On Two Friends for Getting Correct Programs

Automatically Translating Event B Specifications to Recursive Algorithms in Rodin

Zheng Cheng[1], Dominique Méry[2(✉)], and Rosemary Monahan[1]

[1] Computer Science Department, Maynooth University, Co. Kildare, Ireland
zcheng@cs.nuim.ie, rosemary.monahan@nuim.ie
[2] LORIA, Université de Lorraine,
Campus Scientifique, BP 70239, 54506 VandœUvre-lès-nancy, France
dominique.mery@loria.fr

Abstract. We report on our progress-to-date in implementing a software development environment which integrates the efforts of two formal software engineering techniques: program refinement as supported by Event B and program verification as supported by the Spec# programming system. Our objective is to improve the usability of formal verification tools by providing a general framework for integrating these two approaches to software verification. We show how the two approaches Correctness-by-Construction and Post-hoc Verification can be used in a productive way. Here, we focus on the final steps in this process where the final concrete specification is transformed into an executable algorithm. We present EB2RC, a plug-in for the Rodin platform, that reads in an Event B model and uses the control framework introduced during its refinement to generate a graphical representation of the executable algorithm. EB2RC also generates a recursive algorithm that is easily translated into executable code. We illustrate our technique through case studies and their analysis.

1 Introduction

The problem that we address is as follows: Given a program specification how do we provide an integrated software development environment in which we can (a) refine the specification into one that is algorithmic and (b) automatically verify that the derived algorithm meets the specification?[1] Our proposed solution is to combine the efforts of two formal software engineering techniques: program refinement as supported by Event B [1] and program verification as supported by the Spec# Programming System [3]. Our objective is to improve the usability of formal verification tools by providing a general framework for integrating these two approaches to software verification. We focus on the strengths of each so that

[1] We acknowledge the Irish Research Council and Campus France for the joint funding of this research collaboration via the Ulysses scheme 2013. This work was supported by grant ANR-13-INSE-0001 (The IMPEX Project http://impex.gforge.inria.fr) from the Agence Nationale de la Recherche (ANR).

T. Margaria and B. Steffen (Eds.): ISoLA 2016, Part I, LNCS 9952, pp. 821–838, 2016.
DOI: 10.1007/978-3-319-47166-2_57

their integration makes the verification task more approachable for users. The final architecture induces a methodology which is useful for the specification, the construction and the verification of correct sequential algorithms.

Here, we report on progress in implementing our integrated software development environment. The input to our system is an abstract specification which is then refined into a more concrete specification using the Event B modelling language and its associated tool-set, the Rodin platform. The output from our system is a concrete Spec# program containing both the executable code and the proof obligations that are necessary for its automatic verification. Here, we focus on the later steps in this process where the final concrete specification is transformed into an executable algorithm. We present a plug-in for the Rodin platform that reads in an Event B model and uses the control framework introduced during its refinement to generate both a graphical representation of the executable algorithm, and a recursive algorithm that is easily translated into executable code.

In [10], we presented and verified the transformations involved in generating executable code from Event B. We verified the correctness of the transformed executable code in a static program verification environment for C# programs, namely the Spec# programming system.

In this paper, we focus on implementing one of the core transformations, which is the final concrete specification is transformed into an executable recursive algorithm. This has been implemented by the EB2RC, a plug-in for the Rodin platform. We analysis the impact of our tool through several case studies. The analysis leads us to identify and discuss on the strengths of program refinement and post-hoc program verification so that their integration makes the verification task more approachable for users.

Paper organization. We provide a brief overview of program refinement as supported by Event B (Sect. 2). We then give an overview of our framework for refinement based program verification (Sect. 3). The technical details of our translation procedure and its implementation as EB2RC are presented in Sect. 4. The impact of our tool is shown in Sect. 5. An analysis of more case studies that illustrate our technique and our conclusion are then presented in Sects. 6 and 7 respectively.

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* which define the static components of the model and *machines* which define the dynamic components of the model. Event B *machines* are characterized by a finite list x of *state variables*, modified by a finite list of *events*, where an invariant $I(x)$ states properties that must always be satisfied by the variables x and maintained by the events. For an example see events $find$ and $fail$ in Sect. 5 which provides for an initial model of a binary search algorithm.

Each event has three main parts: a list of local parameters, a guard G and a relation R over values denoted *pre*-values (x) and *post*-values (x') of state variables. When the guard holds the actions in the event body modify the state variables according to the relation R. A *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. The most common representation of an event 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 $\exists\, t \cdot\, (G(t, x) \;\wedge\; R(x, x', t))$.

2.1 Verifying Event B Models

The Event B modelling language is supported by the RODIN platform [12]. These both provide facilities for editing 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. Note that our models can only express *safety* and *invariance* properties, which are state properties. Proof obligations produced via the RODIN platform (see Listing 1) include that the initialisation event establishes the invariant I, that the event e preserves the invariant I and that the event e is feasible with respect to the invariant I. By proving feasibility, we prove that $BA(e)(x, y)$ provides an after-state whenever $grd(e)(x)$ holds. This means that the guard indeed represents the enabling condition of the event.

- (INV1) $Init(x) \;\Rightarrow\; I(x)$
- (INV2) $I(x) \;\wedge\; BA(e)(x, x') \;\Rightarrow\; I(x')$
- (FIS) $I(x) \;\wedge\; \mathsf{grd}(e)(x) \;\Rightarrow \exists y. BA(e)(x, y)$
- (GLU) $I(x) \wedge J(x, y) \wedge BA(f)(y, y') \;\Rightarrow\; \exists x' \cdot (BA(e)(x, x') \wedge J(x', y'))$
- (SKIP) $I(x) \wedge J(x, y) \wedge BA(f)(y, y') \Rightarrow\; J(x, y')$

Listing 1: Event B proof obligations

Refinement of an Event B model 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 *glueing invariant* $J(x, y)$ which must be maintained throughout the system modelling. A number of proof obligations generated by each refinement step (see Listing 2) 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. Through refinement we can enrich our Event B models in a step-by-step manner and validate each decision step as we construct the final concrete model. This is the foundation of the correct-by-construction approach [7].

2.2 The Call-as-event Paradigm

The main idea of our methodology is based on the call-as-event paradigm [9]. It expresses the consequences on the correct-by-construction approach. In this section, we give a short summary of the method.

Abrial [1] shows how sequential programs can be developed by the Event B refinement approach. He lists rules for producing sequential programs by merging events. Kourie and Watson [6] illustrates the use of Morgan's refinement calculus for developing sequential programs without any proof assistant support. Both approaches are based on the same idea of developing invariants to prove verification conditions. However, developing invariants is generally not an easy task. The refinement-based development, which involves several steps of refinements, makes this task even more difficult (i.e. to glue/synchronize the developed invariants across refinements).

The call-as-event paradigm initiates the development of a sequential program by stating its specification (i.e. inputs-outputs behaviours through the pre/post-conditions) as abstract events in an abstract model. Then, the subsequent refinements introduce more concrete models, based on an inductive definition of the outputs with respect to the input. Each concrete model contains concrete events that aim to compute the same sequential program under development, but with more detail of the computation.

The essential idea of the call-as-event paradigm is that the concrete event can be expressed in a way to represent a procedure call (by following a straightforward syntactic naming convention for events [10]), which makes refinement proofs easier: (a) the control variables can be introduced over the events for structuring the inductive computation. (b) the invariants can be defined in a simpler way by analysing the specification of calls.

Specifically, the call-as-event paradigm has three types of events to be used in a concrete model:

- Basic events. An event e is a basic event if it represents a sequence of atomic computation steps.
- Recursive call events. An event e is a recursive call event if it corresponds to the call of the procedure under development.
- Non-recursive call events. An event e is a non-recursive call event if it corresponds to the call of another procedure.

The type of the events are distinguished by their event name. The recursive and non-recursive call events are prefixed with rec and call respectively, followed by the sub-procedure to be called. Moreover, to ensure the soundness of the program development, the sub-procedure to be called should have been defined/specified by an Event B machine, since the developed program should not call a miracle procedure (i.e. a procedure that does not exists). This is a incremental development strategy, where the developer can focus on developing the main-procedure, reuse of developed specifications of sub-procedures and stage their development using the same call-as-event paradigm.

In summary, the refinement process that based on the call-as-event paradigm is straightforward, and writing invariant becomes easier for following the inductive property defining the computation to program. In the next section, we show how we interact with the Spec# language.

3 An Overview of Our Integrated Development Framework

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 algorithms that can be directly implemented as real source code using graph visualisation and applying code generation transformations.
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. First we provide an overview of our integrated development framework to help set the context of our work. Figure 1 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 [2,3]. Spec# comes with a sound programming methodology that 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 executed.

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.

The Event B refinement square (with nodes PREPOST, CONTEXT, PROCESS and CONTROL) in Fig. 1, 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 our EB2RC tool (Sect. 4) and removing recursion [10].

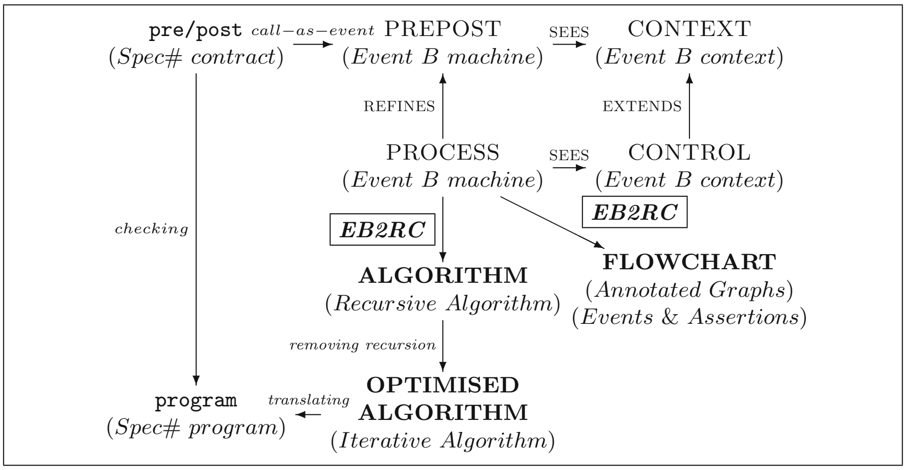

Fig. 1. An overview of our integrated development framework to combine program refinement with program verification

4 EB2RC: A Tool for Translating Event B Models to Recursive Code

We support the generation of a concrete recursive algorithm ALGORITHM from the Event B machine PROCESS with EB2RC, a plug-in for Rodin which we have developed. In the sections that follow we describe the generation process in detail.

4.1 Overview of Our EB2RC Plugin

As seen in Fig. 1 the result of the refinement is a concrete Event B machine PROCESS, which is the input of our EB2RC plugin. Then, our plugin generates a recursive algorithm (ALGORITHM) in text format, and a graphical representation of the recursive algorithm (FLOWCHART) to improve comprehensibility. The textual recursive algorithm can be easily translated into either executable code or artefacts of the post-hoc verification tools (e.g. Spec#).

Our approach for generating a recursive algorithm from a concrete machine is based on a systematic transformation using control labels: each machine has a start and an end label, and each event is characterised by a current label and a next label. The purpose of these control labels is to *simulate* the different computation steps of the developed recursive algorithm. In other words, the computation steps of the recursive algorithm are abstracted by a acyclic graph of control labels, where these labels describe the set of events used in the computation.

Specifically, our plugin first ensures the input machine is ready for recursive algorithm generation by design extra proof obligations (Sect. 4.2). Then, it extracts essential information (e.g. control labels) from the concrete machine (Sect. 4.3). This step is guided by a auxiliary configuration file provided by the user (i.e. the developer of the concrete machine). Next, based on the essential information extracted, our plugin systematically reconstructs a recursive algorithm in textual and graphical representation (Sect. 4.4).

Our EB2RC plugin is written in Java. It interacts with APIs of the Rodin platform (v2.7) to extract information from the concrete machine of interest. Then, after automatic systematic reconstruction, our plugin directly generates textual recursive algorithm, and a input file for the *Dot* tool of *GraphViz*, thereby producing its graphical representation.

4.2 Proof Obligations

A set of extra proof obligations are generated during the *generating-algorithm* stage in our Integrated Development Framework (Fig. 1). They are to ensure that the Event B machine can be safely translated into a recursive algorithm, for example:

- The annotated control labels in the actions and guards of each event are different (i.e. the event always progresses);
- Only one event does not have any control labels in its guards (i.e. the start event);
- Only one event does not have any control labels in its actions (i.e. the end event);
- The labels in an Event B machine forms an acyclic graph;

4.3 Extracting Information from Event B Machine

To guide our plugin to proceed, we require the user to define the following information into a configuration file:

- The name of the input Event B machine (i.e. the concrete Event B machine to be processed).
- The name of the control label used by the input Event B machine.
- The name of the start control label used by the input Event B machine.

Then, based on the configuration file the user provided, our EB2RC plugin extracts essential information from each event of the input machine. One of the essential information our plugin interested in is the *current* and *next* labels of each event. The control labels are used to control the order in which the events are combined to achieve a recursive algorithm. The *current* label informs us of the start state of an event, whereas the *next* label of an event determines which events will follow it. The *current* and *next* control labels are derived from the guards and actions of each event respectively.

Another essential information our plugin recorded is the type of each event. During the refinements, we distinguish three types of events by their names: (a) basic events represent a sequence of atomic computation steps, whose names are without any prefixes. (b) recursive events represent a computation step of a recursive call, whose names are prefixed with *rec.* (c) call events represent a computation step of a external function call, whose names are prefixed with *call.* By categorising events by their types, EB2RC will treat them differently while extracting information.

Our plugin also records the guards and actions of each event. To facilitate textual and graphical recursive algorithm generation, we perform some optimizations: First, for events of basic type, all the guards and deterministic actions[2] are recorded unless they reference control labels. Second, for events of recursive and call type, the guards and actions are derived from their event name. This becomes practical because of the naming convention of our approach [10], i.e. we require the guards and actions need to be explicitly referred by the name of the events of recursive and call type.

Finally, our plugin needs to store the next events of an target event. An event x is regarded as the next event of a target event y if the current label of x equals to the next label of y. In this manner, each event can be related to other events as in a transition system.

4.4 Representing Extracted Information

An intuitive diagram allows easier understanding of the algorithm, and is a prerequisite for modularizing complex algorithms. Therefore, we construct a control flow graph for the input Event B machine by using extracted information.

We start by consulting the configuration file for the start control label used by the input Event B machine. Then, we find an event of the input with this start label as its current label, printing its actions and guards (according to the grammar of the *Dot* language), and recursively apply the same printing procedure to its next events.

[2] In Event B, two types of actions (*becomes_such_that* and *becomes_in_set* actions) are non-deterministic.

Representing the developed Event B machine in textual format is similar to the generation of graphical format, only differs in how it is printed. We chose a general syntax that can be understand by the programmers to print, and it is straightforward to customize the printing to generate artefacts for the post-hoc verification tools (e.g. Spec#). Examples of generating textual and graphical representation for a recursive binary search algorithm is given in Sect. 5. More applications can be seen in Sect. 6.

5 Case Study: The Binary Search Problem

We detail the complete development by refinement of an algorithm which solves the problem of *searching for a value in a table.* We demonstrate concrete example of the output of our EB2RC plugin, which is a recursive algorithm and its graphical representation. Then, we discuss our experience of integrating two popular approaches to formal software development, i.e. refinement and post-hoc verification approaches.

5.1 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 procedures pre and post conditions are presented in Algorithm 1.

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

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

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

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 (Listing 3): 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 $binsearch(t, val, lo, hi; ok, result)$ where $ok = TRUE$
- EVENT fail is $binsearch(t, val, lo, hi; ok, result)$ where $ok = FALSE$

```
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
```

Listing 2: The specification of the binary search algorithm in Event B

These two events form the machine called *binsearch1* which is refined to obtain *binsearch2* (corresponding to PROCESS of Fig. 1). In addition to these events, the events of this refined machine contains a new control label, l, which *simulates* how the binary search is achieved. We illustrate two of the events of *binsearch2* in Listing 3. Its complete refinement is presented in [10].

```
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
```

Listing 3: Refining the specification of the binary search algorithm (shown in Listing 3) in Event B

5.2 Automatic Generation of the Algorithm

The result of our translation is two-fold. Firstly, to help people comprehend the algorithm, EB2RC reads in the Event B machine and visualizes it as in Fig. 2. Specifically, we draw a circular node to present each event. The guards of each event are indicated by the arrows, and the actions of the event are indicated in the text of the rectangular node belonging to each arrow. The outcome of each event is transitions to other events, which is indicated by directed edges between two circular nodes.

Secondly, a textual representation of the binary search algorithm is constructed by the EB2RC. The produced algorithm (as shown in Algorithm 2) has been compared to the algorithm produced by hand by the authors. The two algorithms are identical up to a slight reformatting.

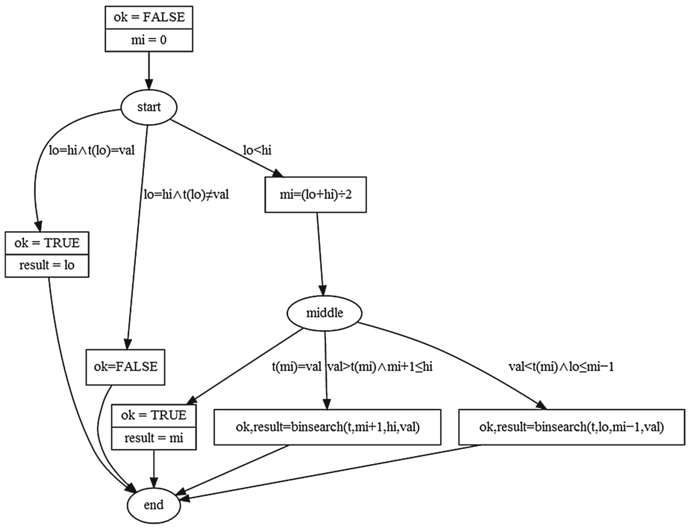

Fig. 2. Visualized representation of the binary search algorithm

Table 1. Proof effort of our refinement approach for the binary search case study

Model	Total	Auto	Manual	Reviewed	Undischarged	% auto
binsearch1	5	5	0	0	0	100 %
binsearch2	71	63	8	0	0	78 %

The proof effort of our refinement approach for the Binary Search case study is illustrated in Table 1. The first abstract model is proved automatically and the second concrete model is automatic in 78 % of its proof obligations.

5.3 Discussion

Upon this point, we have shown that our technique assists in discovering a *good* inductive process which will lead to a recursive solution. However, we do not take dynamic properties (e.g. range of array index) of general programming languages into consideration while applying our technique. As a result, it is possible to produce unreliable executable code from the recursive algorithm we generated. To our knowledge, checking dynamic properties of general programming languages is not currently supported by the Event B approach. It would be cumbersome to mimic this feature in Event B for every developed algorithm. Whereas, Spec# checks several dynamic properties of the C# language by default.

The essential idea of our integrated development framework is to bring together the strengths of the refinement based approaches and post-hoc verification based approaches to software development. This kind of interoperability between approaches allows several techniques to interact with each other

Algorithm 2. Recursive Algorithm binsearch(t,lo,hi,val;ok,result) generated by EB2RC

```
$ok := FALSE; mi := 0;$
**if** $lo = hi \wedge t(lo) = val$ **then**
    $ok := TRUE; result := lo;$
**else if** $lo = hi \wedge t(lo) \neq val$ **then**
    $ok := FALSE;$
**else if** $lo < hi$ **then**
    $mi := (lo + hi) \div 2;$
    **if** $t(mi) = val$ **then**
        $ok := TRUE; result := mi;$
    **else if** $val > t(mi) \wedge mi + 1 \leq hi$ **then**
        $ok, result := binsearch(t, mi + 1, hi, val);$
    **else if** $val < t(mi) \wedge lo \leq mi - 1$ **then**
        $ok, result := binsearch(t, lo, mi - 1, val);$
```

(e.g. share information, chain the proof obligations) so that they can collectively prove tasks/theorems more automatically than any one of them could prove in isolation.

Kaufmann and Moore, based on their experience, suggest that what prevents interoperability between verifiers is that the time it takes to interact with another verifier is often dominated by the time it takes to convert a problem into the representation used by the "foreign" verifier [5]. In our experience, this phenomenon also applies to the interoperability between refinement based approaches and post-hoc verification based approaches.

Take the binary search algorithm we developed in this section for example. We find that the equivalent recursive program in Spec# time-out when it is verified. In fact, our experience shows that Spec# does not perform very well on recursive program (due to the two different semantics of assertion languages).

Our integrated development framework takes this into consideration. As shown in Fig. 1, we suggest to translate every recursive algorithm ALGORITHM into a partially annotated and iterative OPTIMISED ALGORITHM to be verified within the Spec# Programming System. In [10], we have proposed and proved a sound translation procedure from ALGORITHM to OPTIMISED ALGORITHM to perform this task. For example, the iterative version of the binary search algorithm in Spec# is shown in Fig. 3.

By sending this program to Spec#, Spec# reports the program as verified. No user interaction is required in this 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. The automatic verification of the final Spec# program is available online at http://www.rise4fun.com/SpecSharp/kyKW.

```
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;}
   }
 }
}
```

Fig. 3. Binary Search C# program corresponding to the generated iterative procedure.

6 Further Case Studies

We now summarize several case studies that have been developed using our methodology and tool-kit. We give the details of the development of an abstract Event B model into a recursive algorithm by summarizing the number of proof obligations required for each case study. Moreover, the procedures used for discharging proof obligations can help understand the automation of this process.

Insertion Sort: The effort of formalisation of the insertion sort algorithm lies in simplifying how to express inserting an element into a sorted list. Our methodology starts with a procedure *sortingspec* which is simply specified by an event *modelling* the pre/post specification of the insertion sort algorithm. As shown in Table 2, during the development, 75 % of the proof obligations is automatically discharged in both the initial specification and in the refined machine $sortingref$.

As shown in Algorithm 3, the call of the procedure `inserting` illustrates the reusability of an already developed problem within our MOdels DEveloped on the shelf (MODES) library of verified procedures. Proof obligations associated with calling these procedures must be discharged to prove the correctness of this procedure call.

Moreover, in this case study, proofs which are manually discharged relates to permutations of the input array. However, they are easier to prove than in the classical iterative algorithms for sorting, since the complexity is hidden by the recursive call.

Exponentiation: This problem is to compute the function a^b using the fact that when b is even, the value to compute is transformed into $\left(a^2\right)^{b/2}$ and when

Algorithm 3. Recursive Algorithm sorting(m,t;st) generated by EB2RC

```
$st := t$; $at := t$;
**if** $m = 1$ **then**
    $st := ide; t$;
**else if** $m \neq 1$ **then**
    $at := sorting(m-1, t)$;
    $st := inserting(m, at)$;
```

Table 2. Proof effort of our refinement approach for the insertion sort case study

Model	Total	Auto	Manual	Reviewed	Undischarged	% auto
sortingspec	4	3	1	0	0	75 %
sortingref	45	33	1	0	11	75 %

b is odd, the value to compute is $a^{b-1} \times a$. These two ways to compute the exponentiation are easy to express in the following Event B context:

$axm1 : a \in \mathbb{N}_1 \land b \in \mathbb{N} \land p \in \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$
$axm4 : \forall n \cdot n \in \mathbb{N}_1 \Rightarrow p(n \mapsto 0) = 1$
$axm5 : \forall n, m \cdot n \in \mathbb{N} \land m \in \mathbb{N}_1 \Rightarrow p(n \mapsto m) = p(n \mapsto m-1) * n$
$axm6 : \forall n, m \cdot n \in \mathbb{N} \land m \in \mathbb{N} \land m/2 * 2 = m-1 \Rightarrow p(n \mapsto m) = p((n \mapsto m-1)) * n$
$axm7 : \forall n, m \cdot n \in \mathbb{N} \land m \in \mathbb{N} \land m/2 * 2 = m \Rightarrow p(n \mapsto m) = p((n * n \mapsto m/2))$

As shown in Table 3, our methodology leads to two refinement steps to develop the exponentiation algorithm. During the refinement, the Rodin proof system automatically *knows* which axioms to use in the context. However, in our experience, the proof obligations involves these axioms are difficult to prove automatically, which results the score of automatically discharged proof obligations to be 79 % in the refined machine. Finally, the refinement of the exponentiation algorithm generates the Algorithm 4 by the EB2RC.

Table 3. Proof effort of our refinement approach for the exponentiation case study

Model	Total	Auto	Manual	Reviewed	Undischarged	% auto
expspec	4	4	0	0	0	100 %
expref	84	66	18	0	0	79 %

Maximum of a List: The maximum of a list is quite complex for the first model which stating the specification of this algorithm. However, we find that this complexity in the specification contributes to the proof automation in the

Algorithm 4. Recursive Algorithm exp(u,v;r) generated by EB2RC

```
$r := 0; u := 0; v := 0; temp := 0;$
if $b = 0$ then
    $r := 1;$
else if $b \neq 0$ then
    if $b \div 2 * 2 = b$ then
        $u := a * a;\ v := b \div 2;\ r := exp(u, v);$
    else if $b \div 2 * 2 = b - 1$ then
        $u := a;\ v := b - 1;$
        $temp := exp(u, v);\ r := temp * a;$
```

Table 4. Proof effort of our refinement approach for the list maximum case study

Model	Total	Auto	Manual	Reviewed	Undischarged	% auto
specmax	5	4	1	0	0	25 %
refmax	49	46	3	0	0	94 %

Algorithm 5. Recursive Algorithm max(f,n,i;m) generated by EB2RC

```
$m := 0; temp := 0; ftemp := 0;$
if $i = 0$ then
    $m := f(0);$
else if $i \neq 0$ then
    $temp := maximum(f, n, i - 1);$
    if $f(i) < temp$ then
        $ftemp := temp;\ m := ftemp;$
    else if $f(i) \geq temp$ then
        $ftemp := f(i);\ m := ftemp;$
```

second model (a score of 94 % as shown in Table 4). The proof obligations that need to manually discharged in the refined model relates to prove the existence of a maximum in a list, which is proved by using a theorem of the context. Finally, Algorithm 5 is generated by EB2RC.

Shortest Paths by Floyd: Floyd's algorithm [4] computes the shortest distances in a graph and is based on an algorithmic design technique called dynamic programming, where simpler sub-problems are first solved before the full problem is solved. It computes a distance matrix from a cost matrix, where the cost of the shortest path between each pair of vertices is $O(|V|^3)$. The set of nodes N is $1..n$, where n is a constant value, and the graph is simply represented by the distance function d ($d \in N \times N \times N \mathrel{+\!\!\!\rightarrow} \mathbb{N}$). When the function is not defined, it means that there is no vertex between the two nodes. The relation of the graph is defined as

Algorithm 6. Recursive Algorithm floyd(l,a,b,g;D,FD) generated by EB2RC

```
$D := D0; Fpath := FALSE; FD1 := FALSE; FD2 := FALSE; FD3 := FALSE;$
if $l = 0 \wedge a \mapsto b \in dom(D)$ then
    $Fpath := TRUE;$
else if $l = 0 \wedge a \mapsto b \in dom(D)$ then
    $Fpath := FALSE;$
else if $l > 0$ then
    $D1, FD1 := floyd(l-1, a, b, g); D2, FD2 := floyd(l-1, a, l, g); D3, FD3 := floyd(l-1, l, b, g);$
    if $FD1 = TRUE \wedge FD2 = TRUE \wedge FD3 = TRUE \wedge D1 \leq D2 + D3$ then
        $D(a \mapsto b) := D1; Fpath := TRUE;$
    else if $FD1 = TRUE \wedge FD2 = TRUE \wedge FD3 = TRUE \wedge D1 > D2 + D3$ then
        $D(a \mapsto b) := D2 + D3; Fpath := TRUE;$
    else if $FD1 = FALSE \wedge FD2 = TRUE \wedge FD3 = TRUE$ then
        $D(a \mapsto b) := D2 + D3; Fpath := TRUE;$
    else if $FD1 = TRUE \wedge (FD2 = FALSE \vee FD3 = FALSE)$ then
        $D(a \mapsto b) := D1; Fpath := TRUE;$
    else if $FD1 = FALSE \wedge (FD2 = FALSE \vee FD3 = FALSE)$ then
        $Fpath := FALSE;$
```

the domain of the function d. n is clearly greater than 1, meaning that the set of nodes is not empty. The distance function d is defined inductively from bottom to top according to the principle of dynamic programming, and axioms define this function. The optimal property is derived from the definition of d itself, because it starts by defining the bottom elements and applies an optimal principle summarized as follows: $D_{i+1}(a,b) = Min(D_i(a,b), D_i(a,i+1) + D_i(i+1,b))$. This means that the distances in D_i represent paths with intermediate vertices smaller than i. D_{i+1} is defined by comparing new paths including $i+1$. D_i is defined by a partial function over $N \times N \times N$. The partiality of d leads to some possible problems in computing the minimum, and when at least one term is not defined, we should define a specific definition for the resulting term. Floyd's algorithm provides an algorithmic process for obtaining a matrix of all shortest possible paths with respect to a given initial matrix that represents links between nodes together with their distance. The method is applied and leads to compute the function d and to store the value into D. Algorithm 6 is generated by EB2RC.

7 Conclusions and Future Work

In this work, we illustrated the blueprint of our integrated development framework to combine the efforts of two formal software engineering techniques: program refinement as supported by Event B and post-hoc program verification as supported by the Spec# programming system. Our goal is to improve the usability of formal verification tools by providing a general framework for integrating these two approaches to software verification. We identified and discussed on the strengths of each so that their integration makes the verification task more approachable for users.

We detailed one of the core steps in our integrated development framework, which is the final concrete specification is transformed into an executable recursive algorithm. This has been implemented by the EB2RC, a plug-in for the Rodin platform, that reads in an Event B model and uses the control framework introduced during the models refinement, to generate both a graphical representation of the executable algorithm and a recursive algorithm that is easily translated into executable code.

This work builds on a method for code generation that is detailed by one of the authors in [8,9] 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 [11] can also produce a program from the PROCESS machine. However, the control variable is not removed and the resulting code is not structured.

Our experience shows that our approach assists students in developing and understanding the tasks of software specification and verification. Moreover, we used the technique in lectures to demonstrate how the proof process can be made simpler when one uses a recursive program. A recursive program *hides* many aspects of the computations which appear to be magic. The fantasy is obtained by these events modelling recursive calls. The key idea is to use the call-as-event principle. Since the invariants are easy to discover, the proofs are also easier even if the main technical questions lie in the specialisation of prover like arithmetic.

It also makes different forms of formal software development more accessible to software engineers, helping them to build correct and reliable software systems. Future work will include the development of further plugins, which will integrate and facilitate the co-operation between Spec# tools and Rodin tools. One major component of this work is the reuse of annotations generated during the refinement of an Event B model to automatically verify iterative algorithms. Deriving loop invariants using these annotations is our particular interest here.

References

1. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010)
2. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., M. Leino, K.R.: Boogie: a modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
3. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: an overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
4. Floyd, R.W.: Algorithm 97: shortest path. Commun. ACM **5**(6), 345 (1962)
5. Kaufmann, M., Moore, S.J.: Some key research problems in automated theorem proving for hardware software verification. Revista de la Real Academia de Ciencias Exactas, Físicas y Naturales. Serie A. Matemâticas **98**(1), 181–195 (2004)
6. Kourie, D.G., Watson, B.W.: The Correctness-by-Construction Approach to Programming. Springer, Heidelberg (2012)
7. Leavens, G.T., Abrial, J.-R., Batory, D., Butler, M., Coglio, A., Fisler, K., Hehner, E., Jones, C., Miller, D., Peyton-Jones, S., Sitaraman, M., Smith, D.R., Stump, A.: Roadmap for enhanced languages and methods to aid verification. In: 5th International Conference on Generative Programming and Component Engineering, Portland, Oregon, pp. 221–235. ACM (2006)
8. Méry, D.: A simple refinement-based method for constructing algorithms. ACM SIGCSE Bulletin **41**(2), 51–59 (2009)
9. Méry, D.: Refinement-based guidelines for algorithmic systems. Int. J. Softw. Inform. **3**(2–3), 197–239 (2009)
10. Méry, D., Monahan, R.: Transforming Event-B models into verified C# implementations. In: 1st International Workshop on Verification and Program Transformation, Saint Petersburg, Russia, pp. 57–73. EasyChair (2013)
11. Méry, D., Singh, N.K.: The EB2ALL code generation tool (2011). http://eb2all.loria.fr/
12. Project RODIN. Rigorous open development environment for complex systems (2004). http://rodin-b-sharp.sourceforge.net/