A Floating-Point Numbers Theory for Event-B

Idir Ait-Sadoune(✉)

Paris-Saclay University, CentraleSupelec, LMF Laboratory Plateau de Saclay, Gif-Sur-Yvette, France
idir.aitsadoune@centralesuplec.fr

Abstract. Static type checking helps catch errors in manipulating variables values early on, and most specification languages, like Event-B, are strongly typed. However, the type system of Event-B language is relatively simple and provides only a way to specify discrete behaviour using *Integer* type. There is no possibility to model continuous behaviour, which would have helped analyse hybrid systems. More precisely, the Event-B language doesn't consider in its type-checking system the possibility of defining such behaviours and checking the correctness of the values of the continuous variables within the Event-B models. In this article, we propose to extend the type-checking system of Event-B to include *Float* variables by specifying a floating point numbers theory using the theory plugin.

Keywords: Hybrid systems · Event-B · Type checking · Floating-point numbers

1 Introduction

Since its invention, the use of the Event-B formal method [2] has continued to increase, and it has been applied to various applications and domains [6]. The Event-B method is practical and adapted to analyse discrete systems, and its type system offers the possibility of modelling discrete behaviours. Today, with the need to model and analyse hybrid systems to include different types of complex and cyber-physical systems, extending the Event-B type-checking systems becomes necessary to specify and analyse continuous behaviours or represent numerical algorithms in Event-B. This need involves considering the definition of real numbers, more concretely, floating-point numbers.

The interest and motivation for using the floating-point arithmetic in the case of the classical B method [1] was discussed in [12]. Today, with classical B language, it is possible to specify a treatment with real numbers and to ensure that its floating point implementation is "close" to its specification. In the case

This work was supported by a grant from the French national research agency ANR ANR-19-CE25-0010 (EBRP Project https://www.irit.fr/EBRP/).

M. Mosbah et al. (Eds.): MEDI 2023, LNCS 14396, pp. 30–43, 2024.
https://doi.org/10.1007/978-3-031-49333-1_3

of the Event-B method, we can use discretisation like in [5] and other works cited in the same paper to formalise continuous behaviours. Another known tool around the classical B and Event-B methods is the ProB model-checker [13]. Currently, *Leuschel et al.* work on integrating the floating-point arithmetic in ProB [14].

The Rodin platform [3] is the most used développement environment in the Event-B ecosystem. Since it is an Eclipse product, it can be extended by adding plugins. Among all the plugins developed for the Rodin platform, the theory plugin [7,10] is mainly used to extend the Event-B modelling possibilities by defining theories. The theory plugin provides the facility to define mathematical and prover extensions. Mathematical extensions are new operator definitions and new datatype definitions, and axiomatic definitions. In this article, we propose to develop a floating-point numbers theory using the theory plugin to extend the Event-B type-checking system with the possibility of handling floating-point numbers. We note that there is a theory for reals in the "Standard Library" of theories[1].

This paper is organized as follows: Sect. 2 presents the main concepts of the Event-B method. Section 3 gives an example to illustrate why there is a need to use floating-point arithmetic. Section 4 details the proposed approach, and Sect. 5 shows how the proposed theories improve the motivating example. The paper concludes with a summary and outlook in Sect. 6.

2 The Event-B Method

The Event-B method [2] is an evolution of the classical B method [1]. This method is based on the notions of pre-conditions and post-conditions [11], weakest pre-condition [9], and the calculus of substitution [1]. It is a formal method based on first-order logic and set theory.

2.1 The Event-B Model

An Event-B model is made of several components of two kinds: machines and contexts. The machines contain a model's dynamic parts (states and transitions), whereas the contexts contain the static parts (axiomatization and theories). A machine can be refined by another machine, and a context can be extended by another. Moreover, a machine can see one or several contexts (see Listings 1.1 and 1.2).

A context is defined by a set of clauses (see Listing 1.1) as follows:

- `SETS` describes a set of abstract and enumerated types.
- `CONSTANTS` represents the constants used by a model.
- `AXIOMS` describes, in first-order logic expressions, the properties of the attributes defined in the `CONSTANTS` clause. Types and constraints are described in this clause as well.

[1] https://sourceforge.net/projects/rodin-b-sharp/files/Theory_StdLib/.

– THEOREMS are logical expressions that can be deduced from the axioms.

An Event-B machine is defined by a set of variables, described in the VARIABLES clause, that evolves thanks to events depicted in the EVENTS clause. It encodes a state transition system where the variables represent the state, and the events represent the transitions from one state to another.

Listing 1.1. The Event-B context

```
CONTEXT ctx1
EXTENDS ctx2

SETS s
CONSTANTS c
AXIOMS
    A(s, c)
THEOREMS
    T(s, c)
END
```

Listing 1.2. The Event-B machine

```
MACHINE mch1
REFINES mch2
SEES ctxi

VARIABLES v
INVARIANTS
    I(s, c, v)
THEOREMS
    T(s, c, v)
EVENTS
    < events_list >
```

Similarly to contexts, a machine is defined by a set of clauses (see Listing 1.2). Briefly, the clauses mean.

– VARIABLES represents the state variables of the specification model.
– INVARIANTS describes, by first-order logic expressions, the properties of the variables defined in the VARIABLES clause. Typing information and functional and safety properties are usually expressed in this clause. These properties need to be preserved by events.
– THEOREMS defines a set of logical expressions that can be deduced from the invariants.
– EVENTS defines all the events that occur in a given model. Each event is characterized by its guard and the actions performed when the guard is true. Each machine must contain an "*Initialisation*" event.

The refinement operation offered by Event-B encodes model decomposition. A transition system is decomposed into another transition system with more and more design decisions while moving from an abstract level to a less abstract one. A refined machine is defined by adding new events, new state variables and a glueing invariant. Each event of the abstract model is refined in the concrete model by adding new information expressing how the new set of variables and the new events evolve.

2.2 The Proof Obligations (PO)

Proof obligations (PO) are associated with any Event-B model. They define the formal semantics associated with each Event-B component. PO are automatically generated, and *the PO generator plugin* in the Rodin platform [3] is in charge

of generating them. These PO need to be proved to ensure the correctness of developments and refinements. The obtained PO can be proved automatically or interactively by *the prover plugin* in the Rodin platform.

The rules for generating PO follow the substitutions calculus [1] close to the weakest precondition calculus [9]. To define some PO rules, we use the notations defined in Listings 1.1 and 1.2 where s denotes the seen sets, c the seen constants, and v the variables. Seen axioms are represented by $A(s, c)$ and theorems by $T(s, c)$, whereas invariants are denoted by $I(s, c, v)$ and local theorems by $T(s, c, v)$. For an event, the guard is denoted by $G(s, c, v, x)$ and the action is represented by the before-after predicate $BA(s, c, v, x, v')$ (a predicate expressing the relationship between the variable contents before and after an event triggering). Here we give a list of the most used/generated PO rules :

- *The theorem PO rule*: ensures that proposed theorems of a context or machine are provable.

$$A(s, c) \Rightarrow T(s, c)$$

$$A(s, c) \wedge I(s, c, v) \Rightarrow T(s, c, v)$$

- *Invariant preservation PO rule*: ensures that each event preserves each invariant in a machine.

$$A(s, c) \wedge I(s, c, v) \wedge G(s, c, v, x) \wedge BA(s, c, v, x, v') \Rightarrow I(s, c, v')$$

- *Feasibility PO rule*: ensures that a non-deterministic action is feasible.

$$A(s, c) \wedge I(s, c, v) \wedge G(s, c, v, x) \Rightarrow \exists v'.BA(s, c, v, x, v')$$

There are other rules for generating PO to prove the correctness of variables construction and using operators (Well definedness - WD) and refinement listed in [2].

2.3 The Theory Plugin

To extend the Event-B modelling possibilities with new mathematical objects, the theory plugin [7,10] extends the Rodin platform by providing a new syntax to define mathematical and prover extensions with the theory component. A theory can contain new datatype definitions, new polymorphic operator definitions, axiomatic definitions, theorems and associated rewrite and inference rules. The installation for the theory plug-in is available under the main Rodin Update site[2] under the category "Modelling Extensions". If you have never used the theory plugin, consult the user manual available in this link[3].

In this work, we use the theory Plugin to define the power operator (with its axiomatic definitions, theorems and inference rules) and the floating-point numbers data type (with all its operators, theorems and associated rewrite and inference rules). The justification of why we need these two theories will be given in the following sections.

[2] http://rodin-b-sharp.sourceforge.net/updates.
[3] https://wiki.event-b.org/images/Theory_Plugin.pdf.

3 The Motivating Example

To illustrate our approach for extending the Event-B core with the floating-point numbers data type, we propose to model a system that continuously calculates a moving object's speed (cf. listing 1.3). The main objective of this example is to show some modelling and validation problems that we can face when we analyse physical phenomena, mainly when we use integer variables to handle small values and expressions that come from the laws of physics. For simplicity reasons, we ignore in this study all the problems related to the units of measurement and which will be treated in our future work.

Listing 1.3. An Event-B model calculating a moving object's speed.

```
MACHINE mch_integer_version
...
INVARIANTS
    @inv1: traveled_distance ∈ ℕ
    @inv2: measured_time ∈ ℕ1
    @inv3: speed ∈ ℕ
    @inv4: starting_position ∈ ℕ
    @inv5: starting_time ∈ ℕ
    @inv6: speed = travelled_distance ÷ measured_time
    @inv7: traveled_distance > 0 ⇒ speed > 0

EVENTS
...
get_speed ≙
    any v t
    where
        @grd1: v ∈ ℕ1 ∧ v > starting_position
        @grd2: t ∈ ℕ1 ∧ t > starting_time
    then
        @act1: travelled_distance := v - starting_position
        @act2: measured_time := t - starting_time
        @act3: speed := (v - starting_position) ÷ (t - starting_time)
    end
END
```

The proposed Event-B model formalises two functional properties: **PROP 1 -** the speed of the moving object is equal to the *travelled_distance* divided by the *measured_time* ($v = d/t$), and **PROP 2 -** when the *travelled_distance* is strictly positive, the *speed* of the moving object must also be strictly positive (the object moves when its speed is different from zero). These two properties are formalised by the invariants @*inv*6 and @*inv*7 of listing 1.3.

The main event of the proposed Event-B model is called *get_speed*. It captures the new position of the moving object and calculates the new values of the *measured_time*, *travelled_distance*, and *speed* variables. These new values depend on the initial position stored in the *starting_time* and *starting_position* variables captured by another event that doesn't interest us in this study.

From the model validation point of view, we encountered a problem with the invariant preservation proof obligation of the @*inv*7 invariant generated for the *get_speed* event (cf. Fig. 1, all the OPs are green except the one maintaining the @*inv*7 invariant by the *get_speed* event). For recall, this invariant formalises the **PROP 2 -** property of our system (if the value of the *travelled_distance* variable is strictly positive, the *speed* variable must also be strictly positive). However, in the *get_speed* event, the value of the expression "$v - starting\_position$" can be less than that of "$t - starting\_time$". In this case, the new value of the *speed* variable becomes equal to zero while the one of the *travelled_distance* variable is not because all variables of our model are integer variables, and the "÷" Event-B operator makes an integer division. For these reasons, the `get_speed/inv7/INV` PO cannot be proved. Conceptually, our model correctly specifies our requirements; on the other hand, the basic types and operators of the Event-B language are not adapted to our needs and do not allow us to validate continuous behaviours requirements and manipulate small and big values simultaneously.

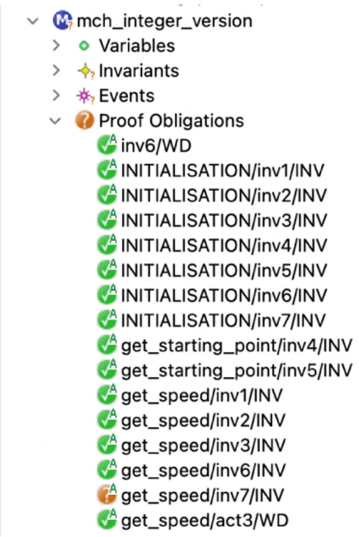

Fig. 1. The summary of the generated and proven (or not) proof obligations.

For these reasons and those discussed in [12], we propose to develop a floating-point numbers theory using the theory plugin to extend the Event-B type-checking system with the possibility of handling floating-point numbers.

4 The Proposed Approach

As known, the floating point is the most used method for representing and approximating real numbers in computer-based arithmetic. Therefore, we propose to represent real numbers by using floating-point arithmetic. This approach represents floating-point numbers using an integer called the **significand**, scaled by an integer **exponent** of a fixed **base**. We have chosen that the base always equals ten in our models (see the following example).

$$x = 3.14159265359 = \underbrace{314159265359}_{\text{significand}} \times \underbrace{10}_{\text{base}}^{\overbrace{-11}^{\text{exponent}}}$$

To allow the Event-B language to embed this floating-point representation, we need to define two theories: the first one formalises the power operator that isn't included in the Event-B language (the "^" caret Event-B operator is not

implemented in the automated proofs supported by the Rodin platform, besides power 0 and 1), and the second one formalises floating-point numbers by specifying the corresponding data type, the supported arithmetic operators, and some axioms and theorems that characterise the proposed modelling. The Event-B project containing these theories can be downloaded from this link[4].

4.1 The Power Operator

To have the possibility of comparing two floating point numbers, we need to define a left-shift operator that uses multiplication by powers of ten. In our case, for this reason, and for reasons we explain later, we define a power operator that uses only natural exponents. The power operation with natural exponents may be defined directly from multiplication operations. The definition of exponentiation as an iterated multiplication can be formalized using induction. The base case is $x^0 = 1$ and the recurrence is $x^n = x \times x^{n-1}$ (cf. Listing 1.4). For the case 0^0, in contexts where only natural powers are considered, 0 to the power 0 is undefined (see the `wd condition` defined for the `pow` operator in the Listing 1.4).

Listing 1.4. The theory defining the power operator

```
THEORY thy_power_operator

AXIOMATIC DEFINITIONS
    operators
    pow(x ∈ ℤ, n ∈ ℕ) : ℤ INFIX
        wd condition : ¬ (x = 0 ∧ n = 0)
    axioms
        @axm1: ∀ n. n ∈ ℕ1 ⇒ 0 pow n = 0
        @axm2: ∀ x. x ∈ ℤ ∧ x≠0 ⇒ x pow 0 = 1
        @axm3: ∀ x,n. x ∈ ℤ ∧ x≠0 ∧ n ∈ ℕ1 ⇒ x pow n = x × (x pow (n-1))
        ...
THEOREMS
    @thm1: ∀ x,n,m. ... ⇒ (x pow n) × (x pow m) = x pow (n+m)
    @thm2: ∀ x,n,m. ... ⇒ (x pow n) pow m = x pow (n×m)
    @thm3: ∀ x,y,n. ... ⇒ (x×y) pow n = (x pow n)×(y pow n)
    ...
END
```

The proposed theory also contains some proven theorems formalising some exponent rules (the product rule, the power rule, the multiplying exponents rule, ...). The proofs of all these theorems were made by induction following the rules defined in [8]. Notice that we have chosen to define the `pow` operator in a single theory to offer the possibility of reusing this operator in other Event-B components (theories, machines, or contexts) using the theory path mechanism available in the theory plugin [7,10].

4.2 The Floating-Point Numbers Theory

The proposed theory formalizes a floating-point number by defining a new data type called `FLOAT_Type`. This new data type provides the `NEW_FLOAT` constructor

[4] https://www.idiraitsadoune.com/recherche/modeles/eventb.theories.zip.

that allows creating a floating-point number by following the definition $x = s \times 10^e$ (with s representing the significand part and e the exponent part). This way, it is possible to create constants like 0 and 1 ($F0 = 0 \times 10^0$ or $F1 = 1 \times 10^0$) (cf. Listing 1.5). The proposed theory does not model limited precision floating point numbers. This implies that the operators defined in the theory involve no precision loss. This choice is made in order to allow the user to refine the proposed theory towards any implementation, the IEEE Standard 754, for example. This also allows us to remain compliant with the definition of the Event-B integer type, which is independent of any implementation. This theory provides an operator (`FLOAT`) to convert any Event-B `integer` to `FLOAT_Type`. We consider that the abstract numbers are those defined in the Event-B theory, and the concrete ones are those described by the IEEE Standard.

Listing 1.5. The theory defining the floating-point numbers (part 1)

```
THEORY thy_floating_point_numbers

DATATYPES
    FLOAT_Type ≙ NEW_FLOAT(s ∈ ℤ, e ∈ ℤ)

OPERATORS
    F0 ≙ NEW_FLOAT(0,0)
    F1 ≙ NEW_FLOAT(1,0)
    FLOAT(x ∈ ℤ) ≙ NEW_FLOAT(x,0)

    l_shift(x ∈ FLOAT_Type, offset ∈ ℕ) ≙
        NEW_FLOAT(s(x) × (10 pow offset), e(x)-offset)

    eq(x ∈ FLOAT_Type, y ∈ FLOAT_Type) INFIX ≙
        s(l_shift(x, e(x)-min({e(x),e(y)}))) =
        s(l_shift(y, e(y)-min({e(x),e(y)})))

    gt(x ∈ FLOAT_Type, y ∈ FLOAT_Type) INFIX ≙
        s(l_shift(x, e(x)-min({e(x),e(y)}))) >
        s(l_shift(y, e(y)-min({e(x),e(y)})))

    geq(x ∈ FLOAT_Type, y ∈ FLOAT_Type) INFIX ≙
        x eq y ∨ x gt y

    lt(x ∈ FLOAT_Type, y ∈ FLOAT_Type) INFIX ≙
        ¬(x geq y)

    leq(x ∈ FLOAT_Type, y ∈ FLOAT_Type) INFIX ≙
        ¬(x gt y)
    ...
END
```

The floating-point theory redefines all essential numeric operators (comparison and calculation operators), and the operator we have to define to overload all

the numeric operators is the left shift operator (`l_shift` operator). This operator uses a positive offset to perform a left shift by multiplying the significand part by powers of ten[5]. To compare two numbers, we left-shift the number containing the biggest exponent to have the same exponent as the other number. Then, when two numbers have the same exponents, it's possible to compare them by comparing their significand parts. In this way, we have defined the operators `eq`, `gt`, `geq`, `lt`, and `leq` ($=, >, \geq, <, \leq$) comparing two floating-point numbers (cf. Listing 1.5).

Listing 1.6. The theory defining the floating-point numbers (part 2)

```
THEORY thy_floating_point_numbers
...
OPERATORS
    ...
    plus(x ∈ FLOAT_Type, y ∈ FLOAT_Type) INFIX ≙
        NEW_FLOAT(s(l_shift(x, e(x)-min({e(x),e(y)}))) +
        s(l_shift(y, e(y)-min({e(x),e(y)}))) , min({e(x),e(y)}))

    neg(x ∈ FLOAT_Type) ≙
        NEW_FLOAT(−1 × s(x), e(x))

    minus(x ∈ FLOAT_Type, y ∈ FLOAT_Type) INFIX ≙
        x plus neg(y)

    mult(x ∈ FLOAT_Type, y ∈ FLOAT_Type) INFIX ≙
        NEW_FLOAT(s(x) × s(y) , e(x) + e(y))

    f_pow(x ∈ FLOAT_Type, n ∈ ℕ) INFIX ≙
        NEW_FLOAT(s(x) pow n, n × e(x))
    ...
END
```

Using the same reasoning for the comparison, we have generalized the idea to the addition and subtraction operators. A left-shift of one of the two operands is necessary to perform the addition and subtraction operations (cf. Listing 1.6). However, the multiplication operation is performed by multiplying the significand parts of the two operands, and the resulting exponent is obtained by adding the exponent parts of the two operands (cf. Listing 1.6). The `f_pow` operator generalises the `pow` operator for the floating-point numbers.

While the proposed theory involves no precision loss for multiplication and addition, division sometimes induces a precision loss. For example, we cannot precisely represent the result of 1/3 or 2/3. That is why, for the case of the division and inverse operators, we have firstly defined the well-defined conditions (by the `inv_WD` and `div_WD` operators in listing 1.7). To calculate the inverse of x, we must find a z, which we multiply by the significand part of x to obtain a power of ten (The value of z corresponds to the significand part of the result of the inverse of x) . For example, to calculate the inverse of 2, 5 corresponds to

[5] This is why we have defined a power operator with only natural exponents.

z in our case, which does not exist for the inverse of 3. The same reasoning is done for the division operator.

Listing 1.7. The theory defining the floating-point numbers (part 3)

```
THEORY thy_floating_point_numbers
...
OPERATORS
    ...
    inv_WD(a ∈ FLOAT1_Type) ≙
        ∃ n,z. n ∈ ℕ ∧ z ∈ ℤ ∧ 10 pow n = s(a) × z

    div_WD(a ∈ FLOAT_Type, b ∈ FLOAT1_Type) ≙
        ∃ n,z. n ∈ ℕ ∧ z ∈ ℤ ∧ s(a) × (10 pow n) = s(b) × z

AXIOMATIC DEFINITIONS
    operators
    inv(x ∈ FLOAT_Type) : FLOAT1_Type
        wd condition : inv_WD(x)
    axioms
        @inv_1: ∀ x,y.(... ⇒ ((x mult y) = F1 ⇔ inv(x) = y))
        @inv_2: ∀ x,y.(... ⇒ ((x mult y) eq F1 ⇔ inv(x) eq y))

    operators
    div(x ∈ FLOAT_Type, y ∈ FLOAT_Type) : FLOAT_Type INFIX
        wd condition : div_WD(x,y)
    axioms
        @div_1: ∀ x,y,z.(... ⇒ ((y mult z) = x ⇔ (x div y) = z))
        @div_2: ∀ x,y,z.(... ⇒ ((y mult z) eq x ⇔ (x div y) eq z))
        @div_3: ∀ x,y.(... ⇒ x mult inv(y) = x div y)
    ...
END
```

The last basic arithmetic operations, inverse and division, are formalized by axiomatic definitions, and both are invocable if their well-defined conditions are true (defined in the `wd condition` clause). The inverse of x is y if and only if y is the number we multiply by x to obtain $F1$, and the result of dividing x by y is z, if and only if z is the number we multiply by y to obtain x (cf. Listing 1.7). We must prove the `WD PO` generated from the `wd condition` for both operators. The axiom @div_3 gives the relationship between the inverse operator and the division operator.

Finally, the floating-point data type is often used in laws of physics and scientific calculations. Functions calculating the integer part, the fractional part, the floor function and the ceiling function are very useful. This theory provides all these operators, and due to the page number limitations, these operators are not presented in this article. The reader may consult them by downloading this theory from this link[6].

The last part of the proposed theory contains a set of theorems that we have proved, and that correspond to laws defining properties of arithmetic operators

[6] https://www.idiraitsadoune.com/recherche/modeles/eventb.theories.zip.

(equality, addition, and multiplication are commutative, the order is total, reflexive, anti-symmetric and transitive, addition and multiplication have an inverse, ...) and others theorems combining the comparison operators and the arithmetic operators (cf. Listing 1.8).

Listing 1.8. The theory defining the floating-point numbers (part 4)

```
THEORY thy_floating_point_numbers
...
THEOREMS
    @thm1: ∀ x,y.(... ⇒ x eq y ⇔ y eq x)
    @thm2: ∀ x.(... ⇒ x geq x ∧ x leq x)
    @thm3: ∀ x,y.(... x leq y ∧ y leq x ⇒ x eq y)
    @thm4: ∀ x,y.(... ⇒ x leq y ∨ y leq x)
    @thm5: ∀ x,y,z.(... x leq y ∧ y leq z ⇒ x leq z)
    @thm6: ∀ x,y,z.(... x leq y ⇒ (x plus z) leq (y plus z))
    @thm7: ∀ x,y,z.(... x leq y ⇒ (x mult z) leq (y mult z))

    @thm8: ∀ x.(... ⇒ x plus F0 eq x)
    @thm9: ∀ x,y.(... ⇒ x plus y = y plus x)
    @thm10: ∀ x,y.(... ⇒ x plus neg(y) = y minus x)
    @thm11: ∀ x.(... ⇒ x minus F0 eq x)
    @thm12: ∀ x.(... ⇒ x minus x eq F0)

    @thm13: ∀ x.(... ⇒ x mult F0 eq F0)
    @thm14: ∀ x.(... ⇒ x mult F1 = x)
    @thm15: ∀ x,y.(... ⇒ x mult y = y mult x)

    @thm16: ∀ x.(... ⇒ inv(x) = F1 div x)
    @thm17: ∀ x.(... ⇒ x div F1 = x)
    @thm18: ∀ x.(... ⇒ x div x = F1)
    @thm19: ∀ x.(... ⇒ x mult inv(x) = F1)
    ...
END
```

Due to our choice to formalise unlimited precision floating-point numbers (the operators defined in the proposed theory involve no precision loss), we can deduce some properties that are not true in the floating-point numbers world (the associativity of addition and multiplication, for example). When this theory is refined towards any implementation (the IEEE Standard 754, for example), the developer must pay attention to this point.

5 Revisiting the Motivating Example

The example presented in Sect. 3 is updated to use the floating-point numbers theory. All `NATURAL` variables are typed by `PFLOAT_Type` set containing positive

floating-point numbers (cf. Listing 1.9), and the rest of the model was adapted using the equivalent operators from the proposed theory. The obtained Event-B machine contains almost the same invariants and the same events (cf. Listing 1.10). The only difference is the addition of the invariant @*inv*6 concerning the well-defined condition of the division operator used to formalise the speed of the moving object (**PROP 1** of the motivating example). Thus, **PROP 1** and **PROP 2** of the initial model are formalised by the invariants @*inv*7 and @*inv*8 (cf. Listing 1.10).

Listing 1.9. The definition of the positive floating-point numbers

```
THEORY thy_floating_point_numbers
    ...
    PFLOAT_Type = { x · x ∈ FLOAT_Type ∧ s(x) ≥ 0 | x }
END
```

Listing 1.10. The new version of the model calculating the speed of a moving object

```
MACHINE mch_floating_point_version
...
INVARIANTS
    @inv1: traveled_distance ∈ PFLOAT_Type
    @inv2: measured_time ∈ PFLOAT_Type ∧ s(measured_time) ≠ 0
    @inv3: speed ∈ PFLOAT_Type
    @inv4: starting_position ∈ PFLOAT_Type
    @inv5: starting_time ∈ PFLOAT_Type
    @inv6: div_WD(traveled_distance, measured_time)
    @inv7: speed eq traveled_distance div measured_time
    @inv8: traveled_distance gt F0 ⇒ speed gt F0

EVENTS
...
get_speed ≙
    any v t
    where
        @grd1: v ∈ PFLOAT_Type ∧ v gt starting_position
        @grd2: t ∈ PFLOAT_Type ∧ t gt starting_time
        @grd3: div_WD(v minus starting_position, t minus starting_time)
    then
        @act1: traveled_distance := v minus starting_position
        @act2: measured_time := t minus starting_time
        @act3: speed := (v minus starting_position) div (t minus starting_time)
    end
END
```

From the model validation point of view, contrary to the initial model, all generated proof obligations have been proven. The problem with the invariant linked to the integer division operator no longer arises. As shown in the Fig. 2, the `get_speed/inv8/INV` PO becomes green, and it has been proven using the interactive prover of the Rodin platform. As we have said, the @*inv*8 formalises the following property : if the value of the *travelled_distance* variable is strictly positive, the *speed* variable must also be strictly positive. Even if in the *get_speed* event, the value of the expression "*v minus starting_position*" can be less than that of "*t minus starting_time*", the new value of the *speed* variable is never equal to zero because the value of "*v minus starting_position*" is also never equal to zero (thanks to the guard @*grd*1 of the *get_speed* event). All this is possible thanks to the new `div` operator specification, which acts on the floating-point numbers.

Fig. 2. The summary of the generated and proven POs of the new Event-B machine.

This is one of the reasons that allow us to conclude that our floating-point numbers theory is more suitable than the basic integers of Event-B in modelling hybrid systems and continuous behaviours.

6 Conclusion

In this article, we have proposed an approach using the theory plugin to extend the Event-B type-checking system with the possibility of handling floating-point numbers. We have developed a floating-point numbers theory that formalises a floating-point number using an integer called the **significand**, scaled by an integer **exponent** of a fixed **base** (equals ten in our theory). Our proposition includes an extension of the Event-B power operator to handle powers of ten more than 0 and 1. We have proposed an abstract representation of the floating-point numbers to offer the possibility to refine the proposed theory to any more concrete implementation (the IEEE standard, for example).

For the next step of our work, we consider the floating-point numbers theory as the first step before developing a more general theory that will formalise the standard units of measurement defined by the International System of Units (SI). Such theories will be helpful in modelling cyber-physical, and these works will be integrated into our framework [4] for generating the Event-B model from ontologies that can define concepts in the context of hybrid systems.

References

1. Abrial, J.: The B-book - assigning programs to meanings. Cambridge University Press, Cambridge (1996). https://doi.org/10.1017/CBO9780511624162
2. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press, Cambridge (2010). https://doi.org/10.1017/CBO9781139195881
3. Abrial, J., Butler, M.J., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. Int. J. Softw. Tools Technol. Transf. **12**(6), 447–466 (2010). https://doi.org/10.1007/s10009-010-0145-y
4. Ait-Sadoune, I., Mohand-Oussaid, L.: Building formal semantic domain model: an Event-B based approach. In: Schewe, K.-D., Singh, N.K. (eds.) MEDI 2019. LNCS, vol. 11815, pp. 140–155. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32065-2_10
5. Babin, G., Aït-Ameur, Y., Singh, N.K., Pantel, M.: Handling continuous functions in hybrid systems reconfigurations: a formal Event-B development. In: Butler, M., Schewe, K.-D., Mashkoor, A., Biro, M. (eds.) ABZ 2016. LNCS, vol. 9675, pp. 290–296. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33600-8_23
6. Butler, M.: The first twenty-five years of industrial use of the B-method. In: ter Beek, M.H., Ničković, D. (eds.) FMICS 2020. LNCS, vol. 12327, pp. 189–209. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58298-2_8
7. Butler, M., Maamria, I.: Practical theory extension in Event-B. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) Theories of Programming and Formal Methods. LNCS, vol. 8051, pp. 67–81. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39698-4_5
8. Cervelle, J., Gervais, F.: Introducing inductive construction in B with the theory plugin. In: Glässer, U., Creissac Campos, J., Méry, D., Palanque, P. (eds.) ABZ 2023. LNCS, vol. 14010, pp. 43–58. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-33163-3_4
9. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Hoboken (1976)
10. Hoang, T.S., Voisin, L., Salehi, A., Butler, M.J., Wilkinson, T., Beauger, N.: Theory Plug-in for Rodin 3.x. CoRR abs/1701.08625 (2017). http://arxiv.org/abs/1701.08625
11. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969). https://doi.org/10.1145/363235.363259
12. Lecomte, T., Burdy, L., Dufour, J.L.: The B method takes up floating-point numbers. In: Embedded Real Time Software and Systems (ERTS2012) (2012)
13. Leuschel, M., Butler, M.: ProB: a model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45236-2_46
14. Rutenkolk, K.: Extending modelchecking with ProB to floating-point numbers and hybrid systems. In: Glässer, U., Creissac Campos, J., Méry, D., Palanque, P. (eds.) ABZ 2023. LNCS, vol. 14010, pp. 366–370. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-33163-3_27