. In 1965, Gordon [16] investigated the tranformation of *recursive relations to recurrence or iterative relations*. He also addressed the transformation of non primitive recursive functions (namely Ackerman's function) into iterative functions, opening a new domain of research on transformations and promoting the use of recursive definitions of algorithms. Later Strong [9] specified the problem of transforming *recursive equations* into iterative equations expressed by flowcharts while Pettorossi [13], aimed to improve a program's memory usage. Research by Darlington and Burstall [4] proposed a list of transformations, which can be automatically applied for removing recursive calls from a program.

```
class BS {
 int BinarySearch(int[] t, int val, int lo, int hi, bool ok)
  requires 0 <= lo && lo < t.Length && 0 <= hi && hi < t.Length;
  requires lo <= hi && 0 < t.Length;
  requires forall {int i in (0:t.Length), int j in (i:t.Length); t[i] <= t[j]};
  ensures -1 <= result && result < t.Length;
  ensures (0 <= result && result < t.Length)==> t[result] == val;
  ensures result == -1 ==> forall {int i in (lo..hi); t[i] != val};
 {
      int mi = (lo + hi) / 2;
      while (!(lo == hi && t[lo] == val) || ( lo == hi && t[lo] != val)
              || (lo < hi && (mi == (lo + hi) /2) && t[mi] == val))
       invariant 0 <= lo && lo < t.Length && 0 <= hi && hi < t.Length;
       invariant 0 <= mi && mi < t.Length;
       invariant (val < t[mi]) ==> forall {int i in (mi..hi); t[i] != val};
       invariant (val > t[mi]) ==> forall {int i in (lo..mi); t[i] != val};
      {
          mi = (lo + hi) /2;
          if  ((mi+1 <= hi) && (val > t[mi])) lo = mi +1;
          else if ((lo <= mi-1) && (val < t[mi])) hi = mi - 1;
      }
      if ((lo == hi) && (t[lo] == val)) {ok = true; return lo;}
      else{
          if ((lo == hi) && (t[lo] != val)) {ok = false; return -1;}
          else if ((lo < hi) && (t[mi] == val)) {ok = true; return mi;}
               else {ok = false; return -1;}
     } } }
```

Figure 5: Binary Search C# program corresponding to the generated iterative procedure.

Our work does not claim to discover new transformations. Instead we have extended these transformations within EVENT-B models to integrate a *correctness* phase in the transformation. Our work relates two complementary frameworks: the programming framework of C# and the modelling framework of EVENT-B. We promote the development of annotated programs and present this as the main contribution over previous work. We consider that the recursive algorithms are easier to generate using our approach and they can be easily transformed to get an efficient iterative solution.

In our seminal work on code generation [12] we have developed a shortest path algorithm based on the *dynamic programming* paradigm where the discovery of program invariants utilises the underlying inductive properties of the algorithm. The operational aspect of the iterative solution is useful for improving the quality and efficiency of the resulting code. This is our motivation for transforming our recursive algorithms into iterative ones, obtaining their correctness for free using our integrated development framework, that brings together the world of system modelling and the world of program verification.

8 Conclusion

We have presented and verified the correctness of two transformation rules, which transform EVENT B models into iterative algorithms. The resulting algorithms are correct-by-construction and can be directly mapped into an executable programming language. We provide a cross-proof by verifying the correctness of our final program using the Spec# programming system. Our integrated development framework (Figure 2) indicates where our transformations are used for producing a program that is *correct-by-construction*. The translation of the PROCESS machine into a recursive algorithm is straightforward and removes the control variable used to relate events when generating the code.

This work builds on a method for code generation that is detailed by one of the authors in [11, 12] and provides the foundation for an integrated development framework that brings together the world of system modelling and the world of program verification. The EB2ALL code generation tool [10] can also produce a program from the PROCESS machine. However, the control variable is not removed and the resulting code is not structured. The advantage of our approach is the production of a structured iterative program, which can be automatically verified using the Spec# programming system.

Our experience shows that our approach assists students in developing and understanding the tasks of software specification and verification. It also makes different forms of formal software development more accessible to the Software Engineers, helping them to build correct and reliable software systems. Future work will include the development of adequate plugins, which will integrate and facilitate the co-operation between Spec# tools and RODIN tools.

References

[1] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.

[2] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *LNCS*, pages 49–69. Springer, 2005.

[3] ClearSy, Aix-en-Provence (F). *Atelier B*, 2002. Version 3.6.

[4] J. Darlington and R.M. Burstall. A system which automatically improves programs. *Acta Informatica*, 6(1):41–60, 1976.

[5] Leonardo de Moura and Nikolaj Björner. Z3: An efficient SMT solver. In *Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS*, 2008.

[6] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, May 2005.

[7] Gary T. Leavens et al. Roadmap for enhanced languages and methods to aid verification. In *Fifth Intl. Conf. Generative Programming and Component Engineering (GPCE 2006)*, pages 221–235. ACM, October 2006.

[8] Mike Barnett et al. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005*, volume 4111 of *LNCS*, pages 364–387. Springer, 2006.

[9] H.R. Strong Jr. Translating recursion equations into flow charts. *Journal of Computer and System Sciences*, 5(3):254 – 285, 1971.

[10] D. Méry and N. Singh. eb2all.loria.fr, 2011.

[11] Dominique Méry. A simple refinement-based method for constructing algorithms. *ACM SIGCSE Bulletin*, 41(2):51–59, 2009-06.

[12] Dominique Méry. Refinement-based guidelines for algorithmic systems. *International Journal of Software and Informatics*, 3(2-3):197–239, 2009-09.

[13] Alberto Pettorossi. Improving memory utilization in transforming recursive programs (extended abstract). In Józef Winkowski, editor, *MFCS*, volume 64 of *LNCS*, pages 416–425. Springer, 1978.

[14] Project RODIN. Rigorous open development environment for complex systems. http://rodin-b-sharp.sourceforge.net/, 2004.

[15] John C. Reynolds. *The Craft of Programming*. Prentice-Hall International series in computer science. Prentice-Hall International, 1982.

[16] H. Gordon Rice. Recursion and iteration. *Commun. ACM*, 8(2):114–115, 1965.