From Event-B Models to Dafny Code Contracts

Mohammadsadegh Dalvandi, Michael Butler, Abdolbaghi Rezazadeh

From Event-B Models to Dafny Code Contracts

Mohammadsadegh Dalvandi, Michael Butler, Abdolbaghi Rezazadeh

Electronic and Computer Science School, University of Southampton
Southampton, United Kingdom
{md5g11,mjb,ra3}@ecs.soton.ac.uk

Abstract. The constructive approach to software correctness focuses on early phases of development and aims at formal modelling of the intended behaviour and structure of a system in different levels of abstraction and verification of its formal specification. In contrast, the analytical approach focuses on code level and its target is to verify the properties of the final program code. At a high level it would seem that the constructive and analytical approaches should complement each other well. In this paper, we present a development methodology by linking two existing verification methods, Event-B (constructive) and Dafny (analytical). This methodology benefits from Event-B abstraction and stepwise refinement for building Dafny contracts which is correct with regards to an Event-B high-level specification and then using Dafny for verifying the final implementation against the abstract specification. We outline the rules for transforming Event-B models to Dafny contracts and present a tool for automatic generation of Dafny contracts from Event-B formal models.

Keywords: Event-B, Dafny, Formal Methods, Program Verification, Methodologies

1 Introduction

Different formal approaches may be employed in various phases of software development cycle. We have identified and distinguished two major formal approaches to software correctness based on their target phases in development cycle: *constructive approach* and *analytical approach*. The constructive approach focuses on early stages of the development and aims at formal modelling of the intended behaviour and structure of a system in different levels of abstraction and verifying the formal specification of it. In contrast, the analytical approach focuses on code level and its target is to verify the properties of the final program code. In other words, the constructive approach is concerned with the derivation of an algorithm from the specifications of the desired dynamic behaviour of that, in a way that the algorithm satisfies its specification[9] while the analytical approach is concerned with verifying that if a given algorithm satisfies its given specifications. Both approaches are supported through a range of verification tools from groups worldwide. At a high level it would seem that the constructive and analytical approaches should complement each other well. However there is

little understanding or experience of how these approaches can be combined at a large scale and very little tool support for transitioning from constructive formal models to annotated code that is amenable to analytical verification. This represents a wasted opportunity, as deployments of the approaches are not benefiting from each other effectively.

In addition to what has been mentioned above, Clarke et al in [10], Hoare et al in [11], and Leavens et al in [12] stated that no single formal method can cover all aspects of a verification problem therefore engineering bridges between complementary verification tools to enable their effective interoperability may increase the verification capabilities of verification tools.

This paper presents a tool-supported development methodology by linking two existing verification tools, Rodin[2] and Dafny [7]. Rodin platform supports the creation and verification of Event-B formal models. Dafny tool is an extension to Microsoft Visual Studio for writing and verifying programs written in Dafny language. Event-B [1] is a formal modelling language which follows the constructive approach. It is based on set theory and first-order predicate logic and it is supported by the Rodin tool set. It supports abstraction and refinement and provides a way in which a software system can be gradually modelled and verified through a number of successive refinement levels. Dafny [7] is a programming language and a verifier which follows the analytical approach. Danfy programs can be specified formally with the help of its rich set of built-in specification constructs. Formally specified code can be verified automatically by Dafny tool which is a SMT-based verifier. Event-B in its original form does not have any support for the final phase of the development(implementation phase). On the other hand, Dafny has a very little support for abstraction and refinement. Our combined methodology is beneficial for both Event-B and Dafny users. It makes the abstraction and refinement of Event-B available for generating Dafny specifications which are correct with regards to a higher level of abstract specification in Event-B and allows Event-B models to be implemented and verified in a programming language.

We provide a set of transformation rules for transforming Event-B formal models to annotated Dafny method declarations. Our focus here is only on generating code contracts (pre- and post-conditions) from Event-B models rather than implementations. Generated method contract with this approach can be seen as an interface that can be implemented and verified later against the high level abstract specification. We also present a tool for automatic generation of Dafny annotations from Event-B models. We have validated our transformation rules with applying our tool to an Event-B model of a map abstract datatype which is presented in this paper.

The organisation of the rest of the paper is as follows: in section 2, background information on Event-B and Dafny is given. Section 3 contains an example of transformation of an Event-B model of a map abstract datatype to Dafny contracts. Transformation rules for transforming an Event-B machine to an annotated Dafny class are described in section 4. In section 5 related work is

presented. In section 6 future work are discussed and finally section 7 contains conclusions.

2 Background

2.1 Event-B

Event-B is a formal modelling language for system level modelling based on set theory and predicate logic for specifying, modelling and reasoning about systems, introduced by Abrial[1]. Modelling in Event-B is facilitated by a platform called Rodin[2, 3]. Rodin is an extensible open source software which is built on top of the Eclipse IDE. A model in Event-B consists of two main parts: *contexts* and *machines*. The static part(types and constants) of a model is placed in a context and the dynamic part (variables and events) is specified in a machine. To describe the static part of a model there are four elements in the structure of a context: *carrier sets*, *constants*, *axioms*, and *theorems*. Carrier sets are represented by their name and they are distinct from each other. Constants are defined using axioms. Axioms are predicates that express properties of sets and constants. Theorems in contexts can be proved from axioms. A machine in Event-B consists of three main elements: (1) a set of *variables*, which defines the states of a model (2) a set of *invariants*, which is a set of conditions on state variables that must hold permanently by all events and (3) a number of *events* which model the state change in the system. Each event may have a number of assignments called actions. Each event may also have a number of guards. Guards are predicates that describe the necessary conditions which should be true before an event can occur. Note that an event might not have any guards, in this case that event may always be enabled. An event may have a number of parameters. Event parameters are considered to be local to the event. Figure 1 illustrates machine *m0* with two events *Add* and *Remove*.

Modelling a complex system in Event-B can largely benefit from refinement. Refinement is a stepwise process of building a large system starting from an abstract level towards a concrete level[4, 5]. This is done by a series of successive steps in which, new details of functionality are added to the model in each step. The abstract level represents key features and the main purpose of the system. Abstract model does not care about details of the system and *how* the goal of the system is going to be achieved. Instead, it focuses on *what* is the goal of the system. It is important to prove the correctness of refinement steps in Event-B. Refining an Event-B model may involve context extension and machine refinement. When a context is extended, new sets, constants, and axioms are added and sets and constants in abstract context will be kept in the extension. Refinement of a machine may consist of refining existing events, adding new events, and adding new variables and invariants. Refining an existing abstract event may have one of the following two forms: (1) Extending the abstract event with new parameters, guards, and actions (*horizontal refinement*) or (2) Modifying parameters, guards, and actions of the abstract event (*vertical refinement*). In the former, abstract parameters, guards, and actions do not change. In the latter

Machine m0 **Sees** c0
Variables *map*
Invariants $map \in KEYS \pfun VALUES$
Initialisation $map := \varnothing$
Event *Add*
  any k,v
  where
    grd1: $k \in KEYS$
    grd2: $v \in VALUES$
  then
    act1: $map(k) := v$
Event *Remove*
  any k
  where
    grd1: $k \in dom(map)$
  then
    act1: $map := \{k\} \domsub map$

Fig. 1. Machine *m0*: the Most Abstract Level of Map ADT Model

however, replacing and adding new parameters, guards, and actions are allowed. In both cases guards of concrete event should be stronger than guards of abstract event. New events can be added in a refined machine. All new events refine a dummy event in abstract machine. This dummy event does nothing. The new events must not diverge. This means that they should not run for ever. Each refinement may involve introducing new variables to the model. This usually results in extending abstract events or adding new events to the model. It is also possible to replace abstract variables by newly defined concrete variables. Concrete variables are connected to abstract variables through *gluing invariants*. A gluing invariant associates the state of the concrete machine with that of its abstraction. All invariants of a concrete model including gluing invariants should preserve by all events. All abstract events may be refined by one or more concrete events.

The built-in mathematical language of the Rodin platform is limited to basic types and constructs like integers, boolean, relations and so on. The *Theory Plug-in*[6] has been developed to make the core language extension possible. A theory, which is a new kind of Event-B component, can be defined independently from a particular model and it is the mean by which the mathematical language and mechanical provers may be extended.

2.2 Dafny

Dafny[7] is an imperative sequential programming language which supports generic classes, dynamic allocation and inductive datatypes and has its own specification

constructs. A Dafny program may contain both specification and implementation details. Specifications and ghost variables are omitted by the compiler and are used just during the verification process. Programs written and specified in Dafny can be verified using Dafny verifier which is based on a SMT-solver called Z3[8]. Standard pre- and post-conditions, framing construct and termination metrics are included in the specifications. The language offers updatable ghost variables, recursive functions, sets, sequences and some other features to support specification. The verification power of Dafny originates from its annotations (contracts). A program behaviour can be annotated in Dafny using specification constructs such as methods' pre- and post-conditions. The verifier then tries to prove that the code behaviour satisfies its annotations. This approach leads to producing correct programs not only in terms of its syntax but also in terms of its behaviour. A basic program in Dafny consists of a number of methods. A method in Dafny is a piece of imperative, executable code. Danfy also supports *functions*. A function in Dafny is different from a method and has very similar concept to mathematical functions. A Dafny function cannot write to memory and consists of just one expression. Functions are required to have only one unnamed return value. A special form of functions which returns a boolean value is called *predicate*. Dafny uses the `ensures` keyword for post-condition declaration. A post-condition is always a boolean expression. Each method can have more than one post-condition which can either be joined with boolean *and* (&&) operator or be defined separately using the `ensures` keyword. To declare a pre-condition the `requires` keyword is used. Like post-conditions, multiple pre-conditions are allowed in the same style. Pre- and post-conditions are placed after method declarations and before method body. Dafny does not have any specific construct for specifying class invariants. Class invariants are specified in a predicate named *Valid()* and this predicate is incorporated in all methods pre- and post-conditions so the verifier checks if each method preserve all invariants or not. The Dafny verifier verifies each method in isolation and it does not reason about other methods by looking at their bodies. Dafny forgets about every other methods' body in the program except for the one which it is trying to prove. The only thing that Dafny knows about other methods in the program is their pre- and post-conditions. With this approach Dafny can reason about each method in isolation and this leads to simplification of verification.

3 Case Study: A Map Abstract Data Type

In this section we present a map abstract datatype as a case study and show the Event-B formal model and its transformation to a Dafny contract that is performed by our tool. A map (also called associated array) is an abstract data type which associates a collection of unique keys to a collection of values. This case study is originally taken from [13] where the map ADT is specified, implemented and verified in Dafny.

The most abstract model of the map in Event-B (machine *m0*) is illustrated in Figure 1. The map is simply modelled using a partial function from *KEYS* to

VALUES. *KEYS* and *VALUES* are generic types which are defined in context *c0*(Figure 3) as carrier sets. There is only one invariant in this model which says that the variable *map* is a partial function. The model contains two events for adding and removing keys and values to the map. By proving that these events are preserving the invariant of the model, the uniqueness of the map keys is verified.

Machine m1 **refines** m0 **Sees** c0
Variables *keys, values*
Invariants
inv1: keys∈ seq(KEYS)
inv2: values∈ seq(VALUES)
inv3: seqSize(keys) = seqSize(values)
inv4: ∀ i,j· i∈ 1.. seqSize(keys) ∧ j∈ 1.. seqSize(keys) ∧ i≠ j ⇒ keys(i)≠ keys(j)
g_inv1: seqSize(keys) = card(map)
g_inv2: map={ i· i∈ 1.. seqSize(keys)| keys(i)↦values(i)}
Initialisation *keys, values:= ∅*

Event *Add1* **refines** *Add*
any k,v
where
grd1: k∈KEYS
grd2: v∈VALUES
grd3: k∉ ran(keys)
then
act1: keys:= seqPrepend(keys,k)
act2: values:= seqPrepend(values,v)

Event *Add2* **refines** *Add*
any k,v,i
where
grd1: k∈KEYS
grd2: v∈VALUES
grd3: i∈ 1.. seqSize(keys)
grd4: keys(i)=k
then
act1: values(i):= v

Event *Remove* **refines** *Remove*
any k,i
where
grd1: k∈KEYS
grd2: i∈ 1.. seqSize(keys)
grd3: keys(i)=k
then
act1: keys:=seqSliceToN(keys,i-1) seqConcat seqSliceFromN(keys,i+1)
act2: values:=seqSliceToN(values,i-1) seqConcat seqSliceFromN(values,i+1)

Fig. 2. Machine *m1*: the First Level of Refinement

Dafny does not support relations (and functions) as data structures so we cannot directly transform machine *m0* to Dafny annotations. Machine *m0* should be refined in order to reduce the abstraction and syntax gap between the Event-B model and Dafny specification. Machine *m1* refines machine *m0* to decrease the

Fig. 3. Context *c0*: the Static Part of the Model

gap. In order to do this we use sequences to model the map. Sequences are built-in data structures in Dafny but they are not part of the built-in mathematical language of Rodin. However sequences are available through the standard library of theory plug-in. In this refinement the map is modelled by two sequences: *keys* and *values*. One sequence stores keys and the other stores values where a value in position i of sequence of values is associated with the key that is stored in position i of the sequence of keys. Figure 2 shows machine *m1*. In this machine, event *Add* is refined by two new events: *Add1*(for adding new keys to the map) and *Add2*(for updating an associated value to an existing key) and also event *Remove* is refined to remove a key and its associated value from the map. We have extended the standard theory of sequences by adding two new operators: *seqSliceToN* and *seqSliceFromN* to make removal of a value from a sequence possible.

Invariants *inv1* and *inv2* declare the type of variables *keys* and *values*. *inv3* states that the size of both sequences (*keys* and *values*) must always be equal and *inv4* states that all map keys are unique. Invariants *g_inv1* and *g_inv2* are gluing invariants. By proving that *g_inv1* and *g_inv2* hold for all events in *m1*, we can prove that *m1* is a correct refinement of the abstract machine *m0*.

Machine *m1* can be transformed to Dafny code. Events *Add1* and *Add2* will form a single method called *Add* and event *Remove* will be transformed to a method with the same name in Dafny. Listing 1.1 shows the transformation of events of machine *m1* to annotated Dafny method declarations.

```
method Add(k : KEYS,v : VALUES)
requires Valid();
ensures Valid();
ensures k !in old(keys) ==> keys==[k] + old(keys) && values
    ==[v] + old(values);
ensures forall i :: i in (set k0| 0<=k0 && k0<|old(keys)|) &&
     old(keys)[i]==k ==> values==old(values)[i:= v] && keys
    == old(keys);

method Remove(k : KEYS)
requires Valid();
ensures Valid();
ensures forall i :: i in (set k0| 0<=k0 && k0<|old(keys)|) &&
     old(keys)[i]==k ==> keys==old(keys)[..i] + old(keys)[i
    +1..] && values==old(values)[..i] + old(values)[i+1..];
```

Listing 1.1. Transformation of Machine *m1* to a Dafny Contract

As it was mentioned in 2.2, the predicate *Valid()* must be incorporated in all methods' pre- and post-condition as it contains class invariants which must be preserved by all methods. Other post-conditions of each method are directly driven from those events that form the method and they specify the behaviour of the method. Method *Add* is specified by two events in Event-B therefore two `ensures` clauses are generated. The reason for specifying a method with two Event-B events is that each event represents a separate case of the method and each case in a Dafny method is represented with a separate post-condition in the method contract. The keyword `old` which is used in the post-conditions of methods expresses that the variable is evaluated on entry to the method. Internal variables of each events are defined using universal quantifiers with regards to the event's guards. The class declaration, predicate *Valid()* and other details of the generated class are not shown here. The transformation of an Event-B machine to a Dafny class is discussed in the next section.

4 Transforming Event-B Models to Dafny Contracts

We outline the rules for transforming an Event-B machine to a Dafny class. The generated class consists of variable declarations, a predicate (*Valid()*) which contains a conjunction of all invariants except gluing invariants, an initialisation method, and a number of methods generated from events of the machine.

4.1 Transformation Rules

Generic Types Generic types in Event-B are defined using carrier sets. In Dafny generics can be declared in angle brackets after the name of the class or method. All carrier sets that are defined in the context that is seen by the machine that is being transformed will be translated as generics of the generated class.

Class Invariants When transforming a machine to a Dafny class, all invariants of that machine except gluing invariants must be translated to Dafny and placed in *Valid()* predicate. Gluing invariants are not included because they refer to variables of abstract Event-B models that do not become ghost variables in the Dafny. The predicate *Valid()* must be a pre- and post-condition for all methods in the Dafny class.

Constructor Statement Like any other object-oriented programming language, each method in Dafny is supposed to define a specific behaviour of a class. Each method may consist of a number of cases in the form of conditional statements to perform appropriate operation with regards to various conditions.

Due to this, all different behaviours of a method for different situations should be formally specified in Dafny in order to be able to verify the correctness of the behaviour of each method. Each event in Event-B can only model one specific behaviour with regards to one specific condition (its guards) therefore different cases of a single behaviour should be modelled in different events. Due to this, transforming an Event-B machine to a Dafny class, usually consists of merging a number of events to form a single method. Before the transformation can take place the events that should be merged together must be made explicit. To facilitate this, we have added a new element to Event-B machines called the *constructor statement*. Each constructor statement has the following form:

$$\texttt{method}\ mtd\_name(prm\_1,\ prm\_2,\ldots,\ prm\_n) = \{evt\_1,\ evt\_2,\ldots,\ evt\_n\}$$

In the left hand side of the above constructor statement, *mtd_name* is the name of the target method in Dafny. A method may have input parameters. Input parameters are listed after the method name in the constructor statement. Those Event-B events from which we are going to generate code contracts should be listed in the right hand side of the constructor statement. For each defined constructor statement one method will be generated in the Dafny class.

Method's Parameters A method may or may not have input parameters. Input parameters which are stated in the constructor statement must exist in all events which are listed in the right hand side of the statement and also the type of the parameters must be explicitly declared in Event-B events as guards of the event. If a method in right hand side of a constructor statement has a parameter which is not listed as a method's input parameter, it should be treated as an *internal parameter*. An internal parameter will be defined with the help of quantifiers.

Method's Contracts As it was mentioned, one or more events together may form a method. A number of post-conditions can be generated from *before-after* predicates of the actions of the forming events together with their guards. A before-after predicate denotes the relation that exists between the value of a variable just before and just after the execution of an action.

In the example shown in the previous section, the method *Add* was generated as the result of the following constructor statement:

$$\texttt{method}\ Add(\mathrm{k,v}) = \{Add1,\ Add2\}$$

Consider *act1* of event *Add1* from Figure 2. The before-after predicate associated with this action is as follows:

$$keys' = seqPrepend(keys, k)$$

In above predicate primed variables denote the value of the variable just after the execution and the unprimed variables denote the value of the variables before

the execution. The following post-condition can be derived from event *Add1* by using the conjunction of all non-typing guards of the event as the premises of the implication and conjunction of before-after predicates of all actions of the event as the consequence of the implication. The result implication expresses the behaviour of the event *Add1*:

$$k \notin ran(keys) \Rightarrow keys' = seqPrepend(keys, k) \wedge values' = seqPrepend(values, v)$$

The same should be done for event *Add2*. As is obvious from the action of event *Add2*, variable *keys* is not changed by this event therefore the value of this variable after the execution of the event is equal to its value before the execution. The following post-condition is derived from this event:

$$i \in 1..seqSize(keys) \wedge keys(i) = k \Rightarrow values'(i) = v \wedge keys' == keys$$

Note that event *Add2* has a third parameter i which is not listed as *Add* method parameter so it is an internal parameter and should be defined with the help of universal quantifier:

$$\forall i \cdot i \in 1..seqSize(keys) \wedge keys(i) = k \Rightarrow values'(i) = v \wedge keys' == keys$$

The above expressions can be translated to Dafny syntax and used as post-conditions for method *Add*(as it is shown in 1.1) for specifying the behaviours that are modelled by events *Add1* and *Add2* and should be implemented by method *Add*. In addition to generated post-conditions from events of the Event-B model, predicate *Valid()* must be added to pre- and post-condition of each method declaration.

4.2 Tool Support for Automatic Transformation

We have developed a Rodin plug-in for automatic transformation of Event-B machines to annotated Dafny classes. The plug-in builds an abstract syntax tree with regards to the Event-B machine and contexts that it sees and constructor statements that are provided by the user. The AST then is translated to Dafny code by a number of translation rules that are encoded in the plug-in source code. The tool only supports the translation of those Event-B mathematical constructs that have a counterpart in Dafny and ignores the rest. So it is important that the model should be refined to a level that only has those constructs that have a Dafny counterpart. Table 1 shows a number of Event-B constructs and their counterparts in Dafny. In this table P and Q denote predicates, E and F denote expressions, and z denotes a single or comma-separated list of variables.

5 Related Work

To the best of our knowledge, no research has been carried out in order to generate annotated Dafny programs from Event-B models and there is very little research on generating verifiable code from Event-B models.

	Event-B	**Dafny**
True	TRUE	true
False	FALSE	false
Conjunction	P ∧ Q	P && Q
Disjunction	P ∨ Q	P || Q
Implication	P ⇒ Q	P==>Q
Equivalence	P ⇔ Q	P<==>Q
Negation	¬P	!P
Universal Quantification	∀z.P⇒Q	forall z :: P ==>Q
Existential Quantification	∃z.P⇒Q	exists z :: P ==>Q
Equality	E = F	E == F
Inequality	E ≠ F	E != F

Table 1. Some Examples of Translation of Event-B Mathematical Constructs to Dafny Constructs

EventB2Dafny[14] is a Rodin plug-in for translating Event-B proof obligations to Dafny code to use Dafny verifier as an external theorem prover for proving Event-B proof obligations. Another research has been carried out in order to translate Even-B models to JML-specified[18] JAVA code. A Rodin plug-in called EventB2JML[15] has been developed to automate the translation from Event-B models to Java specified code. The plug-in provides some classes for specifying sets and relations in Java. Mery and Monahan in [16] proposed a transformation technique from an Event-B specification to an executable algorithm. In their approach the specification of the algorithm is provided at the start of the development in form of Spec#[17] pre- and post-conditions and the algorithm is modelled in Event-B with regards to those code contracts. At the end the generated code from the Event-B model is verified against the code contracts in Spec#. Tasking Event-B [19] is a code generator that generates code from Event-B models to a target language but it does not support verification of the generated code.

6 Future Work

Our current transformation rules allow us to generate Dafny contracts for abstract data types. We plan to extend our rules and the tool to be able to generate code contracts from Event-B model of complex algorithms. We have already done another case study for transforming an Event-B model of a model checking algorithm to Dafny contracts. In future, we want to link our contract generator to Tasking Event-B code generator to be able to generate both contracts and codes together automatically. We also plan to provide theories in theory plug-in for all Dafny constructs. We intend to extend the current tool to allow the definition of new user-defined translation rules for Event-B constructs.

7 Conclusion

We have presented an approach for generating Dafny code contracts from Event-B models. This approach allows us to start the development with a very high level specification of the program in Event-B and use the Rodin platform facilities to prove the correctness and consistency of specification and refine the specification to a level that is suitable for transformation to Dafny. The implementation can be done later manually and verified against the abstract specification. The abstraction level that can be achieved in a modelling language like Event-B is not achievable at Dafny level therefore using the stepwise manner of Event-B for building specification will help to tackle the complexity that is associated with this task. The meta-data that the user should provide in form of constructor statements will allow us to generate methods in Dafny which are useful in an object-oriented implementation rather than creating only one method per machine or one method per event in a model.

Acknowledgments. This work was funded in part by a Microsoft Research 2014 Software Engineering Innovation Foundation Award.

8 References

References

1. Abrial, J. R.: Modeling in Event-B: system and software engineering. Cambridge University Press (2010)
2. Abrial, J. R., Butler, M., Hallerstede, S., Hoang, T. S., Mehta, F., & Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. International journal on software tools for technology transfer, 12(6), 447-466. (2010)
3. Abrial, J. R., Butler, M., Hallerstede, S., & Voisin, L.: An open extensible tool environment for Event-B. In Formal Methods and Software Engineering (pp. 588-605). Springer Berlin Heidelberg(2006)
4. Butler, M.: Incremental design of distributed systems with Event-B. Engineering Methods and Tools for Software Safety and Security, 22, 131. (2009)
5. Butler, M.: Mastering System Analysis and Design through Abstraction and Refinement. (2012)
6. Butler, M., & Maamria, I.: Practical theory extension in Event-B. In Theories of Programming and Formal Methods (pp. 67-81). Springer Berlin Heidelberg. (2013)
7. Leino, K. R. M.: Dafny: An automatic program verifier for functional correctness. In Logic for Programming, Artificial Intelligence, and Reasoning (pp. 348-370). Springer Berlin Heidelberg. (2010)
8. De Moura, L., & Bjørner, N.: Z3: An efficient SMT solver. In Tools and Algorithms for the Construction and Analysis of Systems (pp. 337-340). Springer Berlin Heidelberg. (2008)
9. Dijkstra, E. W.: A constructive approach to the problem of program correctness. BIT Numerical Mathematics, 8(3), 174-186. (1968)
10. Clarke, E. M., & Wing, J. M.: Formal methods: State of the art and future directions. ACM Computing Surveys (CSUR), 28(4), 626-643. (1996)

11. Hoare, C. A. R., Misra, J., Leavens, G. T., & Shankar, N.: The verified software initiative: A manifesto. ACM Comput. Surv, 41(4), 1-8.(2009)
12. Leavens, G. T., Abrial, J. R., Batory, D., Butler, M., Coglio, A., Fisler, K., ... & Stump, A.: Roadmap for enhanced languages and methods to aid verification. In Proceedings of the 5th international conference on Generative programming and component engineering (pp. 221-236). ACM. (2006)
13. Leino, K. R. M., & Monahan, R.: Dafny meets the verification benchmarks challenge. In Verified Software: Theories, Tools, Experiments (pp. 112-126). Springer Berlin Heidelberg. (2010)
14. Catano, N., Leino, K. R. M., & Rivera, V.: The eventb2dafny rodin plug-in. In Developing Tools as Plug-ins (TOPI), 2012 2nd Workshop on (pp. 49-54). IEEE. (2012)
15. Catano, N., Rueda, C., & Wahls, T.: A Machine-Checked Proof for a Translation of Event-B Machines to JML. arXiv preprint arXiv 1309.2339. (2013)
16. Mry, D., & Monahan, R.: Transforming Event B Models into Verified C# Implementations. In VPT@ CAV (pp. 57-73). (2013)
17. Barnett, M., Leino, K. R. M., & Schulte, W.: The Spec# programming system: An overview. In Construction and analysis of safe, secure, and interoperable smart devices (pp. 49-69). Springer Berlin Heidelberg. (2005)
18. Leavens, G. T., Baker, A. L., & Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. ACM SIGSOFT Software Engineering Notes, 31(3), 1-38. (2006)
19. Edmunds, A., & Butler, M.: Tasking Event-B: An Extension to Event-B for Generating Concurrent Code. Programming Language Approaches to Concurrency and Communication-cEntric Software, 1. (2011)