Correct-by-Construction Synthesis of Sequential Algorithms

Dominique Cansell[1] and Neeraj Kumar Singh[2]

[1] EBRP, Lessy, France
dominique.cansell1@gmail.com
[2] INPT-ENSEEIHT / IRIT, University of Toulouse, Toulouse, France
nsingh@enseeiht.fr

Abstract. Jean-Raymond Abrial introduced a method in 2001 for constructing sequential algorithms using a *correct by construction* approach, which involves using IF and WHILE event merging rules on a concrete model to ensure correctness. However, manual derivation of sequential algorithms is error-prone due to the lack of tool support. To address this issue, we present a tool to automate the merging rules proposed by Abrial. This tool allows users to generate sequential algorithms from a verified abstract model and refined models in Event-B, while preserving given invariants. One key feature of the tool is the creation of a complex guard binary tree structure derived from the Event-B specification, aiding in sequential algorithm generation and error identification. The tool has been evaluated using standard examples developed by Abrial, demonstrating the importance of automating sequential algorithm derivation without relying on post-verification steps.

Keywords: Sequential algorithms · Correct-by-construction · Event-B · Refinement and Proofs.

1 Introduction

A key feature of Event-B [3] and B methods [1] is to offer correct-by-construction support for building complex systems. Several industry initiatives [4, 2, 10] have successfully employed these approaches in the rigorous development of software systems to detect flaws at an early stage of system development and to build confidence in the correctness of their systems. These methods allow us to design a complex system abstractly and then gradually refine it to a concrete level that is very close to implementation. Generating programming language source code manually from concrete models can be a potentially error-prone process. However, there are some prototype tools available, but they may not produce code that closely resembles traditional programs. As a result, it is difficult to employ formal techniques approaches in traditional software development, and many industries avoid using them. We argue that one explanation for this is a lack of tool support for synthesising sequential algorithms or source code as classical programs.

To synthesize sequential algorithms, J. R. Abrial proposed many examples of algorithms construction using refinement with his two merge events rules to produce conditional and loop algorithms and one rule to add initialisation in 2001. From 2014 to

2019 J. R. Abrial gave lecture “Analysing and Constructing Computer Programs" especially in Shangai. Twenty two years after J. R. Abrial ask to D. Cansell to work again on this topic. Many others examples are developted by D. Cansell in the EBRP project: compute the n first prime numbers (many versions), bubbelsort (many versions), Dutch Flag, Better decomposition of a natural n (it's a set of natural numbers where the sum is n and which maximalize the product). A tool to apply JRA's merging rules was developted in 2019 in Shangai [17] (Li Qin team) but the tool is not complete (some side conditions are missing) and require human interaction.

In this study, we present a new automatic tool for automating the merging rules proposed by Abrial under the umbrella of EB2ALL [14, 11, 15] to produce correct sequential algorithms. EB2ALL is a code generation tool developed by N. K. Singh that automates the process of generating code in multiple programming languages, including C, C++, Java, C#, Solidity, and others. EB2Algo tool allows users to generate sequential algorithms from a verified abstract model and refined models in Event-B, while preserving given invariants. One key feature of the tool is the creation of a complex guard binary tree structure derived from the Event-B specification, aiding in code generation and error identification. The tool has been evaluated using standard examples developed by Abrial and Cansell, demonstrating the importance of automating sequential algorithm derivation without relying on post-verification steps. In addition, the evaluation of this tool reveals its potential to improve the creation and verification of sequential algorithms, making it a significant asset for software engineers and academics in the field.

The famous "Dutch Flag" from Dijkstra [6] is used throughout the paper as an example to explain

- how can we develop it correctly using Event-B method
- how can we produce by hand a correct algorithm using JRA's rules
- how can we generate this algorithm (and others) using the new tool EB2Algo illustrated in this paper.

The remainder of this paper is organized as follows. Section 2 provides a brief overview of the key elements of the Event-B modeling language, including refinement. Section 3 describes the development of the Dutch Flag example. Section 4 presents the JRA rules for constructing sequential algorithms and provides side conditions demonstrated using the Dutch Flag example. Section 5 outlines the core idea of designing guard binary tree for deriving sequential algorithms based on JRA rules, followed by the implementation of a Rodin plugin called EB2Algo. Section 6 provides an assessment, and related work is presented in Section 7. Finally, Section 8 concludes the paper with future work.

2 Event-B modeling language

This section presents the fundamental concepts of the Event-B modeling language [3], which is based on set theory and first order logic (FOL), as well as it allows to design a complex system using a *correct-by-construction* approach. The design process consists of a series of refinements of an abstract model (specification) leading to the final concrete model.

There are two main modeling components: *Context* and *Machine*. *Context* model describes the static characteristics of a system using *carrier sets* (s), constants (c), axioms ($A(s,c)$) and theorems ($T\_c(s,c)$ proved with previous axioms or theorems). *Machine* model describes the dynamic behavior of a system using variables (x), invariants ($I(x)$), theorems ($T\_m(x)$ proved with previous invariants, axioms or theorems) and a set of events modifying a set of variables (state) represents the core concepts of a machine. The relationship between Event-B model components is described using terms such as *refines*, *extends*, and *sees*.

An event is a state transition in a dynamic system an event can be deterministic (DE), guarded (GE) or non deterministic (NDE). 0n an event we can define a Before-After predicate (BA(x,x')) which express the relation between x (value of x before the event) and x' (value of x after the event).

DE	$x := E(x)$	$x' = E(x)$
GE	**when** $G(x)$ **then** $x := E(x)$ **end**	$G(x) \wedge x' = E(x)$
NDE	**any** α **where** $G(\alpha, x)$ **then** $x := E(\alpha, x)$ **end**	$\exists \alpha \cdot G(\alpha, x) \wedge x' = E(\alpha, x)$

There are Proof Obligations (PO) to prove the invariant $I(x)$:

$A(s,c) \wedge I(x) \wedge BA(x,x') \Rightarrow I(x')$. We have also a PO for the initialization using an After Predicate.

The refinement process allows for the introduction of new features or more specific behavior to a model of a system. This technique allows for the gradual modeling of a system and the strengthening of invariants to incorporate more detailed behavior. By modifying the state description, the refinement approach transforms an abstract model into a more concrete version. This is achieved by refining each abstract event to its corresponding concrete version or by adding new events. The abstract and concrete state variables are connected through gluing invariants. The verification process ensures that each abstract event is correctly refined by its concrete version through the generation of proof obligations. For example, if the abstract model AM has a state variable x and an invariant $I(x)$, it is refined by the concrete model CM with a variable y and a gluing invariant $J(x,y)$ (If there are common variables the abstracts one are renamed and the equality between both is added to the gluing invariant). For each event we have an abstract before-after predicate $BAA(x,x')$ and a concrete one $BAC(y,y')$. The following PO prove invariance and refinement of the event:
$A(s,c) \wedge I(x) \wedge J(x,y) \wedge BAC(y,y') \Rightarrow \exists x' \cdot J(x',y') \wedge BAA(x,x')$. We have also a PO for the initialization using an After Predicate.

For a variant $V(y)$ each convergent event decrease the positive variant:
$A(s,c) \wedge I(x) \wedge J(x,y) \wedge BAC(y,y') \Rightarrow V(y) \geq 0 \wedge V(y') < V(y)$.
More details on POs can be found in [3].

Rodin is an open source framework that supports the development, verification and validation of Event-B models. It offers model checking, animation with ProB, and code generation features. It also allows for the integration of external provers related to first-order logic (FOL) and satisfiability modulo theories (SMT), aiding in the proof process for increasing proof automation. Additionally, Rodin enables the development of plug-ins extensions to enhance its core functionality as well as to provide interoperability with other tools.

3 Correct-by-Construction Modeling of Dutch Flag using Event-B

In this section, the Dutch Flag case study is used as a practical example throughout the paper to illustrate the concepts and the JRA's IF and WHILE rules, as well as the development of the plugin EB2Algo for automation.

The well-known "Dutch Flag" from Dijkstra [6] is compared to the quick sort partition operation, but it is also a sorting algorithm based on three ordered colors (blue, white, and red). If all three colors are present in the array (blue, white, and red), the array will contain all blue values first, then all white values, and finally all red values.

At the end, g contains all of the values of f, but in order. A sorting algorithm of f finds a permutation PI of index of f such that $PI; f$ is sorted. We will demonstrate how to build this program from this property.

3.1 Abstract model of Dutch National Flag

In a Rodin context, there are constants n and f representing a natural number ($n \in \mathbb{N}1$) and a function (($f \in 0..n-1 \rightarrow 0..2$), respectively. n represents a number greater than or equal to 1, while f contains values to be sorted. The values in f correspond to colors - 0 is blue, 1 is white, and 2 is red. The algorithm's result will be stored in the variable g. An abstract event final will compute the result in a single "magically" shot.

```
EVENT final
  any
    PI
  where
    PI ∈ 0..n − 1 ⤖ 0..n − 1
    ∀i, j · i ∈ 0..n − 1 ∧ j ∈ i..n − 1 ⇒ f(PI(i)) ≤ f(PI(j))
  then
    g := PI; f
  end
```

$g \in 0..n-1 \rightarrow\!\!\!\!\!\!\!+\;\, 0..2$ is the trivial invariant. Each new event (in refinement) holds this invariant. If final occurs, g will well contain all the values of f in the good order. It's clearly a sorting algorithm. This model is very close to the PRE and $POST$ conditions of an algorithm.

$$\{PRE\}\ Algo\ \{POST\}$$

```
EVENT Algo
  when
    POST
  end
```

If $Algo$ is an abstract algorithm we get this event, where constants hold PRE. Thanks to the refinements which ensure that all refinements of event final hold post-condition and thanks to variants which ensure that all new events will not take the control for ever then event final will trigger.

3.2 First refinement: compute permutation

To handle the refinement proof for the event final, we require the following invariants: we have the permutation variable Pi, as well as the variables b, w, and r.

$Pi \in 0..n-1 \rightarrowtail\!\!\!\!\twoheadrightarrow 0..n-1$
$b \in 0..n$
$w \in 0..n$
$r \in -1..n-1$

$b \leq w$
$\forall i \cdot i \in 0..b-1 \Rightarrow\ f(Pi(i)) \ = \ 0$
$\forall i \cdot i \in b..w-1 \Rightarrow\ f(Pi(i)) \ = \ 1$
$\forall i \cdot i \in r+1..n-1 \Rightarrow\ f(Pi(i)) \ = \ 2$

The defined variables are initialize as $Pi := 0..n-1 \lhd id \mid\mid g := f \mid\mid b := 0 \mid\mid w := 0 \mid\mid r := n-1$. The event final has been refined in the following ways, and three new events (swap_wr, swap_bw and progress_w) have been added. In the event final, a new guard $w > r$ is added, and a witness for the abstract variable PI is defined. This event's action is updated with a new witness Pi and becomes $g := Pi; f$. In the event swap_wr, we introduce two guards $w \leq r$ and $f(Pi(w)) = 2$, as well as three actions for swapping in Pi, decreasing r by 1, and abstractly computing g.

```
EVENT final
  when
    w > r
  with
    PI = Pi
  then
    g := Pi; f
  end
```

```
EVENT swap_wr
  when
    w ≤ r
    f(Pi(w)) = 2
  then
    Pi := Pi ⊲- {w ↦ Pi(r), r ↦ Pi(w)}
    r := r − 1
    g :∈ 0..n − 1 ⇸ 0..2
  end
```

In the next event swap_bw, we introduce two guards $w \leq r$ and $f(Pi(w)) = 0$, and four actions for swapping in Pi, increasing b and w by 1, and abstractly computing g. Finally, in the event progress_w, we introduce two guards $w \leq r$ and $f(Pi(w)) = 1$, and an action for increasing w by 1.

```
EVENT swap_bw
  when
    w ≤ r
    f(Pi(w)) = 0
  then
    Pi := Pi ⊲- {w ↦ Pi(b), b ↦ Pi(w)}
    b := b + 1
    w := w + 1
    g :∈ 0..n − 1 ⇸ 0..2
  end
```

```
EVENT progress_w
  when
    w ≤ r
    f(Pi(w)) = 1
  then
    w := w + 1
  end
```

Additionally, we introduce a variant $r - w$ in this refinement for all the introduced events. To ensure the correct refinement, all the generated proof obligations (POs) are discharged using Rodin proving tools.

3.3 Second refinement

In this refinement, we remove the permutation variable Pi that is now hide in variable g by using the gluing invariant: $g \ = \ Pi \ ; \ f$. All events are further refined below.

There is only one guard $w > r$ that must be satisfied for the event final in order to determine the final results of algorithms computed in g. Because of the gluing invariant, no action is required in this event. In the refined event swap_wr, we introduce two new guards $g(w) \neq 1 \ \wedge \ g(w) \neq 0$ by refining the abstract guard $f(Pi(w)) = 2$ $((Pi; f)(w) = g(w)$. The gluing invariant holds because $g \lhd\!\!- \{w \mapsto g(r), r \mapsto g(w)\} = (Pi \lhd\!\!- \{w \mapsto Pi(r), r \mapsto Pi(w)\}); f$

EVENT final
when
$w > r$
then
end

EVENT swap_wr
when
$w \leq r$
$g(w) \neq 1$
$g(w) \neq 0$
then
$g := g \lhd\!\!- \{w \mapsto g(r), r \mapsto g(w)\}$
$r := r - 1$
end

In the next event, swap_bw, the abstract guard $f(Pi(w)) = 0$ is refined by introducing two new guards: $g(w) = 0$ and $g(w) \neq 1$ (which can be a guard theorem). In the actions, the action related to Pi is eliminated, and g is updated with a swapping between $g(b)$ and $g(w)$. Finally, in the event progress_w, the abstract guard $f(Pi(w)) = 1$ is refined to $g(w) = 1$, while the actions of the event remain the same.

EVENT swap_bw
when
$w \leq r$
$g(w) = 0$
$g(w) \neq 1$
then
$g := g \lhd\!\!- \{w \mapsto g(b), b \mapsto g(w)\}$
$b := b + 1$
$w := w + 1$
end

EVENT progress_w
when
$w \leq r$
$g(w) = 1$
then
$w := w + 1$
end

4 Construction of Sequential Algorithms

In 2001, J. R. Abrial defined two rules for generating a correct sequential algorithm (see Chapter 15 in [3]). These two rules, M_IF and M_WHILE, merge two events (or algorithms). A guarded algorithm can be written as **when** B **then** S **end**, where S is an algorithm: such as assignment (:=), a conditional, a sequential or a loop.

The JRA rules for M_IF, M_WHILE, and M_INIT are shown below. For more information, please see Chapter 15 of the Abrial book in [3].

when P Q **then** S **end**	**when** P $\neg Q$ **then** T **end**	$\leadsto$	**when** P **then** **if** Q **then** S **else** T **end** **end**	M_IF
when P Q **then** S **end**	**when** P $\neg Q$ **then** T **end**	$\leadsto$	**when** P **then** **while** Q **do** S **end** ; T **end**	M_WHILE
Init	**final_Algo**	$\leadsto$	**Init ; final_Algo**	M_INIT

4.1 Side conditions

A list of side conditions proposed by Abrial is provided as follows:

- Rule M_IF is applied when both events are defined in the same refinement or when the rule M_WHILE cannot be applied,
- Rule M_WHILE is applied when the first event is defined in a deeper refinement and when P is preserved under S when Q holds. When the loop finish $\neg Q$ holds and T occurs. It's the second event only is P holds (P must be invariant under S). In other case we apply rule M_IF,
- Rule M_INIT is apply only when we have a non guarded event FINAL_ALGO (all events are merged).

J. R. Abrial defined these rules before the notion of *anticipated* or *convergent* events. The order to decide between an `IF` or a `WHILE` is slightly different. We assign two levels for an event evt. To identify a machine we assign to each machine a number: 0 for the abstract machine and i for the ith refinement.

To compute convergent and refinement levels, we define the following two functions: `convlvl(evt)` is the refinement level where evt is convergent; `deflvl(evt` is the refinement level at which evt is defined for the first time.

The level of an event evt is `convlvl(evt)` $\mapsto$ `deflvl(evt)`, and the order to compare two events is the lexicographic order. The event final is never convergent, but for convenience `convlvl(final)` is set to 0. The following are the convergent and refinement levels for all events:

$level(\texttt{final}) = 0 \mapsto 0, level(\texttt{swap\_bw}) = 1 \mapsto 0, level(\texttt{swap\_wr}) = 1 \mapsto 0$ and $level(\texttt{progress\_w}) = 1 \mapsto 1$.

The level of an algorithms is the small level (the more abstract one).

4.2 Construction of the Dutch Flag algorithm

This section details the merging process specifically related to the last concrete model of the Dutch Flag.

To merge the events swap_bw and swap_wr together, an `IF` rule can be used since both events have the same level, which is $1 \mapsto 0 = 1 \mapsto 0$. The merged event swap_bwr is shown below.

EVENT swap_bwr
when
$w \leq r$
$g(w) \neq 1$
then
if $g(w) = 0$ **then**
$g := g \Leftarrow \{w \mapsto g(b), b \mapsto g(w)\}$
$b := b + 1$
$w := w + 1$
else
$g := g \Leftarrow \{w \mapsto g(r), r \mapsto g(w)\}$
$r := r - 1$
end

EVENT swap_bwr_progress_w
when
$w \leq r$
then
if $g(w) = 1$ **then**
$w := w + 1$
else
if $g(w) = 0$ **then**
$g := g \Leftarrow \{w \mapsto g(b), b \mapsto g(w)\}$
$b := b + 1$
$w := w + 1$
else
$g := g \Leftarrow \{w \mapsto g(r), r \mapsto g(w)\}$
$r := r - 1$
end
end

Furthermore, we can merge the events swap_bwr (`Level:` $1 \mapsto 0$) and progress_w (`Level:` $1 \mapsto 1$) together. It is not possible to use a `WHILE` rule since the condition $w \leq r$ is not preserved by $w := w + 1$. Instead, we apply the rule M_IF. Apply the M_IF rule, the merged event swap_bwr_progress_w is shown above.

Finally, we can merge the events swap_bwr_progress_w (`Level:`$1 \mapsto 0$) and final (`Level:` $0 \mapsto 0$) together using the M_WHILE rule since the condition $\top$ **true** is always preserved. The merged event swap_bwr_progress_w_final is shown below.

EVENT swap_bwr_progress_w_final
while $w \leq r$ **do**
  if $g(w) = 1$ **then**
    $w := w + 1$
  else
    if $g(w) = 0$ **then**
      $g := g \Leftarrow \{w \mapsto g(b), b \mapsto g(w)\}$
      $b := b + 1$
      $w := w + 1$
    else
      $g := g \Leftarrow \{w \mapsto g(r), r \mapsto g(w)\}$
      $r := r - 1$
    end
  end
od ;

EVENT Algo_Dutch_Flag
$g := f \;||\; b := 0 \;||\; w := 0 \;||\; r := n - 1$;
while $w \leq r$ **do**
  if $g(w) = 1$ **then**
    $w := w + 1$
  else
    if $g(w) = 0$ **then**
      $g := g \Leftarrow \{w \mapsto g(b), b \mapsto g(w)\}$
      $b := b + 1$
      $w := w + 1$
    else
      $g := g \Leftarrow \{w \mapsto g(r), r \mapsto g(w)\}$
      $r := r - 1$
    end
  end
od ;

Finally, we can use the rule M_INIT to merge the events Initialisation and swap_bwr_progress_w_final together. The final merged event `Algo_Dutch _Flag` is shown above.

5 Tool Support for Sequential Algorithms

This section describes the core development of the EB2Algo tool, specifically the derivation of the guard binary tree, determining side conditions for JRA rules, determining `IF` and `WHILE` rules, synthesising sequential algorithms, and Rodin plugin implementation. Furthermore, each step is illustrated with the Dutch Flag example.

5.1 Derivation of Guard Binary Tree

A guard binary tree is a form of binary tree that uses guards to split a set of events in its nodes. Each node in the tree has a guard condition and a collection of events. The guard condition is used to determine how to partition the events into left and right child nodes. For example, let's say we have a guard binary tree with a root node containing the guard condition Q. If an event satisfies Q, it is placed in the left child node; otherwise, it is placed in the right child node. The partitioning process is applied recursively to assign each event in the leaf node, creating the entire tree structure. In addition to the guard condition, a guard binary tree can also include other checks, such as ensuring that the tree is a full binary tree. A full binary tree is a tree in which every node has either 0 or 2 children. This requirement helps maintain the structural integrity of the tree.

Furthermore, a great care must be taken to guarantee that all tree nodes include only events that meet either Q or $\neg Q$. Each leaf node can only contain one event. There is an issue with generating the guard binary tree if a leaf node includes more than one

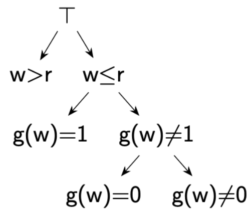
Algorithm 1 An algorithm for deriving Guard Binary Tree

```
function GuardBinaryTree(treeNode, evtList, grd)
    evtLeftList ← ϕ
    evtRightList ← ϕ
    for each e_i ∈ evtList do
        if grd ∈ guardsOf(e_i) then
            evtLeftList ← evtLeftList ∪ {e_i}
            treeNode.Left ← setGrdEvts(grd, evtLeftList)
        else if ¬grd ∈ guardsOf(e_i) then
            evtRightList ← evtRightList ∪ {e_i}
            treeNode.Right ← setGrdEvts(¬grd, evtRightList)
        else
            return False
        end if
    end for
    if (card(evtLeftList) = 1) ∧ card(evtRightList) = 1) ∧ No guard left in evtLeftList and evtRightList then
        return True
    else
        return False
    end if
    if (evtLeftList > 1) then
        for each g_i ∈ guardsOf(evtLeftList(1)) do
            if g_i is not added in Binary Tree then
                grdLeft ← g_i
            end if
        end for
        return GuradBinaryTree(treeNode.Left, evtLeftList, grdLeft)
    end if
    if (evtRightList > 1) then
        for each g_i ∈ guardsOf(evtRightList(1)) do
            if g_i is not added in Binary Tree then
                grdRight ← g_i
            end if
        end for
        return GuradBinaryTree(treeNode.Right, evtRightList, grdRight)
    end if
end function
```

event, there is no common guard condition between the left and right child nodes, or there are no events in either the left or right child node. The core algorithmic structure for recursively deriving a guard binary tree is presented by Algorithm 1. We employ additional predefined functions in our algorithm. These functions are `guardsOf` to extract a list of guards for an event, `setGrdEvts` to update a tree node with a guard and a list of events, and `selectCommonGuard` to determine a common guard for a group of events.

5.1.1 Guard Binary Tree of Dutch Flag

The guard binary tree of the Dutch Flag is depicted in Fig. 1. Fig. 1(a) is derived from the concrete model of the selected case study, and this tree is equivalently presented in the implemented tool in Fig. 1(b). This tree is derived using our implemented Algorithm 1. The root node is initially empty, indicated as $\top$ and the first guard ($r < w$) is chosen from the final event to discover a list of events for the left and right nodes. Because this guard is only in the final event, the left node has only one event and the right node contains the remaining events (progress_w; swap_bw; swap_wr). Then a common guard ($w \leq r$) is identified from the list of events on the right node. We do not need to split any more as the left node



(a) Guard binary tree layout

```
(⊤) [final; progress_w; swap_bw; swap_wr]
├── (w>r) [final]
└── (w≤r) [progress_w; swap_bw; swap_wr]
    ├── (g(w)=1) [progress_w]
    └── (g(w)≠1) [swap_bw; swap_wr]
        ├── (g(w)=0) [swap_bw]
        └── (g(w)≠0) [swap_wr]
```

(b) Guard binary tree in Rodin

Fig. 1: Guard binary tree of the Dutch Flag

has only one event. However, the right node includes three events, we choose a guard from the list of events not used in the guard binary tree. This guard ($g(w) = 1$) is selected from the event progress_w, which is used to split all three events for the left and right nodes. The left node has only one event, progress_w, and the right node has two events, (swap_bw; swap_wr) with the same guard ($g(w) \neq 1$). As the right node has two events, we choose a guard from the list of events not used in the guard binary tree. This guard ($g(w) = 0$) is selected from the event swap_bw, and it used to split the events into the left and right nodes, and a guard is chosen form the right node. Finally, the left and right nodes have only one event, we do not need to split any more left and right nodes. Note that the obtained binary is a full binary tree satisfying the required condition for synthesizing sequential algorithms.

5.2 Determining side conditions

This section presents an overview of additional developed algorithms for establishing convergence and refinement levels, guard preservation, determining `IF` and `WHILE` rules for merging events, and finally synthesis of sequential algorithms.

5.2.1 Determining convergence and refinement levels. There are two important elements, *convergence* and *refinement*, of Event-B play a crucial role in choosing IF and WHILE rules on the left and right nodes for synthesizing sequential algorithms. In Event-B, convergence refers to how a model behave along the refinement hierarchy. Convergence happens when a model's behavior remains constant as refinement advances, which means that any properties met in the initial model are satisfied in all successive refinements. Convergence is important because it ensures that the refined models consistent with the initial model and maintain the required properties are preserved. Event-B employs refinement levels to represent various stages of abstraction and detail in system development. The refinement hierarchy begins with an abstract model and evolves through refinements to a concrete implementation. Each refinement level adds additional detail, perfecting previous levels' behaviors and attributes. The levels are organized in a hierarchical system, with each level improving on the one before it. The convergence and refinement level for each event is defined by the first time an event is labeled as a convergent event, and the concrete event is presented for the first time at any level of the refinement chain, respectively. Note that the leaf nodes contain only a single event, the convergence level and refinement level may be derived directly, whereas nodes with many events must identify the convergent level and refinement level based on lexicography comparison of merging events. The determined convergent and refinement levels are set for each node of the guard binary tree (see Fig. 2).

Convergent and refinement levels of the Dutch Flag. Fig. 2 depicts a guard binary tree with convergent and refinement levels. The final event is introduced with the convergent tag at the abstract level, so it defined as $(0, 0)$. The events swap_bw, swap_wr are refined events of the abstract event swap and were tagged as convergent in the first refinement, therefore they have the same convergent and refinement levels $(1, 0)$. The progress event is introduced in the first refinement and tagged as convergent event, so it has both convergent and refinement levels $(1, 1)$. The convergent and refinement levels of the combined events swap_bw; swap_wr is determined through the lexicography comparison of convergent and refinement levels of each events, so it is determined as $(1, 0)$. Similarly, for the combined events progress_w; swap_bw; swap_wr the convergent and refinement levels are determined as $(1, 0)$. The levels computed by the tool are equivalent to the manually computed levels in Section 4.

5.2.2 Guard Preservation. Guard preservation is an important property to determine side conditions, particularly when selecting the `WHILE` rule. If this property holds then the `WHILE` rule is used; otherwise the `IF` rule is used. Guard preservation in the same node and all upper level guard binary tree nodes refers to the property that if a guard predicate is true at the current node, it remains true as we move up the tree towards the root. In order to demonstrate guard preservation, we need to show that all the events for the current node and other upper level binary tree nodes do not modify the free variables used for defining the upper level guards. This ensures that the guard predicate is preserved throughout the tree. Note that if we do not modify free variables, the guard is retained; however, if we modify free variables, we must check it using another method (which is beyond the scope of this study). POs are an alternate method for ensuring guard property preservation. In the future, we will incorporate such a method within our tool. The guard binary tree contains preserved guards at each node, which can be used to determine the necessary condition for the `WHILE` rule (refer to Fig. 2).

Guard preservation of the Dutch Flag. Fig. 2 depicts a guard binary tree with a list of guards visited in upper level nodes. For example, in the node [progress_w; swap_bw; swap_wr], both the left and right nodes do not have the same convergence and refinement levels, so we can use the `WHILE` rule if only if the the guard ($w \leq r$) is preserved. But, it is not preserved by left or right node, because, the free variables (w and r) of the guard ($w \leq r$) are modified by the events progress_w; swap_bw; swap_wr, so the `WHILE` rule is not applicable for merging the events, thus based on side conditions (for more detail see Section 4), we select the `IF` rule for merging these events. Similarly, in the root node, both the left and right nodes do not have the same convergence and refinement levels, but guard is preserved (GP:($\top$)) as shown in Fig. 2, thus the `WHILE` rule is determined to merge all the root node's events [final; progress_w; swap_bw; swap_wr].

```
(⊤) [final; progress_w; swap_bw; swap_wr] (0,0) WHILE ; GP:(⊤)
├── (w>r) [final] (0,0)
└── (w≤r) [progress_w; swap_bw; swap_wr] (1,0) IF
    ├── (g(w)=1) [progress_w] (1,1)
    └── (g(w)≠1) [swap_bw; swap_wr] (1,0) IF
        ├── (g(w)=0) [swap_bw] (1,0)
        └── (g(w)≠0) [swap_wr] (1,0)
```

Fig. 2: Final guard binary tree with side conditions

5.2.3 Determining IF and WHILE Rules Section 3 describes the IF and WHILE rules, proposed by J. R. Abrial, for synthesizing sequential algorithms. These rules are encoded in the developed tool for defining IF and WHILE rules whenever there is a need to merge two nodes. However, it is important to note that these rules are not applicable to the leaf nodes of the guard binary tree. To ensure correct node merging and condition preservation at multiple refinement levels, we calculate the convergent and refinement levels, compare the nodes, verify for upper level guard preservation, and define merging procedures. It is important to note that IF and WHILE condition correspond to the guard predicate Q, while the negation condition of the IF and WHILE corresponds to the guard predicate $\neg Q$. This selection process is explained in more detail in Section 4.

IF and WHILE rules of the Dutch Flag. Fig. 2 depicts a guard binary tree with non-leaf node IF and WHILE rules based on the JRA side conditions by determining convergent and refinement levels as well as guard preservation. For example, the non-leaf node [swap_bw; swap_wr] has an IF rule because the both child nodes have same convergent and refinement level. Similarly, the other intermediate node [progress_w; swap_bw; swap_wr] has also an IF rule, because guard is not preserved, and finally in the root node, we have WHILE rule to merge all the root node events [final; progress_w; swap_bw; swap_wr] because the child nodes have different convergent and refinement levels and guard is preserved (GP:($\top$)).

5.3 Synthesizing sequential algorithm

This section describes synthesizing sequential algorithm using the built guard binary tree. Once the tree is constructed, the algorithm can be synthesized by traversing the tree in a depth-first manner. Furthermore, the current node is utilised to build the required algorithm based on the left and right nodes, the guard chosen based on IF and WHILE rules, and the choice of IF or WHILE rule.

5.3.1 Synthesizing sequential algorithm of Dutch Flag Listings.1.1 shows the generated algorithm from the Dutch flag example. Lines 8-14 are generated by the node, which consists of two events (swap_bw, swap_wr) that use IF rules (see Fig. 2). Similarly, another intermediate node of the guard binary tree with three events (progress_w, swap_bw, swap_wr) using the IF rule with the condition (guard) of the left node in lines 6-8. Finally, Lines 5 and 15 are generated from the root node applying the WHILE rule. Lines 1-4 are used to set the initial value of each variable extracted from the Initialisation event.

```
g := f ||
b := 0 ||
w := 0 ||
r := n - 1
while w ≤  r do
    if g(w)=1 then
        w := w+1
    else if g(w)=0 then
        g := g <+ {w ↦ g(b),b ↦ g(w)} ||
        w := w+1 ||
        b := b+1
    else
        g := g <+ {w ↦ g(r),r ↦ g(w)} ||
        r := r - 1
od;
```

Listing 1.1: Generated Algorithm of Dutch Flag

5.4 Implementation as Rodin plugin

In this section, we introduce our newly developed plugin tool EB2Algo[1], which is designed to generate sequential algorithms from Event-B models in the Rodin platform using the Eclipse development environment. The EB2Algo plugin is part of the EB2ALL [14, 11] project, which focuses on code generation tools and methodologies for Event-B to different programming languages. The plugin utilizes the core architecture of EB2ALL to parse Rodin projects and includes new algorithms for designing guard binary trees, side conditions, guard preservation, and algorithm generation. The plugin allows users to create algorithmic models within the Rodin platform using the user-friendly interface provided by Eclipse. Users can select a Rodin project and the tool will automatically generate a sequential algorithm, which is then stored in a file. The tool also generates a guard binary tree and logs details related to the algorithmic generation, which are stored in a log file. The log file is primarily used to identify any bugs or issues that may have occurred during the algorithm generation process. A dialog box is also displayed to notify users whether the algorithm was generated successfully or if there are any bugs that need to be addressed. With the ability to extend the tool to generate algorithms in different programming languages, EB2Algo provides a convenient and efficient way to automatically generate sequential algorithms from Event-B models within the Rodin platform.
Fig. 3 depicts a screen capture of the EB2Algo within the Rodin environment. Once the plug-in is installed successfully, the Translator/EB2Algo menu along with a tool button will become visible. To create a sequential algorithm for a formal model, the user can select it from either the EB2Algo menu or the tool button, which will bring up a dialog box. This dialog box presents a list of currently active projects, and the user can choose any project to generate the sequential algorithm. The generated algorithm will be accompanied by a log file containing information about the algorithm generation process in the Rodin project folder.

[1]Download: `https://sites.google.com/site/singhnne/eb2algo`

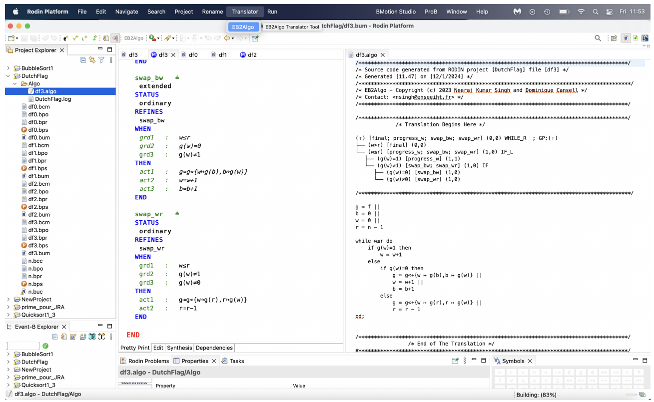


Fig. 3: Screenshot of the EB2Algo plugin in Rodin

6 Evaluation

The developed tool, EB2Algo[1], facilitates the sequential algorithm generation from classic Event-B models using JRA's IF and WHILE rules. These models are designed to support correct-by-construction approaches. We analyze our source model in the Rodin tool and generate sequential algorithms while preserving the given properties. To meet the required side conditions and preserve guards, we derive a guard binary tree from the Event-B models. Our tool, EB2Algo, has successfully generated sequential algorithms for a total of 25 Rodin projects[2]. These projects can be categorized into two sets, with 17 projects initially developed by J. R. Abrial and 8 projects developed by D. Cansell.
The 17 projects developed by J. R. Abrial consist of various algorithms, including:
2 algorithms for performing division using the Euclidean method, 2 algorithms for calculating the square root of an integer, 2 algorithms for searching a specific value within an array or a matrix, 2 algorithms for finding the maximum value in an array, 2 algorithms for finding the minimum value in an array, 1 algorithm for reversing the elements of an array, 1 algorithm for partitioning an array, 1 algorithm for inverting a natural function (an abstract version of the square root), 1 algorithm for sorting elements in an array using the selection sort, method, 2 algorithms for reversing pointers, 1 algorithm for calculating the greatest common divisor (gcd).
On the other hand, the 8 projects developed by D. Cansell consist of different algorithms, including: 7 algorithms for sorting elements in an array using the bubble sort method, 2 algorithms for sorting elements in an array using the quicksort method, 1 algorithm for sorting elements in an array using the selection sort method (a new version using permutation), 1 algorithm for the Dutch flag, 8 algorithms for generating the n^{th} first prime numbers, 1 algorithm for calculating the greatest common divisor (gcd) using the modulus operator, 1 algorithm for better decomposition of natural number.
Overall, our tool has successfully generated sequential algorithms for a diverse range of projects, covering various computational problems and algorithms used in Rodin

[2]Download: `https://www.irit.fr/EBRP/software`

projects. Note that some of the projects consist of multiple models, resulting in different sets of algorithms. We first verify these examples in Rodin and then use our tool to produce sequential algorithms. Additionally, we manually check each algorithm to ensure their correct generation.
Some of the Event-B projects have only two or three refinements, while others have a maximum of 8 to 10 refinements. Some models are complex and contain the required axioms and theorems. Despite this complexity, the verification of the algorithms and the generation of sequential algorithms occur without any problems. In each generated algorithm, a guard binary tree is successfully created, preserving the guards and producing the required algorithm. Through lexicographic analyses of convergence and refinement structure, the IF and WHILE conditions are correctly identified. During the construction of a guard binary tree, if any element (i.e., guard) is not found in the generated tree, the tool raises an exception with precise details. Rodin tool may not determine that the missing element is an error in the model, but it is a new condition that needs to be fulfilled based on JRA's rules.
Our tool, EB2Algo, contains over 5000 lines of code and is user-friendly. It is easily extendable to generate code in various target programming languages. The generated algorithms are stored in a file, and the code generation process is logged in another file. Users can choose to manually evaluate the guard binary tree by selecting the appropriate option. Furthermore, if any issues arise during the generation of the guard binary tree, an exception can be generated along with the guard binary tree in construction: a guard is perhaps missing or too many (theorem).
The developed tool, EB2Algo, and the generated sequential algorithms are available for download from[1].

7 Related Work

Only [17] produces an algorithm uisng JRA's merging rules, but side conditions on guard preservation are not verified. We regret that no other work uses them. Singh et al. [14, 11, 15] propose the EB2ALL tool set as a Rodin plugin for generating code in several programming languages. The fundamental concept of generating source code is quite similar to the structure of Event-B events. Each event is generated as a function with the arguments provided. Finally, all of these functions can be called from the main program via scheduling or employed in the development of complex software systems. A simple plugin B2C is presented in [16] to generate code in C language from Event-B concrete model. In [13], the authors present a code generation tool called EventB2Java for Event-B models. This tool is designed to convert Event-B models into JML-annotated Java programs, with support for a subset of Event-B operations. In [12], the authors present a method for producing VHDL code from Event-B formal models. They use structural similarities between the formal model and hardware description language statements to create an automatic translation algorithm. This algorithm is implemented as a Rodin tool plug-in. In [9], the authors propose an approach to ensure that program code generated from Event-B models is correct. It achieves this by using refinement and well-definedness restrictions, preventing runtime errors caused by semantic differences and addressing issues with different interpretations of integer values.

Refinement techniques are used to show that the generated code correctly implements the original model. A user-friendly scheduling language is also proposed for specifying event execution sequences, including the assertions properties. Note that there is no tool has been implemented with this approach. In [7], the authors describe a new method for building concurrent programs in Ada in Event-B by utilizing Tasking and Shared Machines. A tasking extension structures projects for generating code for multitasking implementations by using refinement, decomposition, and the extension. Further this approach is explored for generating Ada programs in [8]. In [5], the authors describes a code generation approach to produce efficient code from B formal methods [1]. They describe a new translation process architecture that translates the B0 language to a target programming language in sequential code. A case study on Java Card Virtual Machine development demonstrates the approach's effectiveness in generating efficient code.
It should be noted that there is no treatment for developing any sequential algorithm in above mentioned tools (except in [17], which is incomplete) particularly for Event-B modeling language. This is the first study, as far as we know, to propose a tool for synthesizing sequential algorithms for Event-B models. The developed sequential algorithms can be used to create source code in any programming language.

8 Conclusion and Future Work

This paper presented a new automatic tool, EB2Algo, that automates the merging rules proposed by Abrial. The tool generates sequential algorithms from verified abstract and refined Event-B models that preserve the required invariants. A significant feature of the tool is the creation of a complex guard binary tree structure derived from the Event-B specification, which assists in code generation and error identification. The tool is developed in the Eclipse framework under the EB2ALL umbrella and tested using standard 25 Event-B projects, in which 17 projects developed by J. R. Abrial, and 8 projects (contain 21 algorithms) developed by D. Cansell. The developed tool, EB2Algo, and the generated codes can be downloaded from[1]. This is a key step in automatically deriving sequential algorithms from the Event-B model without relying on post-verification.
This work leads to several new perspectives. First, to derive a concurrent algorithm from the sequential version actions, where the derived algorithm can be implemented for concurrent systems. Another key aspect in our work involves generating sequential algorithms in the preferred programming language. The next challenge is to derive more merging rules by expressing the sequential order of events using control variables. Finally, the last goal is for generating proof obligations that ensure the correct preservation of conditions (guards) associated with control flow statements or constructs during program execution. This ensures that the desired behavior specified by the guards is maintained and prevents inconsistencies in the program's logic.

Acknowledgements. Thank you so much, J. R. Abrial, for your merging rules and all. Thanks to Li Qin for many discussions on this topic and interesting feedback on our proposition. The authors also acknowledge the ANR-19-CE25-0010 *EBRP:EventB-Rodin-Plus* project.

References

1. Abrial, J.: The B-book - assigning programs to meanings. Cambridge University Press (1996)
2. Abrial, J.R.: Formal methods: Theory becoming practice. JUCS - Journal of Universal Computer Science **13**(5), 619–628 (2007)
3. Abrial, J.: Modeling in Event-B - System and Software Engineering. Cambridge University Press (2010)
4. Behm, P., Benoit, P., Faivre, A., Meynadier, J.M.: Météor: A successful application of b in a large project. In: Wing, J.M., Woodcock, J., Davies, J. (eds.) FM'99 — Formal Methods. pp. 369–387. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)
5. Bert, D., Boulmé, S., Potet, M.L., Requet, A., Voisin, L.: Adaptable translator of b specifications to embedded c programs. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003: Formal Methods. pp. 94–113. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
6. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976), `https://www.worldcat.org/oclc/01958445`
7. Edmunds, A., Butler, M.: Tasking Event-B: An extension to Event-B for generating concurrent code. In: PLACES 2011 (02/04/11) (February 2011), `https://eprints.soton.ac.uk/272006/`, event Dates: 2nd April 2011
8. Edmunds, A., Rezazadeh, A., Butler, M.: Formal modelling for ada implementations: Tasking Event-B. In: Brorsson, M., Pinho, L.M. (eds.) Reliable Software Technologies – Ada-Europe 2012. pp. 119–132. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
9. Fürst, A., Hoang, T.S., Basin, D., Desai, K., Sato, N., Miyazaki, K.: Code generation for Event-B. In: Albert, E., Sekerinski, E. (eds.) Integrated Formal Methods. pp. 323–338. Springer International Publishing, Cham (2014)
10. Lecomte, T., Déharbe, D., Prun, É., Mottin, E.: Applying a formal method in industry: A 25-year trajectory. In: da Costa Cavalheiro, S.A., Fiadeiro, J.L. (eds.) Proceedings of Formal Methods: Foundations and Applications - 20th Brazilian Symposium, SBMF 2017, Brazil. Lecture Notes in Computer Science, vol. 10623, pp. 70–87. Springer (2017)
11. Méry, D., Singh, N.K.: Automatic code generation from Event-B models. In: Thang, H.Q., Tran, D.K. (eds.) Proceedings of the 2011 Symposium on Information and Communication Technology, SoICT 2011, Hanoi, Viet Nam, October 13-14, 2011. pp. 179–188. ACM (2011)
12. Ostroumov, S., Tsiopoulos, L.: Vhdl code generation from formal Event-B models. In: 2011 14th Euromicro Conference on Digital System Design. pp. 127–134 (2011)
13. Rivera, V., Cataño, N., Wahls, T., Rueda, C.: Code generation for Event-B. International Journal on Software Tools for Technology Transfer **19**(1), 31–52 (2017)
14. Singh, N.K.: Using Event-B for Critical Device Software Systems. Springer (2013). https://doi.org/10.1007/978-1-4471-5260-6
15. Singh, N.K., Fajge, A.M., Halder, R., Alam, M.I.: Chapter 8 - formal verification and code generation for solidity smart contracts. In: Pandey, R., Goundar, S., Fatima, S. (eds.) Distributed Computing to Blockchain, pp. 125–144. Academic Press
16. Steve, W.: Automatic generation of C from Event-B. In: Workshop on Integration of Model-based Formal Methods and Tools. http://www.lina.sciences.univ-nantes.fr/apcb/IM_FMT2009/im_fmt2009_proceedings.html (2009)
17. Zhang, X.: Design and implementation of event-b code generation software based on combination rule, East China Normal University Shanghai (2019)