Proceedings of the 11th Rodin User and Developer Workshop, 2024

June 25th, 2024
Bergomo, Italy

Editors:

Asieh Salehi Fathabadi	University of Southampton, UK
Guillaume Verdier	Université de Paris-Est Créteil
Kristin Rutenkolk	Heinrich Heine University Düsseldorf
Neeraj Kumar Singh	University of Toulouse
Sebastian Stock	Johannes Kepler University
Laurent Voisin	Systerel

UNIVERSITY OF Southampton[1]

[1] This proceeding is supported by the HD-Sec project (https://hd-sec.github.io/), which was funded by the Digital Security by Design (DSbD) Programme delivered by UKRI to support the DSbD ecosystem.

Contents

Table of Contents ii

I Summary **iii**

Executive Summary iv

Workshop Programme 1

II Contributions **2**

What's new in Rodin 3.9 and the Theory plug-in 3

Semantics formalisation: Some experience with the Theory Plug-in 5

Validation of Domain and Meta Models: From Event-B Theories to Practice 8

Developing the UML-B modelling tools 10

Correct-by-Construction Synthesis of Sequential Algorithms 18

Schemata of Recursive Functions and Iterative Algorithms 35

Part I

Summary

Executive Summary

Event-B is a formal method for system-level modelling and analysis. The Rodin platform is an Eclipse-based toolset for Event-B that provides effective support for modelling and automated proof. The platform is open source and is further extendable with plug-ins. A range of plug-ins has already been developed including ones that support animation, model checking, UML-B and text editor. While much of the development and usage of Rodin takes place within past and present EU/UK-funded projects: RODIN, DEPLOY, Advance, EBRP, HiClass, HD-Sec, there is a growing group of users and plug-in developers outside these projects.

The purpose of the 11th Rodin User and Developer Workshop was to bring together existing and potential users and developers of the Rodin toolset and to foster a broader community of Rodin users and developers. For Rodin, the workshop provided an opportunity to share tool experiences and to gain an understanding of ongoing tool developments. For plugin developers, the workshop provided an opportunity to showcase their tools and to achieve better coordination of tool development effort.

The one-day programme consisted of presentations on tool development and tool usage. The presentations are delivered by participants from academia and industry. This volume contains the abstracts of the presentations at the Rodin workshop on June 25th, 2024. The presentations are also available online at https://wiki.event-b.org/index.php/Rodin_Workshop_2024.

The workshop was co-located with the ABZ 2024 conference, Bergamo, Italy. The Rodin Workshop was supported by the University of Southampton.

Finally, we would like to thank the contributors and participants, the most important part of our successful workshop.

Organisers

Asieh Salehi Fathabad, University of Southampton
Guillaume Verdier, Université de Paris-Est Créteil
Kristin Rutenkolk, Heinrich Heine University Düsseldorf
Neeraj Kumar Singh, University of Toulouse
Sebastian Stock, Johannes Kepler University
Laurent Voisin, Systerel

Programme of the Rodin Workshop 2024

09:00–10:30

- What's new in Rodin 3.9 and the Theory plug-in - ***Guillaume Verdier***, *Laurent Voisin, Idir Ait-Sadoune*
- Semantics formalisation: Some experience with the Theory Plug-in - ***Son Hoang***, *Laurent Voisin, Colin Snook, Karla Vanessa Morris Wright and Michael Butler*
- Validation of Domain and Meta Models: From Event-B Theories to Practice - ***Michael Leuschel***, *Yamine Ait-Ameur, Guillaume Dupont, Peter Riviere and Neeraj Kumar Singh*

10:30–11:00 Break

11:00–12:30

- Developing the UML-B modelling tools - ***Colin Snook***, *Michael Butler, Thai Son Hoang, Dana Dghaym, Asieh Salehi Fathabadi*
- Correct-by-Construction Synthesis of Sequential Algorithms - ***Dominique Cansell***, *Neeraj Kumar Singh*
- Schemata of Recursive Functions and Iterative Algorithms - ***Dominique Cansell***

Part II

Contributions

What's new in Rodin 3.9 and the Theory plug-in

Guillaume Verdier[1], Laurent Voisin[2], Idir Ait-Sadoune[3]

[1] UPEC
guillaume.verdier@irit.fr
[2] Systerel
laurent.voisin@systerel.fr
[3] Paris-Saclay University, CentraleSupélec, LMF laboratory
idir.aitsadoune@centralesupelec.fr

1 Introduction

The Rodin platform [1] is an integrated development environment for designing software with Event-B [2]. Thanks to support from the French ANR project Event-B Rodin Plus (EBRP, ANR-19-CE25-0010), Rodin and the Theory plug-in are actively updated with bug fixes and implementation of feature requests. We present the evolution of the Rodin platform since ABZ 2023 and provide some news on the ongoing effort to improve the Theory plug-in [3].

2 Rodin 3.9

A release candidate for Rodin 3.9 was released on April 23rd, 2024 and the final release will be published around mid-June, just before ABZ 2024.

Many new proof rules have been implemented:

- several rules have been added to the auto-rewriter:
  - $\min(A) \in A \equiv \top$
  - $\max(A) \in A \equiv \top$
  - $\text{bool}(B = \text{TRUE}) \equiv B$
  - $E \mapsto E \in \text{id} \equiv \top$
  - $E \mapsto E \in r \setminus \text{id} \equiv \bot$
  - $E \mapsto E \in S \lhd \text{id} \equiv E \in S$
  - $E \mapsto E \in r \setminus (S \lhd \text{id}) \equiv E \mapsto E \in S \ntriangleleft r$
- some auto-rewriter rules can also be applied manually:
  - $F \in \{x, y \cdot P(x, y) \mid E(x, y)\} \equiv \exists x, y \cdot P(x, y) \land E(x, y) = F$
  - $E \in \{x \cdot P(x) \mid x\} \equiv P(E)$
- some new manual rules have been added:
  - $f(x) = y \equiv x \mapsto y \in f$
  - rewrite a^n to $a \times a^{n-1}$ with an additional sub-goal $n \neq 0$
  - infer $\text{finite}(\{i \cdot P(i) \mid F(i)\})$ from $\text{finite}(\{i \mid P(i)\})$
- new reasoners have been created on inductive types:
  - $\mathit{datatype}(T1, U1, ...) = \mathit{datatype}(T2, U2, ...) \equiv T1 = T2 \land U1 = U2 \land ...$
  - $\mathit{cons}(a, b, ...) \in \mathit{datatype}(T, ...) \equiv a \in \mathit{destr1set}(T, ...) \land b \in \mathit{destr2set}(T, ...) \land ...$

It is now possible to provide names for new identifiers generated during proofs. For *abstract expression*, one can write $ident = expr$ instead of just $expr$. For *universal quantification introduction*, *existential quantification elimination* and *datatype distinct case*, a comma-separated list of identifiers can be provided in the proof control input. If the provided identifiers are not fresh, they will be used as a base to generate fresh names.

Among bugs fixed, there have been two major ones.

In Rodin 3.8, feature request #371 introduced hiding of rewritten equality identifiers. However, only selected hypotheses were rewritten: the identifier could still appear in default hypotheses, while its equality had been hidden. Now, the identifier is either deselected or hidden depending on whether it appears in default hypotheses or not.

Yannis Benabbi found a breaking bug in the auto rewriter. In some rare cases with nested comprehension sets, the auto rewriter could "prove" a false goal. The bug has been fixed and the auto rewriter's version incremented.

Besides these bugs, miscellaneous crashes and issues related to exception handling have been fixed. Also, the "Prove automatically" setting is now persisted correctly.

3 Theory plug-in

Before the start of the EBRP project, a release candidate for version 4 had been released in 2017, without a final release. A final release of version 4 with a few more bug fixes was released on December 22, 2020, followed by three more bug-fix releases in 2021 and 2022. The remaining issues in the Theory plug-in are complex ones that require more important changes. A large refactoring has been started and is ongoing.

First, the handling of formula factories and extensions (created to represent constructions of the Theory plug-in, such as inductive data types and operators) has been completely reworked. This should fix many issues related to incompatible formula factories being mixed up. Then, the static-checker has been cleaned up and improved, as well as the representation of elements in the Rodin database. The proof obligation generator is the main remaining part that has to be worked on. It will require quite some work as there are many problems with it, particularly in relation to proof rules.

4 Conclusion

Rodin is under active development and new versions are released yearly. The Theory plug-in, which had many issues, is also being updated. It is now going through an important refactoring to fix most complex underlying issues.

References

[1] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. *Rodin: an open toolset for modelling and reasoning in Event-B.* International Journal on Software Tools for Technology Transfer, 12(6):447–466, Nov 2010.

[2] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[3] Thai Son Hoang, Laurent Voisin, Asieh Salehi, Michael J. Butler, Toby Wilkinson, and Nicolas Beauger. *Theory plug-in for Rodin 3.x.* CoRR, abs/1701.08625, 2017.

Semantics Formalisation: Some Experience with the Theory Plug-in (Extended Abstract)

Thai Son Hoang[1 [0000−0003−4095−0732]], Colin Snook[1 [0000−0002−0210−0983]], Karla Vanessa Morris Wright[3 [0000−0002−0146−3176]], Laurent Voisin[2 [0000−0002−2426−0101]], and Michael Butler[1 [0000−0003−4642−5373]]

[1] ECS, University of Southampton, Southampton SO17 1BJ, United Kingdom
{t.s.hoang,cfs,m.j.butler}@soton.ac.uk
[2] Systerel, 1115 rue René Descartes, 13100 Aix-en-Provence, France
laurent.voisin@systerel.fr
[3] Sandia National Laboratories, 7011 East Avenue Livermore, California 94550, USA
knmorri@sandia.gov

In [3], we model the semantics of SCXML [2] using standard Event-B constructs, i.e., contexts and machines (Approach 1). The Event-B contexts capture the SCXML's syntactical elements while SCXML's semantical elements are formalised using Event-B machines. In this talk, we report on our experience formalising SCXML using the Theory Plug-in [1] (Approach 2), in particular in comparison to Approach 1.

Approach 1. Formalisation using Event-B contexts and machines. The formalisation using the contexts and machines is summarised in Figure 1. The main features of this formalisation are:

- The use of constants to define the syntactical elements of SCXML.
- The use of context extension to build the syntactic model gradually.
- The use of axioms to define the syntactic constraints.
- The use of variables and events to capture SCXML's semantical elements.
- The use of invariants to specify the constraints for the consistency of the semantics.
- The use of the composition mechanism to combine different parts of SCXML, namely untriggered statecharts and run-to-completion scheduling.

Approach 2. Formalisation using Theories Plug-in. The formalisation using theories can be seen in Figure 2 The main features of this formalisation are:

- The use of operators and datatypes to define the syntactical elements of SCXML.
- The use of theory inclusion to build the syntactic model gradually.
- The use of well-definedness (WD) operators to define the syntactic constraints.
- The use of operators and datatypes to capture SCXML's semantical elements.

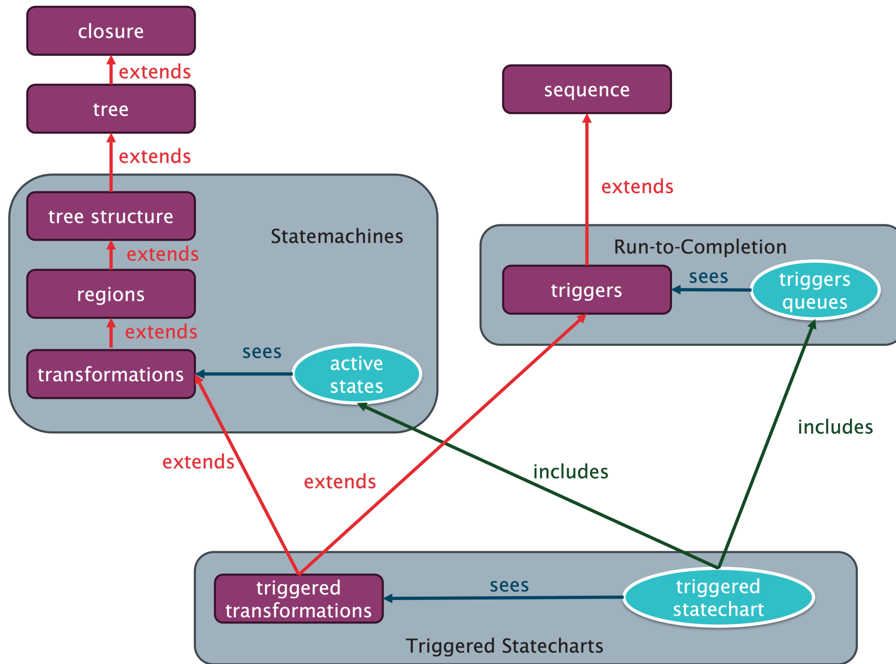


Fig. 1. Formalisation of triggered statecharts using Event-B contexts and machines

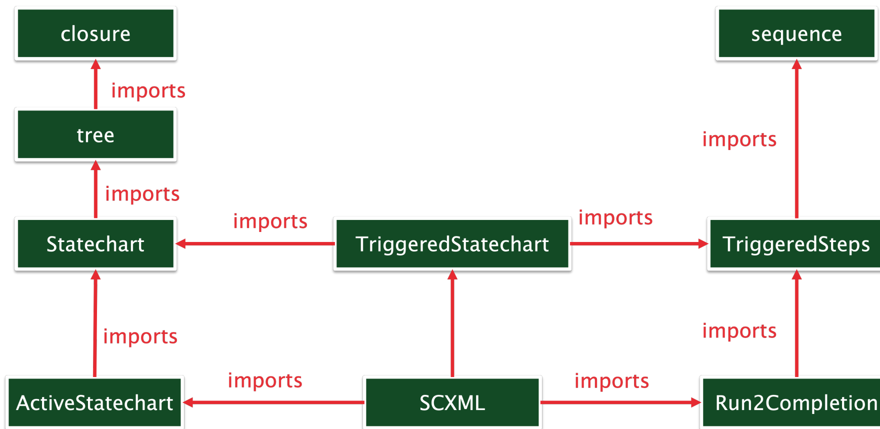


Fig. 2. Formalisation of SCXML statecharts using theories

Approach 1. Standard Event-B	Approach 2. Theory Plug-in
– Model a single SCXML statechart	+ Model a datatype of SCXML statecharts
= Syntactical elements are captured using contexts	= Syntactical elements are captured using theories
+ Syntactical elements are gradually added to the model using context extension	– Gradually introduce syntactical elements results in nested datatype
= Syntactic constraints are represented as context axioms	= Syntactic constraints are represented as WD operators
– Combination of different parts of the language using the composition plugin (i.e., outside of standard Event-B)	+ Composition is done by defining composite datatypes.
= Semantical consistency is encoded as machine invariants	= Semantical consistency is enconded as theory theorems
+ Consistency proof obligations are decomposed automatically (per individual invariants)	– Must manually construct theorems for decomposing the consistency proof
– No customisation for the provers to discharge proof obligations	+ Define proof rules for the provers to discharge proof obligations
– Model-related properties (e.g., refinement) requires additional tool	+ Model-related properties (e.g., refinement) can be stated as theory theorems

Table 1. Comparison between standard Event-B and Theory plug-in

- The use of theorems to specify the constraints for the consistency of the semantics.
- The use of theory inclusion to combine different parts of SCXML, namely untriggered statecharts and run-to-completion scheduling.

Comparison Summary. The comparison between Approach 1 and Approach 2 can be seen in Table 1.

References

1. Butler, M.J., Maamria, I.: Practical theory extension in Event-B. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday. Lecture Notes in Computer Science, vol. 8051, pp. 67–81. Springer (2013), https://doi.org/10.1007/978-3-642-39698-4_5
2. W3C: SCXML specification website. http://www.w3.org/TR/scxml/ (September 2015)
3. Wright, K.V.M., Hoang, T.S., Snook, C.F., Butler, M.J.: Formal language semantics for triggered enable statecharts with a run-to-completion scheduling. In: Ábrahám, E., Dubslaff, C., Tarifa, S.L.T. (eds.) Theoretical Aspects of Computing - ICTAC 2023 - 20th International Colloquium, Lima, Peru, December 4-8, 2023, Proceedings. Lecture Notes in Computer Science, vol. 14446, pp. 178–195. Springer (2023), https://doi.org/10.1007/978-3-031-47963-2_12

Validation of Domain and Meta Models: From Event-B Theories to Practice ⋆

Michael Leuschel[0000−0002−4595−1518],
Yamine Aït-Ameur, Guillaume Dupont, Peter Rivière, Neeraj Kumar Singh

Heinrich-Heine Universität Düsseldorf
leuschel@uni-duesseldorf.de
INP-ENSEEIHT/IRIT, University of Toulouse
{yamine,guillaume.dupont, peter.riviere, neeraj.singh}@enseeiht.fr

The Theory plug-in [5] of Rodin provides the ability to add new data types and operators to Event-B's mathematical toolkit and extend Rodin's proof rules. The theory plug-in plays a major role in a variety of applications: hybrid systems modelling [9, 3], floating point numbers [1], domain modelling [6], and meta modelling (EB4EB) [7]. The EBRP research project aims to improve the support for theories in Rodin. In this article we present how the ProB verification and validation tool [4] was improved to better support Event-B theories and enable validation of the above mentioned case studies.

Meta Models The EB4EB meta modelling framework [7, 2] provides deep and shallow embeddings of Event-B in Event-B itself. We managed to improve ProB so that both embeddings can be validated. This required, e.g., improving symbolic treatment of the relational image operator (used by E4BEB to apply guards, invariants or before-after-predicates to states) and support for inductive data types. Figure 1 shows ProB2-UI for the EB4EB deep embedding of a 24-hour clock. Note that the before-after-predicate of the model is infinite (as it is separate from the guards). Model checking the deep embedding took 1.8 seconds for 1440 states. This is about one order of magnitude slower than the original clock model (0.12 secs), due to the interpretation overhead of EB4EB.

Visualisation of Event-B Models with Theories In the lower middle half of Fig. 1 you can see a visualisation of the current state of the meta model. Here B formulas control the attributes of SVG (sclabale vector graphics) objects, in this case the hour and minute hands of the clock. These attributes are often floating point numbers or strings. As these are not available in Rodin, the VisB formulas are written in classical B with additional access to ProB's external functions. In Fig. 1, an existing SVG image of a clock was used[1] and the hour and minute hands were rotated by setting the transform attributes using floating point calculations. To enable visualisations of Event-B theories in general, we have made the Event-B theory operators available in VisB formulas, as well as in ProB's REPL and other features.

Hybrid Systems and Support for Reals Support for floating point numbers has been added to ProB, in line with Atelier-B's new datatypes FLOAT

⋆ This work was supported by the IVOIRE project funded by DFG/FWF grant # I 4744-N, as well as the EBRP project funded by ANR grant ANR-19-CE25-0010.

[1] https://github.com/tomchen/animated-svg-clock

and REAL. Various B operators now work with floats: +, -, *, /, Σ, Π, min, max. ProB also provides a library (LibraryReals.def) with external functions, e.g., trigonometric functions like RSIN and RCOS. This is useful for visualisations, as seen above, but also for hybrid systems modelled in Event-B. Axiomatic operators in Event-B theories can now be linked to these operators (in .ptm mapping files). This was used to enable animation and visualisation of the floating point theory in [1] as well as floating number approximations of some hybrid models from [9, 3]. In future we want to also support precise reals [8].

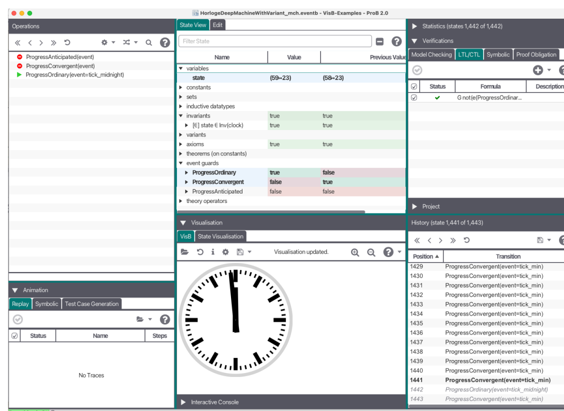


Fig. 1. Screenshot of an animation of deep embedding of clock model

References

1. I. Aït-Sadoune. A floating-point numbers theory for Event-B. In *Proceedings MEDI 2023*, pages 30–43, 2023.
2. Y. A. Ameur, G. Dupont, I. Mendil, D. Méry, M. Pantel, P. Riviere, and N. K. Singh. Empowering the Event-B method using external theories. In *Proceedings IFM 2022*, LNCS 13274, pages 18–35, 2022.
3. G. Dupont, Y. A. Ameur, N. K. Singh, and M. Pantel. Event-B hybridation: A proof and refinement-based framework for modelling hybrid systems. *ACM Trans. Embed. Comput. Syst.*, 20(4):35:1–35:37, 2021.
4. M. Leuschel and M. J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.
5. I. Maamria, M. Butler, A. Edmunds, and A. Rezazadeh. On an Extensible Rule-based Prover for Event-B. In *ABZ2010*, February 2010.
6. I. Mendil, P. Riviere, Y. A. Ameur, N. K. Singh, D. Méry, and P. A. Palanque. Non-intrusive annotation-based domain-specific analysis to certify Event-B models behaviours. In *Proceedings APSEC 2022*, pages 129–138. IEEE, 2022.
7. P. Riviere, N. K. Singh, and Y. A. Ameur. EB4EB: A framework for reflexive event-b. In *Proceedings ICECCS 2022*, pages 71–80. IEEE, 2022.
8. K. Rutenkolk. Extending modelchecking with ProB to floating-point numbers and hybrid systems. In *Proceedings ABZ'23*, pages 366–370, 2023.
9. W. Su, J. Abrial, and H. Zhu. Formalizing hybrid systems with Event-B and the Rodin platform. *Sci. Comput. Program.*, 94:164–202, 2014.

Developing the UML-B modelling tools

Colin Snook[0000−0002−0210−0983], Michael Butler[0000−0003−4642−5373], Thai Son Hoang[0000−0003−4095−0732], Asieh Salehi Fathabadi[0000−0002−0508−3066], and Dana Dghaym[0000−0002−2196−2749]

ECS, University of Southampton, Southampton, U.K.
{cfs, m.j.butler, t.s.hoang, A.Salehi-Fathabadi, D.Dghaym }@soton.ac.uk

Abstract. UML-B is a UML-like diagrammatic front end for the Event-B formal modelling language. We have been developing UML-B for over 20 years and it has gone through several iterations, each with significant changes of approach. The first version was an adaptation of a UML tool, the second generated a complete Event-B project, the third contributed parts of an Event-B model, and the fourth (currently under development) provides a human usable text persistence. Here we outline the reasons for these different developments and summarise the lessons learnt.

1 Introduction and Motivation

Towards the end of the last century it was widely recognised that formal modelling is beneficial in reducing specification errors, but despite various arguments regarding the cost benefits of early error detection, it was difficult to dispel the view that they were costly to achieve and required 'special' engineers or mathematicians. We investigated these beliefs through empirical experiments and interviews with industry experts. The experiments [23] established that formal specifications are no more difficult to understand than computer programs of equivalent complexity. However, when interviewed, industry exponents of formal methods warned that it is the choice of useful abstractions that is difficult and requires experience [22]. Abstraction is something of an art and often counter to the nature of engineers used to looking for solutions. Finding abstractions that are amenable to verification tools adds another complication which can only be mitigated by experience and expertise.

We postulated that a visual modelling tool would aid engineers in exploring and choosing different abstractions. This theory was grounded in 'The Cognitive Dimensions of Notations Framework' [5] which provides a "common vocabulary for discussing many factors in notation, UI or programming language design". (In the following, the terms from the framework are shown in italics). Using this framework, we postulated that, for systems modelling, we need *abstractions* for a *close mapping* to the problem domain, but this requires *premature commitment* (early decisions) which makes specification more difficult especially when compounded by *viscosity* (the effort needed to change the specification) which can be high in a large textual specification with many inter-dependencies. The UML-B

diagrams help by increasing the *visibility* of chosen abstractions through visualisation and reducing viscosity. The reason the diagrams are efficient is because a single diagram entity represents many lines of formal specification text compared to a textual specification. A translation tool then converts the diagram into a textual form for formal verification and validation. This iterative *progressive evaluation* alleviates the difficulty of making premature commitments. A more detailed usability assessment of UML-B using cognitive dimensions is discussed in [17].

The B-method [1] is a method of software development using the formal modelling language, B which is based on set theory and first order predicate logic. It supports the concept of abstraction and incremental refinement with verification by proof. Event-B [2] is a formal modelling language for modelling discrete systems. Event-B was developed from the B-method and hence also supports abstraction and incremental refinement with verification by proof. We chose to use B, and later Event-B, as our underlying formal specification language because they provide a notion of formal refinement with strong tool support for verification using theorem provers as well as model checking and animation tools.

We chose to use the UML (Unified Modelling Language) [18] as the basis for our diagrammatic modelling because it was already fairly widespread and therefore familiar within industry. Event-B models are based on set theory which involves collections of instances and their relationships. This has a natural visualisation as an entity-relationship diagram which can be represented using UML class diagrams. Behaviour in Event-B is modelled as events that fire spontaneously when their guards are true and alter the variables using actions that are treated as a set of simultaneous parallel substitutions. Here there are some important differences between Event-B events and UML state-chart transitions. However, a state-machine representation, similar in structure to UML statecharts, is useful for representing the behaviour of Event-B models. Hence we developed the UML-B diagrammatic modelling tools [19,20] and have been supporting and developing them for over 20 years during which time we have enjoyed many collaborations with various industry sectors. Our current research work, industrial case studies and tool installations are shown on our UML-B website [12].

2 History of UML-B

Driven by experience gained through industrial collaboration, UML-B has been developed over the last 22 years, going through several distinct and fundamentally different versions. This section gives a history of the development of UML-B and the motivation for changing to a new approach in each case.

2.1 Version 1 - Extending standard UML

The initial concept of UML-B (in 2000) was to translate from UML into the B formal notation. (This was before Event-B and Rodin existed). Hence the first

version of UML-B [20] was based on the IBM Rational Rose UML tool. Rational Rose provided a visual basic scripting facility for the user to add tooling features to enhance the diagrams. UML-B was implemented as a script that traversed the UML diagrams and output a B model as a text file. The UML-B model was constructed as a standard UML class diagram but with some restrictions and additional properties added as UML stereotypes. Invariants, could be added to classes and guards and actions could be added to class methods, in order to fully specify the behaviour of the model. The notation used for these textual annotations was derived from the target notation, B, but with support for automatic quantification over instances of a class or parameterisation of the contextual class instance ('self'). Here we may have been able to use OCL for the constraint language and possibly, in a declarative style, for actions. However, this would have entailed more work to invent a translation and caused more separation between the specification and the verification languages. For this reason we took the easier route of basing our constrain/action language on Event-B rather than OCL.

The generated B file was then imported into the B-Core tool [4] for formal analysis. Unfortunately, the Rational Rose tool was a Windows-based application, whereas the B-Core tool was only available for Linux operating systems. Therefore the user had to switch to a different operating system in order to analyse the formal model.

2.2 Version 2 - UML-B: Like UML but different

In 2004 the Rodin project [3,15] was started with the aim of developing a new extensible formal modelling platform to support the new Event-B notation for systems modelling. It includes Event-B editors, static checking tools and mathematical theorem provers for verification of the models. This gave an opportunity to greatly improve UML-B and a new version was developed with a different concept from the first version.

- We no longer tried to bend UML to our purpose but instead, developed our own diagrammatic modelling notation borrowing ideas from UML only when they fitted.
- We had an integrated extensible modelling platform based on Eclipse [8,7] which greatly improved the workflow from source model to verification results.
- The Event-B notation was aimed at systems level modelling and so UML-B followed suit. The concept of UML-B was always more aligned to systems level rather than software development, hence Event-B was a better fit for our purposes.

This version of UML-B [19] generated an entire Event-B project from a UML-B project. Hence all modelling had to be done in UML-B since anything the user did to the Event-B model would be overwritten the next time the UML-B was translated. More and more features were added to UML-B in order to support

different modelling use cases. The action and constraint notation for invariants, guards and actions was continued in this version and developed further by adding new features. Class diagrams and state-machines were supported, but both deviated from their UML counterparts in order to provide a better correspondence with the target formalism. It should be noted that through our industrial collaborations we were gradually appreciating the significance of the very different semantics between UML statecharts and UML-B state-machines. An example of this was that users tended to attach the same event to two transitions of the same state-machine expecting one of them to fire depending on which state was active. However, in UML-B this creates two transitions that must fire together and hence never do so (since both sources can never be active at the same time). Therefore we referred to UML-B as being 'UML-like' from this point on and took care to prepare users for the differences.

The UML-B modelling language used the Eclipse Modelling Framework [24] (EMF) where a meta-model is constructed to define the abstract syntax of a modelling language and the EMF tools then generate Java code that can load model instances of that language and serialise (persist) them. The default format for model serialisation is XMI (an XML based notation for model interchange), but this can be overridden with any user-defined serialisation format. For this version of UML-B, the default XMI format was used for serialisation. EMF is a very useful basis for defining modelling notations and we have continued to use it for all our future version of UML-B as well as any other model tooling that we have developed. We used the Graphical Modelling Framework (GMF) [14] to develop the concrete diagram syntax, editors and tooling.

Although this version of UML-B was quite popular with industrial users that were relatively new to formal modelling, a significant portion of users were already familiar with Event-B and would prefer to have the full flexibility of working in Event-B and using the diagram notations more selectively.

2.3 Version 3 - iUML-B: Extending Event-B

In 2008, The 'Deploy' project [13] was started as a follow on from the Rodin project with the aim of promoting the use of the Rodin platform, and its associated plug-ins such as UML-B, in industry. During this project a new version of UML-B was developed that could work alongside Event-B, rather than overwrite the Event-B models all the time.

Since the new iUML-B needed to be an extension of Event-B rather than a separate language, a new EMF meta-model was needed. An Event-B text editor (Camille) was also developed by Heinrich Heine University in Dusseldorf and since both needed an EMF meta-model for Event-B, researchers at Dusseldorf and Southampton, as well as University of Newcastle, worked together to produce a common EMF based framework and meta-model for Event-B [21] which could be used as the basis for future tools. The iUML-B meta-model then extends the Event-B meta-model to support class diagrams and state-machines using a generic extension mechanism built into the meta-model. The iUML-B model

was serialised (i.e. saved/persisted) within a single extension element within the Rodin Event-B model.

In this version of UML-B, the diagrams still generate Event-B elements but not the complete Event-B model. Some parts of the Event-B model are expected to already exist and the diagrams *elaborate* them by providing further details. For example a UML-B class no longer generates the data item (set, constant or variable) that models the set of instances, but it can generate invariants that constrain the set of instances. Similarly, attributes and associations, 'elaborate' existing data elements by generating invariants about their type (being a relation between the containing class instances and the attribute/association type). Class methods and state-machine transitions 'elaborate' events that already exist in the Event-B and contribute extra parameters, guards and actions. This strategy of elaboration allows the modeller to retain control over the Event-B model and choose which parts to model in Event-B and which parts to model diagrammatically in UML-B. (For expediency, the UML-B diagram editor provides an option to create the elaborated elements if they do not already exist in the Event-B).

However, a disadvantage of diagrammatic models is that it becomes more difficult to get a quick overview of all the details in the model. In a textual syntax all of the details of the model are visible in the same view, even if they are complicated to interpret, whereas in a diagram, it is cumbersome to show everything on the canvas. Hence in UML-B certain model details are given in the associated contextual properties view which only becomes visible when the appropriate model element is selected. This led to some users asking for a human readable text persistence for UML-B. Other advantages of a human readable text persistence are that it may be easier to compare different versions of models (provided order is maintained) and to copy and paste sections of models. (Note that the default persistence is XMI (a variant of XML) which is ASCII text, but designed for machine loading and therefore difficult to read).

2.4 Version 4 - xUML-B: A human usable text persistence

The Camille text editor for Event-B was very popular but still serialised models using the Rodin XML-based format. Another problem with Camille was that it is very difficult to extend a concrete syntax. Hence extensions to the Event-B modelling language (e.g. UML-B) were difficult to accommodate. To obtain an extensible and true human usable text serialisation for Event-B we developed a new 'front-end' for Event-B using XText [6,16] which we call 'CamilleX' [9]. Due to the difficulty of extending Rodin models, CamilleX models are written in a separate human readable text file. Hence the source models are separate from the Rodin models which are automatically re-generated for verification purposes when the CamilleX model is saved. Regeneration is efficient since the translations are very fast and the Rodin verification builders are designed to find and re-use existing proofs wherever possible. Further discussion on the development of CamilleX is given in [10]

However, this meant that the UML-B models can no longer be persisted inside the Rodin models. Hence we are now developing an alternative persistence scheme for iUML-B so that its models are stored separately from the elaborated Rodin models. The new UML-B persistence is also based on XText so that we have a human readable persistence for UML-B. We call this version xUML-B.

3 Conclusions

The main lessons we have learnt from our experiences of developing UML-B are.

- Heavily featured semi-formal modelling languages such as UML are difficult to use for precise formally verifiable specification. While UML covers a wide range of users needs it doesn't support the precise mathematical semantics needed for proof. UML can be specialised through profiles and stereotypes, but users are confused if familiar features are not used or represent different semantics. Therefore, it is better not to try to translate UML but to invent a new notation that is better suited to the target formalism.
- A downside of making a new notation similar to a well-known existing one such as UML, is that users may be confused when the model does not behave as they are used to. An example of this is the difference between UML-B state-machines and UML statechart 'run to completion' semantics.
- Model edition, checking and verification needs to be highly integrated so that changes can be quickly assessed.
- While there are many users that are attracted to a self contained diagrammatic notation, experienced users want the flexibility to choose between diagrammatic and textual representations for different parts of a model.
- Even when diagrams are used, users express a strong desire for a human usable textual persistence which helps with maintenance activities such as version comparison and copy and paste as well as enabling a quick oversight of the content

A common reaction to UML-B is to question the decision not to translate standard UML. There is of course a desire not to proliferate new languages unnecessarily. As we have already discussed, the UML semantics is not easily used for representing Event-B semantics. For example, we have extensively researched ways to reconcile run to completion semantics (used in UML statecharts) with Event-B style refinement [11]. An alternative approach would be to develop a new formalised theory of refinement for UML and provide new theorem provers to support it. However, we believe this would be extremely difficult simply because great care was taken to achieve tractable refinement and proof in Event-B by keeping to a simple and appropriate semantics.

Acknowledgements

This work is supported by the HiClass project (113213), which is part of the ATI Programme, a joint Government and industry investment to maintain and grow the UK's competitive position in civil aerospace design and manufacture.

References

1. J-R. Abrial. *The B Book - Assigning programs to meanings*. Cambridge University Press, 1996.
2. J-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
3. J-R Abrial, M. Butler, S. Hallerstede, T.S. Hoang, F. Mehta, and L. Voisin. Rodin: An open toolset for modelling and reasoning in Event-B. *Software Tools for Technology Transfer*, 12(6):447–466, 2010.
4. B-Core(UK). *B-Toolkit User's Manual, Release 3.2*. Oxford, UK, 1996.
5. A. Blackwell and T. Green. Chapter 5 - notational systems—the cognitive dimensions of notations framework. In J.M. Carroll, editor, *HCI Models, Theories, and Frameworks*, Interactive Technologies, pages 103–133. Morgan Kaufmann, San Francisco, 2003.
6. M. Eysholdt and H. Behrens. Xtext: Implement Your Language Faster Than the Quick and Dirty Way. In *OOPSLA*, pages 307–309. ACM, 2010.
7. The Eclipse Foundation. The Eclipse Project Website. http://www.eclipse.org, 2009. Accessed Sept. 2022.
8. E. Gamma and K. Beck. *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 2003.
9. T. S. Hoang and D. Dghaym. Event-B and Rodin Documentation Wiki: CamilleX. http://wiki.event-b.org/index.php/CamilleX, 2018. Accessed Sept. 2022.
10. T.S. Hoang, C. Snook, D. Dghaym, A. Salehi Fathabadi, and M. Butler. Building an extensible textual framework for the rodin platform. *F-IDE 2022, Lecture Notes in Computer Science (to be published)*, 2022.
11. K. Morris, C. Snook, T. S. Hoang, G. Hulette, R. Armstrong, and M. Butler. Formal verification and validation of run-to-completion style state charts using event-b. *Innovations in Systems and Software Engineering*, Mar 2022.
12. The University of Southampton. The UML-B website. https://uml-b.org/, 2021. Accessed Sept. 2022.
13. The Deploy Project. The deploy project website. http://www.deploy-project.eu/, 2008. Accessed Sept. 2022.
14. The Graphical Modelling Project. The GMP project website. https://www.eclipse.org/modeling/gmp/, 2010. Accessed Sept. 2022.
15. The Rodin Project. Rigorous open development environment for complex systems. http://rodin.cs.ncl.ac.uk/, 2004. Accessed Sept. 2022.
16. The XText Project. The XText project website. https://www.eclipse.org/Xtext/, 2020. Accessed Sept. 2022.
17. R. Razali, C. Snook, M. Poppleton, and P. Garratt. Usability assessment of a UML-based formal modeling method using a cognitive dimensions framework. *Human Technology*, 4(1):26–46, May 2008.
18. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, MA., 1998.
19. M.Y. Said, M. Butler, and C. Snook. Language and tool support for class and state machine refinement in UML-B. In A. Cavalcanti and D. Dams, editors, *FM 2009: Formal Methods*, pages 579–595, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
20. C. Snook and M. Butler. UML-B: formal modelling and design aided by UML. *ACM Transactions on Software Engineering and Methodology*, 15(1):92–122, 2006.

21. C. Snook, F. Fritz, and A. Iliasov. Event-B and Rodin Documentation Wiki: EMF Framework for Event-B. http://wiki.event-b.org/index.php/EMF_framework_for_Event-B, 2009. Accessed Sept. 2022.
22. C. Snook and R. Harrison. Practitioners' views on the use of formal methods: an industrial survey by structured interview. *Information and Software Technology*, 43(4):275–283, 2001.
23. C. Snook and R. Harrison. Experimental comparison of the comprehensibility of a z specification and its implementation in java. *Information and Software Technology*, 46(14):955–971, 2004.
24. D. Steinberg, F. Budinsky, M. Paternostro, and Ed Merks. *Eclipse Modeling Framework*. The Eclipse Series. Addison-Wesley Professional, 2nd edition, 2008.

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 developped 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 postcondition 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
```

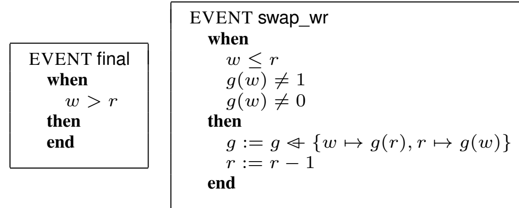
Additionally, we introduce a variant $r - w$ in this refinement for all the introduced events. To ensure the correct refinement, all the generated proof obligations (POs) are discharged using Rodin proving tools.

3.3 Second refinement

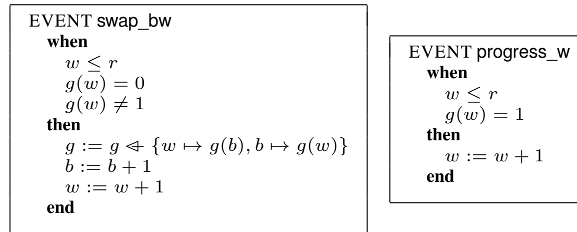
In this refinement, we remove the permutation variable Pi that is now hide in variable g by using the gluing invariant: $g = Pi \,;\, f$. All events are further refined below.

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 $g \Leftarrow \{w \mapsto g(r), r \mapsto g(w)\} = (Pi \Leftarrow \{w \mapsto Pi(r), r \mapsto Pi(w)\}); f$



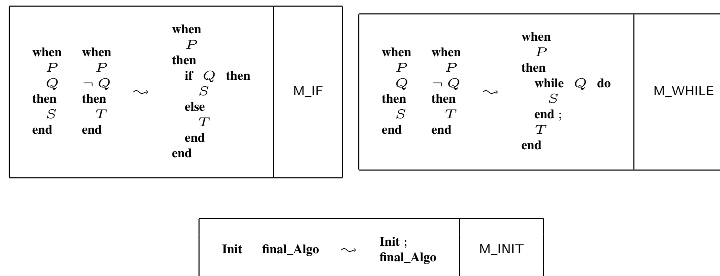
In the next event, swap_bw, the abstract guard $f(Pi(w)) = 0$ is refined by introducing two new guards: $g(w) = 0$ and $g(w) \neq 1$ (which can be a guard theorem). In the actions, the action related to Pi is eliminated, and g is updated with a swapping between $g(b)$ and $g(w)$. Finally, in the event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



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].



Init final_Algo	⇝	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 ≤ r
    g(w) ≠ 1
  then
    if g(w) = 0 then
      g := g ⊲- {w ↦ g(b), b ↦ g(w)}
      b := b + 1
      w := w + 1
    else
      g := g ⊲- {w ↦ g(r), r ↦ g(w)}
      r := r − 1
    end
```

```
EVENT swap_bwr_progress_w
  when
    w ≤ r
  then
    if g(w) = 1 then
      w := w + 1
    else
      if g(w) = 0 then
        g := g ⊲- {w ↦ g(b), b ↦ g(w)}
        b := b + 1
        w := w + 1
      else
        g := g ⊲- {w ↦ g(r), r ↦ g(w)}
        r := r − 1
      end
    end
```

Furthermore, we can merge the events swap_bwr ($\texttt{Level:}\ 1 \mapsto 0$) and progress_w ($\texttt{Level:}\ 1 \mapsto 1$) 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 ($\texttt{Level:}1 \mapsto 0$) and final ($\texttt{Level:}\ 0 \mapsto 0$) 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 ≤ r do
    if g(w) = 1 then
      w := w + 1
    else
      if g(w) = 0 then
        g := g <+ {w ↦ g(b), b ↦ g(w)}
        b := b + 1
        w := w + 1
      else
        g := g <+ {w ↦ g(r), r ↦ g(w)}
        r := r − 1
      end
    end
  od ;
```

```
EVENT Algo_Dutch_Flag
  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)}
        b := b + 1
        w := w + 1
      else
        g := g <+ {w ↦ g(r), r ↦ 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.

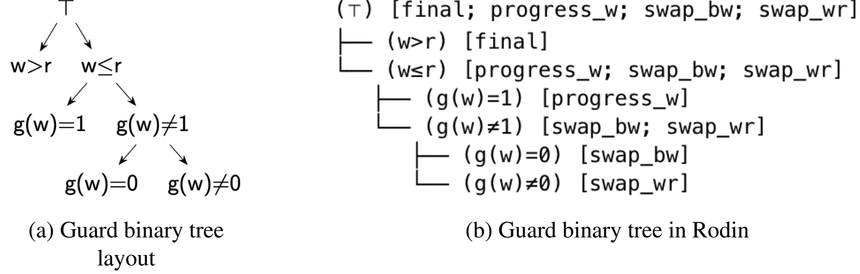
Furthermore, a great care must be taken to guarantee that all tree nodes include only events that meet either Q or $\neg Q$. Each leaf node can only contain one event. There is an issue with generating the guard binary tree if a leaf node includes more than one

Algorithm 1 An algorithm for deriving Guard Binary Tree

```
1: function GUARDBINARYTREE(treeNode, evtList, grd)
2:     evtLeftList ← ϕ
3:     evtRightList ← ϕ
4:     for each e_i ∈ evtList do
5:         if grd ∈ guardsOf(e_i) then
6:             evtLeftList ← evtLeftList ∪ {e_i}
7:             treeNode.Left ← setGrdEvts(grd, evtLeftList)
8:         else if ¬grd ∈ guardsOf(e_i) then
9:             evtRightList ← evtRightList ∪ {e_i}
10:            treeNode.Right ← setGrdEvts(¬grd, evtRightList)
11:        else
12:            return False
13:        end if
14:    end for
15:    if (card(evtLeftList) = 1) ∧ card(evtRightList) = 1) ∧ No guard left in evtLeftList and
   evtRightList then
16:        return True
17:    else
18:        return False
19:    end if
20:    if (evtLeftList > 1) then
21:        for each g_i ∈ guardsOf(evtLeftList(1)) do
22:            if g_i is not added in Binary Tree then
23:                grdLeft ← g_i
24:            end if
25:        end for
26:        return GURADBINARYTREE(treeNode.Left, evtLeftList, grdLeft)
27:    end if
28:    if (evtRightList > 1) then
29:        for each g_i ∈ guardsOf(evtRightList(1)) do
30:            if g_i is not added in Binary Tree then
31:                grdRight ← g_i
32:            end if
33:        end for
34:        return GURADBINARYTREE(treeNode.Right, evtRightList, grdRight)
35:    end if
36: 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

(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].

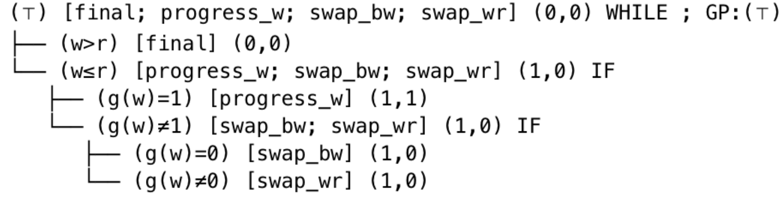


Fig. 2: Final guard binary tree with side conditions

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.

```
1  g := f ||
2  b := 0 ||
3  w := 0 ||
4  r := n - 1
5  while w ≤  r do
6      if g(w)=1 then
7          w := w+1
8      else if g(w)=0 then
9          g := g ⩤ {w ↦ g(b),b ↦ g(w)} ||
10         w := w+1 ||
11         b := b+1
12     else
13         g := g ⩤ {w ↦ g(r),r ↦ g(w)} ||
14         r := r - 1
15 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`

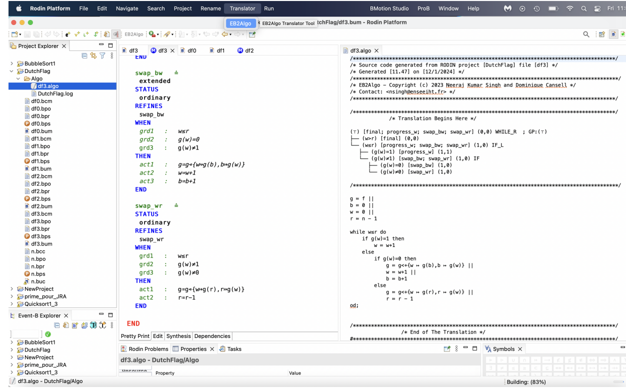


Fig. 3: Screenshot of the EB2Algo plugin in Rodin

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 nth 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)

Schemata of Recursive Functions and Iterative Algorithms

Dominique Cansell (Lessy, EBRP)

1 Description

In [2] we presented the new JRA's instantiation context to define closure, fixpoint (Tarski), well-founded (Noether) and recursion. A new instantiation plugin [4] was developed in the EBRP project [5]. In this paper we describe an instantiation of an eventB development using JRA's instantiation context. We use terminal (as well non-terminal) recursive function and we recall some theorems on closure and recursion. Rodin [6] is used to develop and prove all models describe in this paper.

2 Theorems between Closure an Well-founded Relation

Let r be a relation ($r \in S \leftrightarrow S$) then its transitive closure is defined by a fixpoint and

- if r is well-founded then $closure(r)$ is also well-founded and $\forall x \cdot x \in S \Rightarrow x \notin closure(r^{-1})[\{x\}]$
- if r is well-founded and $r^{-1} \in S \nrightarrow S$ then $finite(closure(r^{-1})[\{x\}])$

3 Well-founded Relation and Fixpoint: Recursion

Recursive functions are defined with a well-founded relation and the fixpoint theorem.
- Let r be a well-founded relation on S: $r \in S \leftrightarrow S$
- Let g be a a function such that: $g \in (S \times (S \nrightarrow T)) \rightarrow T$
- There is a unique total function fr: $fr \in S \rightarrow T$

such that we have: $\forall x \cdot x \in S \;\Rightarrow\; fr(x) = g(x \mapsto r^{-1}[\{x\}] \lhd fr)$

- The value of fr at x depends on its value on the set $r^{-1}[\{x\}]$, $FrSB$ is a function (an operator) which gives the recursive fonction fr: $fr \;=\; FrSB(r \mapsto g)$

Many recursive functions have only one recursive call then $r^{-1}[\{x\}]$ is empty (base case) or a singleton then r^{-1} is a function. In this case we define the function (operator) $FrsB1$ where $FrSB1(r \mapsto f0 \mapsto f) = FrSB(r \mapsto g)$ and $g = \{x, h, b \cdot x \in S \wedge h \in S \nrightarrow B \wedge r^{-1}[\{x\}] \subseteq dom(h) \wedge (x \notin ran(r) \Rightarrow b = f0(x)) \wedge (x \in ran(r) \Rightarrow b = f(x \mapsto h(r^{-1}(x)))) \mid x \mapsto h \mapsto b\}$ in this case we have $\forall x \cdot x \in S \wedge x \notin ran(r) \;\Rightarrow\; fr(x) = f0(x)$ and $\forall x \cdot x \in S \wedge x \in ran(r) \;\Rightarrow\; fr(x) = f(x \mapsto fr(r^{-1}(x)))$

4 Terminal recursion

A function fr is terminal recursive if $fr = FrSB1(r \mapsto f0 \mapsto f)$ and $f(x \mapsto y) = y$ then we have $\forall x \cdot x \in S \wedge x \in ran(r) \;\Rightarrow\; fr(x) = fr(r^{-1}(x))$. The function (operator) $FrsB1Ter$ gives the function : $fr = FrSB1Ter(r \mapsto f0)$.

4.1 An abstract machine

Let fr equals $FrSB1Ter(r \mapsto f0)$ and $x \in S$, a variable R , an event which computes $fr(x)$ in one shot

$$\text{final} = \textbf{then } R := fr(x) \textbf{ end}$$

4.2 A refinement

Let y be a variable initialised to x with the invariant $fr(x) = fr(y)$

$$\text{final} = \textbf{when } y \notin ran(r) \textbf{ then } R := f0(y) \textbf{ end}$$

$$\text{progress} = \textbf{when } y \in ran(r) \textbf{ then } y := r^{-1}(y) \textbf{ end}$$

The variant is trivially $closure(r^{-1})[\{y\}]$ then progress cannot take the control forever.

4.3 An algorithm

Using JRA's merging rules [1] we obtain the following algorithm:

$$y := x;\ \textbf{while } y \in ran(r) \textbf{ do } y := r^{-1}(y) \textbf{ od};\ R := f0(y)$$

4.4 An example: gcd with mod

Xavier Leroy uses this gcd example in [3] and explains how well-founded relations are important in order to define recursive function. We can define $gcdmod$ with the following definition:
$gcdmod = FrSB1Ter(\{a, b \cdot b > 0 \wedge a > b | (b \mapsto a \bmod b) \mapsto (a \mapsto b)\} \mapsto (\lambda x \mapsto y \cdot x > y \wedge y \geq 0 \mid x))$

After proving that the relation is well-founded we got for free: $\forall a \cdot a > 0 \Rightarrow gcdmod(a \mapsto 0) = a$ and $\forall a, b \cdot a > b \wedge b > 0 \Rightarrow gcdmod(a \mapsto b) = gcdmod(b \mapsto (a \bmod b))$

5 Conclusion

If we correctly instantiate S, r and $f0$ in the corresponding context and if we prove the instantiation PO (r is well-founded and its inverse is a function) the instantiation of the algorithm gives for free the instantiated and correct algorithm.

We have similar schemata for non-terminal recursion (with or without stack) and sorted algorithms.

References

1. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010
2. D. Cansell, J.-R. Abrial: *Examples of using the Instantiation Plug-in"*, Rodin Workshop 2021
3. Xavier Leroy. *Well-founded recursion done right*. CoqPL 2024
4. G. Verdier and L. Voisin *Context instantiation plug-in: a new approach to genericity in Rodin.*, Rodin Workshop 2021
5. EBRP *Enhancing EventB and Rodin.* https://irit.fr/EBRP
6. *Rodin Platform.* http://www.event-b.org