Noname manuscript No.
(will be inserted by the editor)

Inferring Physical Units in Formal Models

Sebastian Krings · Michael Leuschel

Received: date / Accepted: date

Abstract Most state-based formal methods, like B, Event-B or Z, provide support for static typing. However, these methods and the associated tools lack support for annotating variables with (physical) units of measurement. There is thus no obvious way to reason about correct or incorrect usage of such units. We present a technique that analyses the usage of physical units throughout B and Event-B machines, infers missing units and notifies the user of incorrectly handled units. The technique combines abstract interpretation with classical animation, constraint solving and model checking and has been integrated into the ProB validation tool, both for classical B and for Event-B. It provides source-level feedback about errors detected in the models. We also describe how to extend our approach to TLA$^+$, an untyped formal language. We provide an in-depth empirical evaluation and demonstrate that our technique scales up to real-life industrial models.

Keywords B-Method · Event-B · Physical Units · Model Checking · Abstract Interpretation

1 Introduction and Motivation

Static type checking is generally[1] considered to be very useful to catch obvious errors early on and most specification languages are strongly typed. In particular, the B language [1] and its successor Event-B [2] are strongly typed. However, their type systems are relatively simple. In particular, there is no way to subtype the integers: a variable holding natural numbers and a variable holding a negative integer have the same type: `INTEGER`. Moreover, there is no way to specify physical units for integers, which are useful to avoid illegal manipulations, such as adding a speed value to a time value. For safety critical systems such a static check would be highly desirable, but currently there is no obvious way to enforce correctness of physical unit manipulations within B models.

This article is the extended version of the conference paper [23], where we proposed a solution to this problem, by integrating an abstract interpretation technique into the ProB animator [26,27]. More precisely:

- We provided an abstract semantics for B, in which integers are represented by their physical units;
- The abstract semantics can be simulated using the ProB toolset, by switching from the concrete mode to the abstract mode;
- We can run ProB in abstract mode until a fixpoint is reached;
- The result (abstract values computed for variables, parameters, ...) of the fixpoint is analyzed and translated into source-level user feedback.

The technique has been implemented both for B and Event-B, and applied to several industrial safety critical models. For this article we also extend our former work [23] in different aspects:

- We combine the technique with a translation from TLA$^+$ to B [18]. This allows us to validate the usage of physical units throughout TLA$^+$ models. See Sect. 3.3 for details.
- We expand our unit analysis for B / Event-B with refinement. This allows units distributed over sev-

Part of this research has been sponsored by the EU funded FP7 project 287563 (ADVANCE) and the DFG funded research project GEPAVAS II.

Institut für Informatik, Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
Tel.: +49 (211) 81-12635, Fax: +49 (211) 81-10712
krings@cs.uni-duesseldorf.de, leuschel@cs.uni-duesseldorf.de

[1] See, however, [25].

eral refinement levels of a B machine, and allows units themselves to be refined. Details can be found in Sect. 6.
- We made it possible to add user-defined units to our analysis.
- For the evaluation, we added several newly crafted benchmarks and case studies from industrial partners.
- Furthermore, we added an in-depth explanation of unit conversions of both SI and non-SI units. It can be found in Sect. 4.3.

An introductory example can be found in Fig. 1. It contains an extract of a simple B machine modeling the movements of a car.[2] The current speed and position are stored in two variables and updated at every tick of a clock. The duration of one tick is defined by a constant. Implicitly, the speed is measured in meters per second, the position in meters from a starting point and the length of a tick is defined in seconds. However, when updating the car's position in the `keep_speed` operation, a multiplication of the speed with the `tick_length` is missing. While this does not lead to an invariant violation or B type error, it leads to wrong results for the position of the car if `tick_length` is different from 1.

Analyzing the physical units of measurement, the error is easy to detect. When given the units of `speed` (meters per second) and `tick_length` (seconds), we see that the position should be in meters. Furthermore, we see that adding `position` (meters) to `speed` (meters per second) does not result in a well-formed unit of measurement. Hence, the missing multiplication is detected. From the perspective of the B typing rules, however, the addition is correct: we add two integers.

```
MACHINE Car
CONSTANTS tick_length
PROPERTIES tick_length = 2
VARIABLES speed, position
INVARIANT speed : INT & position : INT
INITIALISATION speed,position := 0,0
OPERATIONS
 keep_speed =
  PRE position + speed * tick_length : INT
  THEN position := position + speed END
 ...
END
```

Fig. 1 Introductory Example

Below, in Sect. 2 we will give a short introduction to B, Event-B and TLA$^+$ in order to make our examples accessible to readers without prior knowledge of these languages. Typing in the B languages will be discussed in Sect. 3.1. Further on, in Sect. 3.2 we will discuss how the syntax of the B language was extended in order to be able to declare physical units and reason about them. Section 3.3 will explain how we embedded physical units into TLA$^+$. We will mainly use the international system of units (SI) [36], but a user can also declare additional non-SI units.

Afterwards, we explain how we use abstract interpretation to analyze physical units in Sect. 4.1. In Sect. 4.2 we explain how we deal with function calls that are parametric in the units. Unit conversions are treated in details in Sect. 4.3. Section 5 will explain why we had to incorporate constraint solving techniques into our abstract interpreter.

In addition to unit analysis for a single machine, we provide a refinement chain from abstract to concrete units in Sect. 6.

Empirical results will be presented in Sect. 7. We conclude with alternative approaches and related work in Sect. 8 and a discussion of our results and future work in Sect. 9.

2 Introduction to B, Event-B and TLA$^+$

B [1] and its successor Event-B [2] are formal languages for the specification of software and systems. The basic structure is that of an abstract `MACHINE` that can be refined further into more concrete machines. In B, the the user can introduce `CONSTANTS` which have to fulfill certain `PROPERTIES` inside a `MACHINE`. For Event-B, the static part of a model is defined inside of a context file. In addition to constants, the `VARIABLES` clause inside of a machine holds the state variables of the model. An initial value for them can be assigned inside the `INITIALISATION`. Furthermore, an `INVARIANT` can be given to specify valid and invalid states. The types of constants and variables are given together with the properties and invariants. As B is strongly typed, every variable or constants has to be typed either explicitly or by assigning it a value that can be typed.

Possible transitions from one state to another are defined by the `OPERATIONS` clause. Inside, the user can provide generalized substitutions (aka. statements) that change the content of state variables.

In the examples, we will use two different substitutions:

- The simple substitution, which is enclosed in `BEGIN` and `END`. It can be executed anytime.

[2] We return to this example in Sect. 7.1 and provide further details there.

– The substitution with precondition `BEGIN` p `THEN` s `END`, where the assignments in s can only be executed if p evaluates to true.

An operation may have a return value, indicated by `<--` separating the variable to return from the operation body.

A substitution may hold one or more assignments to the state variables that are executed in parallel. The assignments are written as `variable := expression`. For the sake of simplicity we will only use expressions using basic arithmetic. However, B and Event-B support set theory, higher order functions and datatypes as well as quantification.

In contrast to B and Event-B, TLA$^+$ [24] does not rely on substitutions to specify state space transitions. Instead, possible transitions are given in form of a before-after-predicate. Inside, primed versions of the variable identifiers refer to the variables of the next state. A simple TLA$^+$ specification then consists of a predicate describing the initial states together with the before-after-predicate. We will describe TLA$^+$ in greater details once we introduced an example specification.

3 Embedding Units into Existing Modeling Languages

3.1 Typing in the B Language

The B language [1] provides the following basic types:

– integers,
– booleans,
– user-defined types, with are either enumerated sets, abstract deferred sets, or machine parameters, and
– strings.

These types can be combined using the Cartesian product constructor to generate pairs, and the power set constructor to generate sets. Note that functions and relations in B are represented as sets of pairs. The type of the number 1 is thus the set of integers (`INTEGER` in B syntax), and the type of the function $f = \{1 \mapsto 1, 2 \mapsto 4, 3 \mapsto 9\}$ is the set of all sets of integer pairs (`POW(INTEGER*INTEGER)` in B syntax)[3]. We will return to this later in Sect. 4.1.1.

Atelier-B also provides a record constructor for B, which we will ignore in this paper (but our analysis and tool support it). Atelier-B 4.1 and later provide limited support for reals and floats in the B language [11].

The Event-B [2] language has the same underlying type system than classical B. However, Event-B provides mathematical extensions, which allow adding new types such as records, reals or inductive data types. Again, our analysis and tool supports these, but to avoid distracting from the important issues we will not present these types in the present article.

3.2 Unit Declarations in the B Language

At some point the user must provide the physical units for certain variables as a starting point of our analysis. As already mentioned, the B language does not provide a mechanism for sub-typing the integers. As such, we need a mechanism "outside" of the core B language to declare the units of integer variables. For Event-B, this has been achieved by attaching new attributes to variables in the Rodin database [3]. In classical B, this association must be described within the B ASCII syntax. We wanted to ensure that a B machine making use of the new syntax is still usable by other tools (such as Atelier-B). This requirement ruled out an extension involving keywords or constructs which are not part of the standard B language and could therefore not be parsed by tools other than ProB. Instead, we decided to implement the new functionality inside semantically relevant comments, i.e., *pragmas*. While the usual B block comment is enclosed in `/*` and `*/`, a pragma is enclosed in `/*@` and `*/`. Other tools like Atelier-B will treat such a pragma as an ordinary comment.

For our work we have introduced five pragmas to the B language:

1. "`unit`", the pragma used to attach a physical unit to certain B constructs. This can be done by specifying the unit either using a B expression in an SI-compatible form or using a predefined alias like "cm" instead of "10**-2 * m". The given unit has to be a valid SI unit [36]. A derived unit such as "m * s**-2" is thus acceptable. The usage is shown in Fig. 2.
2. "`inferred_unit`", which works similar to `unit`. It is automatically added to the pretty printed version of a machine, attaching units inferred by ProB to variables and constants. This enables the user to generate a model containing the information gathered by our analysis.
3. "`conversion`", used to annotate operations meant as conversions between units. An example can be found in Fig. 3.

[3] Note that this set includes not just functions but all relations between integers. In B and ProB functions and relations have the same type; operators for relations can be applied to functions and vice versa. The fact that a relation is indeed a function is encoded as an invariant and verified for each state, i.e. it is a safety property of the system.

4. "`unit_alias`", used to define new aliases for existing unit definitions. Figure 4 shows how to define the new alias "mps" for meter per second. Once an alias has been defined, it can be used to define further aliases.
5. "`new_unit`", used to define entirely new units for domain specific applications. Figure 5 shows how the pragma can be used to define a new unit called "yard". See the case study in Sect. 7.5 for an extended example. Sadly the new units are somewhat limited, as it is currently impossible to specify conversion rules, arithmetic or special behavior. A user-defined unit works simply as a new base unit and is usable in conjunction with the existing ones.

Please note that the `unit` pragma can only be attached to certain constructs. Among others, these are declarations of variables and constants as well as integer literals. It is not possible to annotate relations or functions directly, i.e. by a single pragma containing the unit of both the range and the domain. However, they can be annotated on their own by introducing intermediate variables for them. For example, annotating the unit of a relation `r : 1 .. 5 <-> 8 .. 20` can be done as shown in Fig. 6. The unit of the relation will then be inferred by the plugin. These restrictions keep our unit type system decidable without limiting the overall expressiveness of our approach.

```
MACHINE UnitExample
VARIABLES
 /*@ unit 10**3 * m */ x,
 y
INVARIANT x:NAT & y:NAT & x>y
INITIALISATION x, y := 1, 0
OPERATIONS
 n <-- addToX = BEGIN n := x + y END;
END
```

Fig. 2 Example Usage of the Unit Pragma

```
MACHINE ConversionExample
VARIABLES
 /*@ unit 10**-2 * m */ x,
 /*@ unit 10**-3 * m */ y
INVARIANT x:NAT & y:NAT
INITIALISATION x, y := 0, 0
OPERATIONS
 mmToCm = x := /*@ conversion */ (10*y)
END
```

Fig. 3 Example Usage of the Conversion Pragma

```
/*@ unit_alias mps m * s**-1 */
MACHINE UnitExample
VARIABLES
 /*@ unit mps */ speed,
.....
END
```

Fig. 4 Example Usage of the Alias Pragma

```
/*@ new_unit yard */
MACHINE UserDefinedUnit
VARIABLES
 /*@ unit yard */ dist,
.....
END
```

Fig. 5 Example Usage of the New Unit Pragma

```
MACHINE RangeAndDomain
CONSTANTS
 /*@ unit m */ range,
 /*@ unit s */ domain
PROPERTIES
 range <: NAT & domain <: NAT
VARIABLES
 r
INVARIANT
 r : domain <-> range
INITIALISATION
 range, domain, r := 8 .. 20, 1 .. 5, {}
END
```

Fig. 6 Annotating Range and Domain

3.3 Unit Annotations for the TLA$^+$ Language

TLA$^+$ and B are both state-based formal methods, with subtle differences (e.g., concerning typing, refinement, or well-definedness) but they share the common foundation of predicate logic, arithmetic and set theory. In [18] Hansen and Leuschel have introduced a translation from TLA$^+$ to B, with the aim of using the ProB animator or B provers. In this article we can reap another side-effect of this translation: being able to use our unit inference algorithm for TLA$^+$ specifications. To achieve this, we have extended the TLA$^+$ syntax to enable unit annotations in a similar way as explained for B and Event-B in the previous section. Then, we have adapted the translator of [18] to ensure that it preserves the unit annotations. Obviously, this approach only works for the TLA$^+$ specifications that can be dealt with by [18], in particular, the translator has to be able to infer types for all variables of the specification.

We added unit pragmas to the TLA$^+$ language the same way we did for B. The only difference is the syntax of block comments in TLA$^+$. Instead of `/*@` and `*/` we

now enclose a pragma in `(*@` and `*)` to follow the usual block comment syntax.

In Fig. 7 we give an example for the syntax. The TLA$^+$ module `clock` specifies a simple clock storing its state in three integer variables: hours, minutes and seconds. It is an extension of the module given in [24]. The before-after-predicate mentioned in Sect. 2 is called `Clock_next`, the predicate describing possible initial values for the three variables is stored in `Clock_init`. Every time a `Clock_next` step occurs, the variable `seconds` is incremented by one. At the same time, hours and minutes are updated accordingly:

- The new value of `minutes` is set to (minutes + 1) modulo 60 if the old value of seconds is equal to 59. Otherwise the new value of minutes is equal to the old one.
- The new value of `hours` is set to (hours + 1) modulo 24 if the old value of minutes is equal to 59. Otherwise the new value of `hours` is equal to the old one.

We combine both to the full specification by stating `Clock == Clock_init /\ [][Clock_next]_{seconds}`, i.e. that the clock specifications behavior renders the predicate `Clock_init` true for the initial states while `Clock_next` is always true (indicated by the temporal operator `[]`). The additional `_{seconds}` allows stuttering steps to occur for which `seconds = seconds'` holds, meaning that the clock's state is allowed to enter an infinite sequence in which `seconds` does not change. See Chapter 2 of [24] for details.

The `clock` module will be part of our empirical evaluation for TLA$^+$ in Sect. 7.

4 Analyzing Units

4.1 Abstract Interpretation on Physical Units

4.1.1 Type Inference as Abstract Interpretation

Inferring units of measurement has a strong connection to type checking, which itself can be seen as a special kind of abstract interpretation [14] as outlined by Cousot in [13]. In consequence, inference of units throughout a B machine can be done by abstract interpretation of the operations of a machine and abstract evaluation of invariants, guards, etc.

Regarded as an abstract interpretation, type checking in B can be performed with the abstract domain outlined in Fig. 8. Initially, any type is still possible, represented by the bottom element $\bot$. Upon type checking, the type of each construct is inferred as one of the following inductively defined B types (see also Sect. 3.1):

- $\bot \in Types$
- $BOOL \in Types$
- $STRING \in Types$
- $\mathbb{Z} \in Types$
- $Given \subseteq Types$ where $Given$ contains all the user-defined deferred, enumerated or parameter sets
- $x \in Types \land y \in Types \Rightarrow x \times y \in Types$
- $t \in Types \Rightarrow \mathcal{P}(t) \in Types$
- $\top \in Types$

Ordinary type checking expressed as abstract interpretation would basically proceed as follows:

1. all variables will start out with the type $\bot$ and
2. literals get assigned their basic type, e.g., 1 will start with $\mathbb{Z}$, `"ab"` with $STRING$, and `TRUE` with type $BOOL$.
3. From this the type of expressions can be deduced in a bottom-up manner from the type signatures of the B operators (e.g., $+ : \mathbb{Z}\times\mathbb{Z} \to \mathbb{Z}$ or $\cup : \mathcal{P}(t)\times\mathcal{P}(t) \to \mathcal{P}(t)$).
4. This information is propagated through the operations of the B machine, by inferring the type of the right-hand side E for every assignment $x := E$ to a variable x, and then computing the least upper bound with the current type of x.
5. Steps 3 and 4 are repeated until a fixpoint is reached. It is considered to be an error if the type of a variable still contains the type $\bot$ after a fixpoint has been reached.

$\top$ denotes the fact that incompatible types have "reached" a certain value or subexpression. This can occur if a variable must have two different incompatible types, e.g., x in the predicate $x = \{x + 1\}$: the variable x is required to have both the type $\mathbb{Z}$ for the addition with 1 and the type $\mathcal{P}(\bot)$ for the equality test with a set.

$\bot$ denotes that we have had no type constraints yet, i.e., any type would be valid. This would, e.g., occur for x in the predicate $\{x\} \neq \{\}$, where the type of $\{\}$ would be inferred as $\mathcal{P}(\bot)$.[4]

Basically, types are ordered using the relation $\sqsubseteq$, forming the lattice in Fig. 8 and defined using the following five rules. We define $\sqsubset$ in the usual way: $s \sqsubset t$ iff $s \sqsubseteq t \land s \neq t$.

- $\bot \sqsubseteq t$ for any $t \in Types$
- $t \sqsubseteq \top$ for any $t \in Types$
- $t \sqsubseteq t$ for any $t \in Types$
- $s \times t \sqsubseteq s' \times t'$ iff $s \sqsubseteq s' \land t \sqsubseteq t'$.
- $\mathcal{P}(t) \sqsubseteq \mathcal{P}(t')$ iff $t \sqsubseteq t'$.

[4] Note that for Event-B, the Rodin tool will produce an error message if a variable or expression's type contains $\top$.

```
---------------------------- MODULE clock ----------------------------
EXTENDS Naturals
VARIABLES
  (*@ unit h *) hours,
  (*@ unit min *) minutes,
  (*@ unit s *) seconds
Clock_init ==
  hours \in (0 .. 23)
  /\ minutes \in (0 .. 59)
  /\ seconds \in (0 .. 59)
Clock_next ==
  hours' = (IF minutes # 59 THEN hours
            ELSE (hours + 1) % 24)
  /\ minutes' = (IF SECONDS # 59 THEN minutes
                 ELSE (minutes + 1) % 60)
  /\ seconds' = (seconds + 1) % 60
Clock == Clock_init /\ [][Clock_next]_{seconds}
---------------------------------------------------------------------
THEOREM Clock => []Clock_init
=====================================================================
```

Fig. 7 Example Usage of Pragmas in TLA$^+$

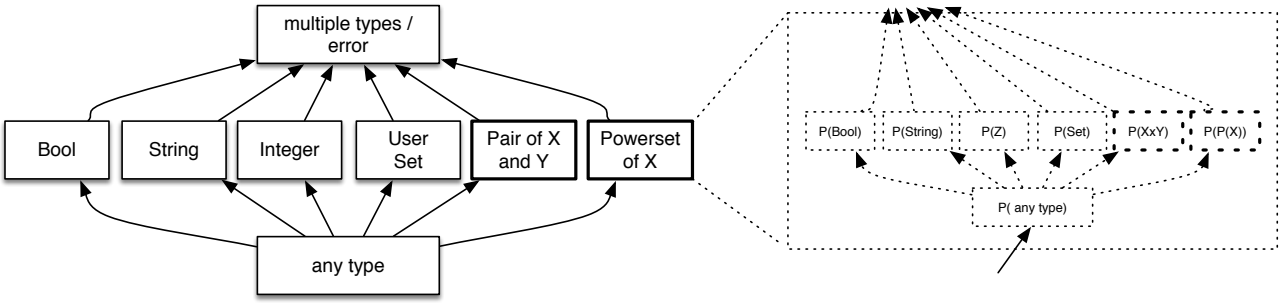

Fig. 8 B Type System and the Relation $\sqsubset$

4.1.2 Concrete Semantics for Units

Below, we will present unit inference in the form of an abstract interpretation. This raises the question which concrete semantics is abstracted by unit inference. Indeed, in the standard concrete semantics of B we cannot observe unit errors: adding an integer representing a speed to an integer representing time is a valid addition and returns a concrete integer value. Hence, for purposes of unit inference via abstract interpretation, we "pretend" that every number x (i.e., integer in the case of B) has an associated "invisible" unit u_x attached to it. In fact, in Sect. 3.2 we have done exactly that: attaching units to certain literals via pragmas.

We then also pretend that the concrete operators on integers follow the physical rules of unit manipulation. For example, the concrete semantics of the addition $x : u_x + y : u_y$ produces the pair $x + y : u_z$ where u_z is the least upper bound in our unit domain. We will describe this unit domain more precisely later on, but basically this domain is a lattice containing regular physical units along with the bottom element $\bot_u$ (representing the unrestricted unit) and the top element $\top_u$ representing an invalid unit.

The purpose of the unit inference, which we present in the rest of the article, is to infer the possible values of this "invisible" unit component for program points and variables of a B machine.

4.1.3 Unit Inference as Abstract Interpretation

The abstract domain used to perform unit analysis is an extension of the abstract domain used for type checking. While the types for boolean, string and the construction of sets, sequences and pairs remain, the integer type is replaced by an entire subdomain.

The domain is illustrated in Fig. 9, using meters as an example for the abstract elements. The subdomain consists of several layers. First of all, a variable of type integer may have no physical unit attached to it by

the user. This will be represented by $int(\bot_U)$ in the definitions below. From there on, an integer might be assigned any SI unit, including the dimensionless unit as a special case.

The dimensionless unit is added to be able to distinguish between variables for which no unit has been inferred yet and variables that definitely do not carry a unit. Furthermore, treating the dimensionless unit in the same way as the SI units allows the user to annotate variables like program counters. That way, an error involving incorrect usage of such a variable can be detected, rather than inferring a meaningless unit.

Figure 9 shows how our system will react to multiple inferred units. As can be seen easily, there is no upper bound for different units other than the error state on top.

Definition 1 A *base unit* is a triple $10^c \times u^e$ such that we have $c \in \mathbb{Z}$, $e \in \mathbb{Z}$, u being a base SI unit.

An abstract integer value can now be represented by a set of triples of the form $10^c \times u^e$ where $c \in \mathbb{Z}$ is the exponent of the coefficient (10^x is called the SI prefix), u an SI base unit symbol and $e \in \mathbb{Z}$ the exponent of the unit[5].

Note that we can not enforce that $e \neq 0$. While a base unit with $e = 0$ cannot be provided by the user, it might occur as an intermediate result. For instance, when dividing $10^2 \times m^1$ by $10^1 \times m^1$ we get $10^1 \times m^0$. Hence, units with $e \neq 0$ might still contribute to the unit analysis.

For convenience we define the following deconstructing functions:

Definition 2 For u a base unit with $u = 10^p \times s^e$, we define

$$\begin{aligned} prefix(u) &= p \\ symbol(u) &= s \\ exponent(u) &= e \\ dimension(u) &= s^e \end{aligned}$$

Definition 3 A *unit* is a set of base units $\{u_1, \ldots, u_k\}$ such that $\forall u_i, u_j \bullet j \neq i \Rightarrow symbol(u_i) \neq symbol(u_j)$.

With the definition above, $\frac{m}{s}$ would be expressed as $\{10^0 \times m^1, 10^0 \times s^{-1}\}$. The empty set of triples denotes a dimensionless integer value.

Another possible representation of a unit would be the sum of the c_i's in the definition above together with a set of pairs, each holding an SI base unit and its exponent. Doing so would certainly simplify the implementation of unit equivalence, arithmetic operations and unit conversions. However, we decided to stay as close as possible to the original user input and wanted to produce more precise error messages which are traceable to individual user provided triples in the input.

Definition 4 The set of all valid units is denoted by *Units*.

As in the type checking domain, we add an element $\bot_U$ to *Units* denoting that initially any unit is possible. Additionally, we define $\top_U$ representing the fact that multiple units were inferred. Again, this should not occur in a correct model.

To summarize, we can now define the concrete domain $\mathcal{C}$ and the abstract domain $\mathcal{A}$ used in our abstract interpretation framework.

Definition 5 The concrete domain $\mathcal{C}$ is the set of all possible B values.

$\mathcal{C}$ is mapped to the abstract domain $\mathcal{A}$, consisting of terms constructed from the function symbols $\Sigma = \{boolean, string, int, given, pair, set\}$. More formally, $\mathcal{A}$ is recursively defined by

- $boolean \in \mathcal{A}$
- $string \in \mathcal{A}$
- $\forall u \in Units \cup \{\bot_U, \top_U\} \Rightarrow int(u) \in \mathcal{A}$
- $\forall S \in Given, u \in Units \cup \{\bot_U, \top_U\} \Rightarrow given(S, u) \in \mathcal{A}$
- $x \in \mathcal{A} \wedge y \in \mathcal{A} \Rightarrow pair(x, y) \in \mathcal{A}$
- $t \in \mathcal{A} \Rightarrow set(t) \in \mathcal{A}$,

where *Given* is the set consisting of the enumerated, deferred or parameter sets given in the B machine.

Note, that we use both $set(t)$ and $given(t, u)$: While the first is a set with elements that may hold a unit themselves, i.e. a set of integers, the second has a unit directly attached to it, i.e. an enumerated set. We could unify them both to $given(t, u)$ and assign the unit of the contained elements of a $set(t)$ to u. However, we keep them separated as it simplifies the implementation.

The rules for the ordering of units are as follows:

- $\bot_U \sqsubseteq_U u$ for any $u \in Units \cup \{\top_U\}$
- $u \sqsubseteq_U \top_U$ for any $u \in Units \cup \{\bot_U\}$

The rules for the ordering of abstract values are as follows:

- $t \sqsubseteq t$ for any $t \in \mathcal{A}$
- $\bot \sqsubseteq t$ for any $t \in \mathcal{A}$
- $t \sqsubseteq \top$ for any $t \in \mathcal{A}$
- $\text{int}(u) \sqsubseteq \text{int}(u')$ iff $u \sqsubseteq_U u'$
- $given(t, u) \sqsubseteq given(t, u')$ iff $u \sqsubseteq_U u'$

[5] For convenience, some SI derived units and units accepted for use with the SI standard (see [36]) are stored on their own rather than converting them.

- $pair(s,t) \sqsubseteq pair(s',t')$ iff $s \sqsubseteq s' \wedge t \sqsubseteq t'$
- $set(t) \sqsubseteq set(t')$ iff $t \sqsubseteq t'$

To provide the abstraction and concretization functions, we need the following definition:

Definition 6 $unit : Variables \to Units \cup \{\bot_U\}$ maps variables to the user given units. It equals to $\bot_U$ if no unit is given.

To perform abstract interpretation the abstraction and concretization functions $\alpha : \mathcal{C} \to \mathcal{A}$ and $\gamma : \mathcal{A} \to \mathcal{P}(\mathcal{C})$ need to be defined. These functions have to be recursively defined, as the B type system contains arbitrarily nested data types. The following definitions of α and γ are used.

$$\alpha(x) = \begin{cases} boolean & \text{if } x \in BOOL \\ string & \text{if type of } x \text{ is } STRING \\ int(unit(x)) & \text{if } x \in \mathbb{Z} \\ given(S, unit(x)) & \text{if } x \in S, S \in Given \\ pair(\alpha(x_1), \alpha(x_2)) & \text{if } x = (x_1 \mapsto x_2) \\ set(\alpha(x_1)) & \text{if } x_1 \in x \\ set(\bot_U) & \text{if } x = \emptyset \end{cases}$$

with $S \in Given$ and

$$\gamma(y) = \begin{cases} BOOL & \text{if } y = boolean \\ \{s | \text{type of } s \text{ is } STRING\} & \text{if } y = string \\ S & \text{if } y = given(S,U) \\ \mathbb{Z} & \text{if } y = int(U) \\ \gamma(y_1) \times \gamma(y_2) & \text{if } y = pair(y_1, y_2) \\ \mathcal{P}(\gamma(y_1)) & \text{if } y = \ set(y_1). \end{cases}$$

Please note, that the definition of α does take neither the integer value of a variable nor the members of a set into account. The analysis we suggest does solely focus on the physical units and drops the concrete values of variables. However, animation and verification of concrete values is available through ProB and is thus not necessary here.

The orders $\sqsubseteq_U$ and $\sqsubseteq$ induce a least upper bound (*lub*) on $Units \cup \{\bot_U, \top_U\}$ and $\mathcal{A}$ respectively. For example, we have the following properties and examples over $\mathcal{A}$:

- $lub(int(\bot_U), int(u)) = lub(int(u), int(\bot_U)) = int(u)$.
- $lub(x,x) = x$.
- $lub(pair(int(\bot_U), int(u)), pair(int(v), int(\bot_U))) = pair(int(v), int(u))$.

Our implementation of unit equivalence is shown in Algorithm 1. We begin by multiplying the SI prefixes of each unit we want to compare[6]. If the products are not the same, the units can not be equivalent. Otherwise, we check if each base unit in the first unit corresponds to a base unit in the second unit and vice versa. If this is the case we consider the units to be equivalent.

The B instructions the abstract interpreter needs to implement can be categorized by their effect on the units of measurement:

1. Instructions like addition of integers or concatenation of sequences expect all operands and the result to hold the same unit.
2. Instructions that work on abstract elements which are composed in a different way while still holding the same units. The Cartesian product for example maps two sets to a set of pairs. In the process, the physical units do not change.
3. Instructions like multiplication or division are able to generate new units based on the units of their operands. A list of these operations is given below.

The first and second kind of operations can be implemented by syntactical unification or returning $\top_U$ if incompatible units are found. Thus, they solely rely on the implementation of the least upper bound we explained above.

The same result could be achieved by a classical type inference algorithm (e.g., Hindley-Milner). The third kind however needs more work, and is one justification for using abstract interpretation rather than syntactic unification-based type inference. Please note that, rather than using abstract interpretation, classical type inference algorithms can be made suitable by switching from syntactic unification to equational unification. See our discussion of alternative approaches in Sect. 8 for details.

The third group consists of all the operators of B, Event-B and TLA$^+$ that can generate new units. These are

- multiplication and division,
- exponentiation,
- the modulo division,
- cardinality of a set,
- elementary arithmetic operations if used as conversions.

Strictly speaking, the cardinality of a set does not create a new unit from the unit of its operand. However, as it returns the dimensionless unit regardless of the argument, it does not fit into the first two categories.

In Sect. 4.3 we explain how we handle conversions. Below, we discuss the remaining operations.

On the representation outlined above, multiplication is implemented by addition of the exponents of triples holding the same unit symbol. In case there is no

[6] Basically, this means adding the exponents of the leading 10 of each SI unit.

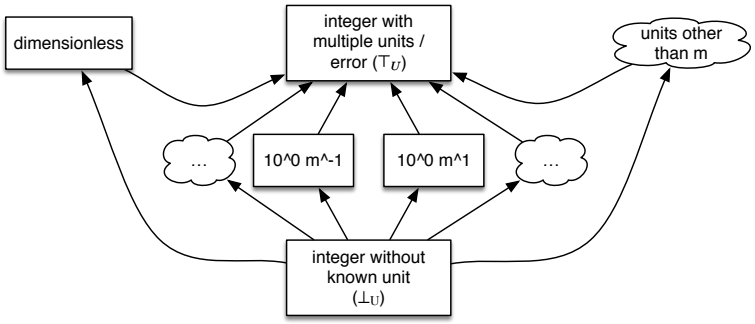

Fig. 9 Abstract Domain - Integers in Detail

other triple holding the same symbol, e.g., if we multiply with a dimensionless value, we copy over the original unit. See Algorithm 2 for an outline. With the multiplication in place, $\frac{a}{b}$ can easily be implemented as $a \times b^{-1}$.

A few operations were not immediately obvious, the first being modulo division. It was not clear what the correct operation on the unit domain had to be. The B Book [1] (page 164) defines the result of the modulo operation in B as

$$n \bmod m = n - m * \left\lfloor \frac{n}{m} \right\rfloor,$$

which is equal to the floored division described by Knuth in [22]. From this follows for the unit of $n \bmod m$

$$unit(n \bmod m) = unit\left(n - m * \left\lfloor \frac{n}{m} \right\rfloor\right) = unit(n).$$

Aside from the floored division modulo can be defined using the Euclidean division as suggested by Boute in [9]. In contrast to the definition above, the result of the modulo operation is required to be positive. The Euclidean division defines the modulo operation by three equations:

1. $q \in \mathbb{Z}$
2. $n = m * q + n \bmod m$
3. $0 \leq n \bmod m \leq |m|$

This definition is followed by the TLA$^+$ language[7].

The two definitions of modulo give different results for negative divisors. Consider for example $n = 1$ and $m = -2$. With the floored division, the result is

$$1 \bmod -2 = 1 - (-2) * \left\lfloor \frac{1}{-2} \right\rfloor = -1,$$

whereas the Euclidean definition results in 1.

The unit of $n \bmod m$ is given by the second equation: An addition is only valid if both addends and the result share the same unit. Hence, the Euclidean definition of modulo used by TLA$^+$ does not contradict the unit definition we made for the B language.

Please note that the Euclidean definition enforces m to share the unit of n and $n \bmod m$ as well, while the floored division cancels out the unit of m. For our implementation we decided not to enforce a restriction on the unit of m to remain compatible to TLA$^+$, B and Event-B at the same time.

All in all, the addition of TLA$^+$ and its interpretation of modulo to our analysis did not make any changes to our abstract semantics necessary. Following the above reasoning, in the current implementation of the unit interpreter the unit of $n \bmod m$ is the unit of n.

However, other definitions or implementations for modulo are certainly possible. See for example the T-, R- or C-division in [9]. These could easily be integrated into our abstract interpretation framework. For this work, we stay with the semantics of modulo described by our input languages. Up to now, our empirical evaluation did not reveal any problems with the given definition.

Another operation that is not immediately obvious is exponentiation. First of all, we decided to enforce that the exponent is dimensionless. This is in accordance with the usual handling of exponents for example in physics.

Another obstacle is that in order to compute the unit of b^e, the value rather than the unit of e is needed. However, the value has been abstracted away and is usually only available at runtime anyway. In our implementation, outlined in Algorithm 3, we therefore added an additional lookup that checks if the concrete value of e corresponds to an integer literal. If so, the value is known and the resulting unit can be computed as expected. If not, no unit can be computed and we have to assume that a unit error occurred to be on the safe side.

[7] In contrast to TLA$^+$, B does not support $n < 0$ as the first argument to the modulo operator. This is taken into account by the translation from TLA$^+$ to B.

Further operations frequently available in programming languages that are non-trivial include the trigonometric functions, the exponentiation function or logarithms. As these are neither available in B, Event-B nor TLA$^+$, we do not discuss them here. However, these operations and their effect on physical units has been widely discussed in the literature regarding unit analysis in physics. See for example [10] or [17]. Our abstract implementation framework could easily be extended to handle these operations.

Data: Units $x_1 \in \mathit{Units}, x_2 \in \mathit{Units}$
Result: Equality of Given Units
if $\sum_{t \in x_1} \mathit{prefix}(t) \neq \sum_{t \in x_2} \mathit{prefix}(t)$ **then**
  return false
else if $\{\mathit{dimension}(t) \mid t \in x_1\} = \{\mathit{dimension}(t) \mid t \in x_2\}$ **then**
  return true
else
  return false
end

Algorithm 1: Abstract Unit Equality

Data: Factors $x_1 \in \mathit{Units}, x_2 \in \mathit{Units}$
Result: Product $p \in \mathit{Units}$
$p := \emptyset$ **foreach** *triple* $10^{c_1} \times u^{e_1} \in x_1$ **do**
  if *there is a triple* $10^{c_2} \times u^{e_2} \in x_2$ **then**
    $p := p \cup \{10^{c_1+c_2} \times u^{e_1+e_2}\}$
    $x_2 := x_2 \setminus \{10^{c_2} \times u^{e_2}\}$
  else
    $p := p \cup \{10^{c_1} \times u^{e_1}\}$
  end
end
$p := p \cup x_2$
return p

Algorithm 2: Abstract Multiplication

Data: Base $b \in \mathit{Units}$, Exponent $\mathit{exp} \in \mathit{Units}$
Result: Product $p \in \mathit{Units}$
if *Sourcecode of exp is integer literal i* **then**
  $p := \emptyset$
  foreach *triple* $10^c \times u^e \in b_1$ **do**
    $p := p \cup \{10^{c*i} \times u^{e*i}\}$
  end
  return p
else
  return $\top_U$
end

Algorithm 3: Abstract Exponentiation

We perform a fixpoint search by executing all operations of a B machine. Additionally, we evaluate properties and invariants in every iteration.

Let us have a closer look at the algorithm using the pseudo code found in Algorithm 4. We start our search on an initial state containing the abstracted elements. First of all, we evaluate properties and invariants of the machine. This step might already infer some units: If, for example, an addition is encountered, the units of the operands and of the result are unified. If one of these units was already known, remaining $\bot_U$ might be replaced. However, we do not need to calculate the least upper bound in order to update the state, as we can not encounter assignments at this stage.

Following, we enter the fixpoint loop. We iterate over all operations of the machine at hand until the state does not change anymore. The most interesting step is the execution of the event. We have to perform several steps:

1. Preconditions or guards are evaluated. This is roughly equal to the evaluation of the invariants mentioned above. The state can not change, but invalid unit usages can occur and have to be tracked.
2. Afterwards, the substitutions can be executed. Here, we first need to compute the new values for all variables. Again, we only use the state and do not need to change it. There can however be unit errors.
3. Once the new values are found, the state has to be updated. At this step the least upper bound comes into play. We determine the least upper bound of the old and the new values, i.e. if an already inferred unit changes in an incompatible way we set the state value to $\top_U$. Otherwise, the state value is set to the least upper bound of the old and the new value.

After the state is updated, we perform error reporting. This involves checking two possible error cases:

- One of the state variables might have been set to $\top_U$. This happens if it was updated during the state transition and the old and new values are not compatible. Hence, their least upper bound is $\top_U$.
- Invalid unit usage might have occurred without causing $\top_U$ to occur in the state variables. This might happen when the invalid usage occurred during the evaluation of a boolean expression like the condition of an if statement.

Additionally, we have to update certain constraints we will explain in Sect. 5.

For the example machine in Fig. 2, the fixpoint search would perform the following steps:

1. Initialize the machine: If a unit is attached to an identifier, the unit is stored. Otherwise, $\bot_U$ is used. In the example, we set the initial state σ_0 to $\{(x, \mathit{int}(\{10^3 \times m^1\})), (y, \mathit{int}(\bot_U))\}$.

2. Evaluate the invariant on σ_0. The predicate $x > y$ allows us to infer the unit of y, updating $\sigma_0 = \{(x, int(\{10^3 \times m^1\})), (y, int(\{10^3 \times m^1\}))\}$. No incorrect usage of units is detected.
3. Execute `addToX` on σ_0:
   (a) Generate local state $\sigma_{IN} = \{(n, int(\bot_U)\} \cup \sigma_0$ by adding the return value n of `addToX` to the state. Initially, no unit is known for it.
   (b) Evaluate $x + y = int(\{10^3 \times m^1\})) + int(\{10^3 \times m^1\})) = int(\{10^3 \times m^1\}))$
   (c) Substitute n by calculating the least upper bound of $\bot_U$ and $int(\{10^3 \times m^1\}))$. The resulting output state is

$$\sigma_{OUT} = \{(n, int(\{10^3 \times m^1\})), (x, int(\{10^3 \times m^1\})), (y, int(\{10^3 \times m^1\}))\}.$$

4. The invariant is checked on σ_{OUT}. No changes occur.
5. Again, no incorrect usage of units is detected.
6. The next iteration executes `addToX` again and checks the invariant a second time. However, the state does not change and the fixpoint is reached.

Obviously, no new units have to be created in this example and a Hindley-Milner style type inference algorithm using syntactical unification would have been sufficient. Figure 10 shows a more involved extension of the example in Fig. 2. This time, a new unit has to be inferred and an error occurs:

1. The initial state is

$$\sigma_0 = \{(x, int(\{10^3 \times m^1\})), (y, int(\{10^3 \times m^1\})), (res, int(\{10^5 \times m^1\}))\}.$$

2. Evaluate the invariant on σ_0. No error is detected.
3. Execute `multXandY` on σ_0:
   (a) The local state of `multXandY` is empty as no local variable is needed.
   (b) Evaluate $x * y = int(\{10^3 \times m^1\})) * int(\{10^3 \times m^1\})) = int(\{10^6 \times m^2\}))$
   (c) Substitute res by calculating the least upper bound of $int(\{10^5 \times m^1\}))$ and $int(\{10^6 \times m^2\}))$. As these do not correspond to the same unit, Algorithm 1 returns false. The least upper bound is $\top_U$. Therefore, the resulting output state is

$$\sigma_{OUT} = \{(x, int(\{10^3 \times m^1\})), (y, int(\{10^3 \times m^1\})), (res, int(\{\top_U\}))\}.$$

4. The invariant is checked without changing the state.
5. This time, a unit error is detected by successfully searching the state for an occurrence of $\top_U$.
6. With the next iteration, a fixpoint is reached.

```
MACHINE UnitExample
VARIABLES
 /*@ unit 10**3 * m */ x,
 /*@ unit 10**3 * m */ y,
 /*@ unit 10**5 * m */ res
INVARIANT x:NAT & y:NAT & res:NAT & x>y
INITIALISATION x, y,res := 1, 0, 0
OPERATIONS
 multXandY = BEGIN res := x * y END;
END
```

Fig. 10 Example containing New Unit / Unit Error

4.2 Dealing with Function Calls

```
MACHINE SquaringFunction
OPERATIONS
 res <-- Square_Call(ss) =
  PRE ss:INTEGER
  THEN res := ss*ss END
END

MACHINE GenericUsageOfSquare
USES SquaringFunction
DEFINITIONS  Square_Def(aa) == (aa*aa)
VARIABLES
 /*@ unit m */ xx,
 /*@ unit m */ yy,
 vv, ww
INVARIANT
 xx:NATURAL & yy:INTEGER & vv:NATURAL & ww:NATURAL
INITIALISATION
 xx:=2 || yy := -2 || vv := 0 || ww := 0
OPERATIONS
 Sq = BEGIN
  vv := Square_Def(xx) ||
  ww <-- Square_Call(yy)
 END
END
```

Fig. 11 Calls with Unit Parameters

One problem that arises in unit analysis is that functions and methods can be parametric in units. In general, we do not want to define several functions that essentially perform the same operation on different units. To overcome this problem, F# introduces a generic annotation that allows to define for example a squaring function to map from any unit u to u^2 [21]. In the remainder of this section, we will reuse this example to show how our approach deals with it.

```
σ = {(identifier of x, α(x)) : x variable or constant}
evaluate properties / invariant (might replace ⊥_U by a U ∈ Units ∪ {⊤_U} in σ)
repeat
    foreach operation / event do
        update σ by executing operation / event:
            evaluate preconditions / guards (might replace ⊥_U by a U ∈ Units ∪ {⊤_U} in σ)
            perform substitutions x := x' by setting x to lub(x, x') in σ
        if parameter or return value contains ⊤_U then
            report error
        end
        evaluate properties / invariant (might replace ⊥_U by a U ∈ Units ∪ {⊥_U, ⊤_U} in σ)
        if σ contains ⊤_U (i.e. a state variable has been set to ⊤_U) then
            report error
        end
        if invalid unit usage detected during evaluation (e.g., in conditions, etc.) then
            report error
        end
        evaluate unit constraints
    end
until σ did not change in loop
```

Algorithm 4: Fixpoint Search (on B / Event-B machine)

In B, there are two ways of defining functions that should be generic. Both are shown in Fig. 11.

- First, the `DEFINITIONS` section of a machine can be used to define macros. In Fig. 11, the machine `GenericUsageOfSquare` defines `Square_Def(aa)` to be replaced by `aa*aa`. As this is implemented in ProB by textual replacement, there is no function call involved when computing `vv`. It is handled by the abstract interpreter just like a direct multiplication.
- Second, B allows to call operations that are defined in another machine. In Fig. 11 the machine `GenericUsageOfSquare` uses the machine `SquaringFunction`. Hence it is allowed to call `Square_Call` and assign the return value to the variable `ww`.

In the second case, a function call has to be performed. Our abstract interpreter based approach is able to handle the genericity very naturally:

- Upon calling `Square_Call` a local environment containing the parameter `ss` and the return value `res` is set up.
- We store the value of `yy` inside the local environment under `ss` to initialize the parameter with the given value. This copies over the abstract value holding the unit to the local state of the called function.
- Using the local state, `Square_Call` computes the abstract multiplication of `ss*ss` and stores the result inside `res`.
- After the function returns, the value of `res` is taken from the local state and assigned to `ww`.

The process is identical to the usual process of calling a function inside an interpreter. In particular, it is identical to the process performed by our regular, non-abstract, B interpreter.

There is, however, one key difference between our approach and the type-inference based one taken by languages like F#. Within a naive abstract interpretation framework, the function body has to be evaluated each time the function is called. In contrast, a type-inference based analysis could compute a generic type like $u \mapsto u^2$, that maps any unit to its square. These generic types are then sufficient to perform unit analysis. No further evaluation of the body of a function is necessary.

For our input languages B, Event-B and TLA$^+$the repeated evaluation of function bodies does not noticeably slow down the interpreter. To extend our approach to other languages, particularly those that heavily rely on function calls, the abstract interpreter could be extended by pre-computing a general signature of the functions with respect to units [33,32]. Generalizations or function summaries could be computed upon the first call to a function or before the analysis starts. These summaries could then be reused during following calls to avoid evaluating the body of a function multiple times.

4.3 Unit Conversion

A special kind of units error is the usage of wrong conversion rules or even skipping a unit conversion all together. While this is a problem for the conversion between different SI units, the situation gets even worse if the conversion between metric and imperial units is taken into account. As state-of-the-art software projects are often developed on a worldwide basis, programmers

accustomed to different units of measurement work together on the same code base. A common example that might be problematic is the decision between and consistent usage of metric or imperial units or the date format.

In fact, errors in unit conversions have already led to different real-world problems:

- On July 23, 1983 a Boeing 767 flying for Air Canada was forced to perform an emergency landing due to low fuel pressure. On a flight altitude of about 41000 feet both engines stopped. Luckily the pilot was able to safely land the plane on an abandoned military airport. Investigations later unveiled that the ground crew only refueled the airplane by half the necessary amount, because they mixed up pounds per liter and kilograms per liter when configuring the fueling equipment [28]. While this error happened in a mental calculation a programmer could have just as easily fallen into the same trap.
- On September 23, 1999 NASA lost the Mars Climate Orbiter, a space orbiter send to Mars as a part of the Discovery program. It was equipped with a radiometer and a camera. Aside from collecting climate data, NASA intended to use it as a communication relay to the Mars Polar Lander. The whole mission had a budget of about 320 million dollars, with the spacecraft accounting for close to $130 millions. When trying to place the probe in a stable orbit around Mars, the Mars Climate Orbiter crashed into it. NASA later revealed a communication problem between NASA and the manufacturer of the navigation software. While NASA was using the SI-unit $\frac{N}{s}$ for the engine's impulses, the navigation software was using the imperial unit $lb_f = \frac{lb}{s}$ [35].

Some of the common programming errors regarding unit conversions can be avoided by an analysis like the one we suggest.

As mentioned in Sect. 3.2 we introduced a pragma that allows us to annotate arithmetic operations as unit conversions as shown in Fig. 3.

This is necessary because the plugin itself is obviously unable to distinguish between modifying the amount and converting the same amount to a different unit just by looking at the operation itself. In Fig. 3, `10*y` could as well set `x` to ten times `y` instead of converting from centimeters to millimeters.

We considered several modes of operation for unit conversions:

- The easiest way to do unit conversion would be to rely on especially crafted units. In order to convert from cm to mm, we could multiply with 100 $mm * cm^{-1}$. However, we would rely on the user to input the correct conversion unit together with the matching conversion factor. We dropped this approach because we wanted to strive for more automatic verification of conversions.
- A second possible implementation would have been to try to auto-detect if the users attempts a unit conversion. For this to work we would set up both possibilities inside the interpreter and evaluate both paths. Eventually, one should lead to a fixpoint containing the final unit information of the model at hand. While the approach seams reasonable, we would have needed to figure out how to deal with conflicts:
  - Certain paths might return an error while others do not. Interaction with the users would be the only proper way to decide if the model is erroneous.
  - There might be a choice of possible conversions, e.g., if one or more units are no inferred yet. Some of them might lead to a later inconsistency or a unit error. Again, user interaction would be necessary.

Implementing the second choice could be done employing the backtracking capabilities of the underlying Prolog system. However, it would render the results of our unit analysis more ambiguous. Error messages dealing with possible conversions that were inserted automatically would certainly be harder to understand. A more advanced version of our analysis could however suggest conversions in cases a unit error is detected somewhere.

In consequence, we decided on an approach quite similar to the first. The user still inputs the required conversion factors. However, instead of crafting a conversion unit, the arithmetic operation is annotated as a conversion. With the added annotation, our plugin is able to verify the correctness of the conversion factor and to figure out the resulting unit. This could not be done without an explicit distinction between converting and multiplying the amount.

Let us discuss our algorithm on another example machine that shows how a common unit conversion mistake is prevented by our analysis. The B machine in Fig. 12 computes the conversion between hours and seconds. Once the fixpoint algorithm reaches a multiplication it has two possibilities:

- If there is no `conversion` annotation, the plugin tries to infer units for `3600` and `seconds`.
- However, with the added pragma, the plugin knows that it is supposed to find a unit for `seconds` that is compatible with the unit of `hours` and the given conversion factor.

Finding a compatible unit is done by consuming the conversion factor step by step. First, the plugin knows that it can use a conversion factor of 60 to convert from hours to minutes. This is stored as an exception to the general rule of "multiply by ten" for SI unit conversion. This leads us to the new expression `seconds := 60*x` where x is stored in minutes. We apply the algorithm recursively as long as the conversion factor is greater than 1[8]. If there are several possibilities for a conversion, we rely on the backtracking capabilities of Prolog to try them out in a nondeterministic way.

In addition to conversion factors, our analysis is able to handle linear conversions like the one Celsius to Fahrenheit.

```
MACHINE NonSIConversion
VARIABLES
  seconds,
  /*@ unit h */ hours
INVARIANT seconds:NAT & hours:NAT
INITIALISATION seconds,hours := 0,5
OPERATIONS
  convert = seconds := /*@ conversion */ (3600*hours)
END
```

Fig. 12 Example Usage of the Conversion Pragma

5 Extending Abstract Interpretation with Constraints

Below we show that our abstract interpretation scheme on its own still has some limitations. Consider the B machine in Fig. 13, where the variable x contains a length in meters, t holds a time interval in seconds, and the unit of y should be inferred. Evaluating the assignment $t := (x*y)*t$ requires several interpretation steps:

1. The interpreter computes the product of x and y. As $y = int(\bot_U)$, the interpreter can only return $int(\bot_U)$ as a result.
2. In consequence, the interpreter finds that $(x*y)*t = int(\bot_U) * t = int(\bot_U) * int(\{10^0 \times s^1\}) = int(\bot_U)$.
3. The assignment $t := (x * y) * t$ is evaluated by computing the least upper bound of t and $int(\bot_U)$, i.e., $int(\{10^0 \times s^1\})$. No information is propagated back to the inner expressions; we are thus unable to infer the unit of y.

The example shows that it is necessary to attach some kind of constraints to the resulting variables containing $\bot_U$. Inside, the operation and the operands that lead to $\bot_U$ are stored. They will be used to re-execute the operation if more information is known.

Definition 7 A *unit constraint* can be attached to any variable x with $x = \bot_U$. It consists of two operands o_1, o_2 for which $o_1 = \bot_U \vee o_2 = \bot_U$ holds. Hence, no result can be computed at the moment. Furthermore, it contains an operation $op \in \{mult, div, pow\}$. It stores the fact that

- $x = o_1 * o_2$, iff $op = mult$,
- $x = \frac{o_1}{o_2}$, iff $op = div$,
- $x = o_1^{o_2}$, iff $op = pow$.

We implemented constraints for multiplication, division and exponentiation, as those cannot be handled by the interpretation-based algorithm alone. There is no need for constraints in case of addition, subtraction, etc. because those can only be correct if operands and result hold the same unit. Hence, the variables storing the inferred units can just be unified.

As shown in Algorithm 4 the constraints are evaluated after each iteration of the fixpoint search.

In general, if a variable with an attached constraint has been unified with another variable in the current iteration, the unit analysis reacts differently depending on the values of the operands. The behavior as outlined in Algorithm 5 is:

- The variables do not hold a physical unit at the moment. Hence, we cannot solve the constraint.
- After the unification, the variable contains a physical unit. Now, we have to look at the operands o_1, o_2 inside the constraint:
  - If $o_1 = \bot_U \wedge o_2 = \bot_U$, there are multiple possible solutions. We again delay the computation to the next iteration of the fixpoint algorithm.
  - If $o_1 \neq \bot_U \wedge o_2 \neq \bot_U$ the constraint is dropped without further verification as there is nothing left to infer.
  - If $o_1 \neq \bot \wedge o_2 = \bot_U$, we can compute o_2 by reversing the operation.
  - If $o_1 = \bot \wedge o_2 \neq \bot_U$, we can compute o_1 by reversing the operation.
- Further unifications in the second step may allow to solve constraints on other variables.

Using the exponentiation in reverse underlies the same limitations mentioned in Sect. 4.1.3.

In the example given in Fig. 13 two constraints are used to infer the unit of variable y. First, a constraint

[8] Note that there is only experimental support for floating point numbers in B and Event-B. Hence, we currently do not support conversions like the one from seconds to minutes using $\frac{1}{60}$ as a conversion factor. It would, however, be possible to implement this as an extension of our approach and we will do so once real / float support stabilizes. TLA$^+$ supports real numbers. However, at the moment, our translation does not.

Data: Unit Constraint c (attached to variable $x \neq \bot_U$)

extract operation op and operands o_1, o_2 from c
if $o_1 \neq \bot_U \wedge o_2 \neq \bot_U$ **then**
  remove constraint c
else if $o_1 = \bot_U \wedge o_2 \neq \bot_U$ **then**
  switch op **do**
    case *mult* **do** $o_1 := x/o_2$
    case *div* **do** $o_1 := x * 0_2$
    case *pow* **do** $o_1 := \sqrt[o_2]{x}$
  end
else if $o_1 \neq \bot_U \wedge uo_2 = \bot_U$ **then**
  switch op **do**
    case *mult* **do** $o_2 := x/o_1$
    case *div* **do** $o_2 := o_1/x$
  end
end

Algorithm 5: Handling of Unit Constraints

containing x and y is attached to the intermediate result of the inner multiplication. We receive an intermediate variable *temp*, with an attached unit constraint holding $op = mult$, $o_1 = x$ and $o_2 = y$. The result of the outer multiplication is annotated in the same way: $t * temp$ results in a variable $\widehat{temp}$ with a constraint holding $\widehat{op} = mult$, $\widehat{o_1} = t$, $\widehat{o_2} = temp$. When computing the assignment $t := \widehat{temp}$, we know that the outer multiplication has to return seconds for the usage of units to be consistent. $\widehat{temp}$ now has the value s and the attached constraint can be used to infer the unit value of $\widehat{o_2}$ by

$$\widehat{o_2} = \frac{\widehat{temp}}{\widehat{o_1}} = \frac{s}{s} = dimensionless.$$

Hence, *temp* is inferred as dimensionless. This enables the constraint attached to *temp* itself, allowing the algorithm to compute a value for o_2 in the same way. We have

$$o_2 = \frac{temp}{o_1} = \frac{dimensionless}{m} = m^{-1}.$$

As $o_2 = y$ the unit of y is now known to be m^{-1}.

We do not perform error handling when evaluating constraints. If a new unit has been inferred, the state has changed and the next iteration of the fixpoint search will eventually discover new errors. If the state did not change, solving the constraint did not add any information. We could only detect an error already reported. See Algorithm 4 for details.

6 A Refinement Chain for Physical Units

The plugin as initially introduced by us in [23] was only able to analyze the usage of physical units throughout a single B machine and its included machines. However, there was no way of assuring the consistent usage of units inside a chain of B machines refining each other. Furthermore, we learned through our empirical evaluation that it would sometimes come in handy to be able to assign a more abstract unit like “length” or “speed” without the need to assign a specific SI-unit.

```
MACHINE InvolvedConstraintUnits
VARIABLES /*@ unit m */ x, y, /*@ unit s */ t
INVARIANT
 x:NAT & y:NAT & t:NAT
INITIALISATION x, y, t := 1, 1, 1
OPERATIONS
 Op = BEGIN t := (x*y)*t END
END
```

Fig. 13 Machine requiring involved constraint solving

```
MACHINE RefinementExample
VARIABLES
 /*@ unit length */ abstract_length,
 /*@ unit m */ distance,
 /*@ unit m**2 */ area
INVARIANT
 abstract_length:NAT & distance:NAT & area:NAT
INITIALISATION abstract_length,distance,area := 0,0,0
OPERATIONS
 n <-- correct =
  BEGIN n := abstract_length + distance END;
 n <-- wrong =
  BEGIN n := abstract_length + area END;
END
```

Fig. 14 Example Usage of Unit Refinement

Both problems boil down to refinement:

- Analyzing physical units through a chain of B machines means being able to convert from more abstract units used in the abstract refinement levels to concrete units that are assigned to variables in the refinement levels closer to an actual implementation. Essentially, a more abstract variable needs to hold a more abstract unit than its refined counterpart.
- The same situation might occur inside a single machine. A user should be allowed to use both fully specified units and more abstract units for different variables occurring in the same model. Beyond the first case, this means that the different refinement levels might occur in a common expression.

We were able to integrate refinement into our approach in a straight forward way. First of all, the definition of a unit has to be relaxed to allow for only partially known units:

Definition 8 A *refinable base unit* is a triple $10^c \times u^e$ such that we have $c \in \mathbb{Z} \cup \{unknown\}$, $e \in \mathbb{Z}$, u being a base SI unit.

The deconstruction functions from Def. 2, such as *symbol*(.) and *prefix*(.), can be lifted to refinable units as expected. Definition 3 can now be adapted for refinable units:

Definition 9 A *refinable unit* is a set of refinable base units $\{u_1, \ldots, u_k\}$ such that

$\forall u_i, u_j \bullet j \neq i \Rightarrow symbol(u_i) \neq symbol(u_j)$.

With this definition, we allow unit triples that contain the placeholder "unknown" as the first element. We do not allow units without a known base SI unit or exponent, as these would more or less be equal to a fully unknown unit. Now, we can define which units are *abstract*, i.e. they can be refined and which units are *concrete*, i.e. they can not be refined anymore.

Definition 10 A refinable unit u is called *abstract*, if there exists a triple $t \in u$ such that $prefix(t) = unknown$.

Definition 11 A refinable unit u is called *concrete*, if it is not abstract.

These changes result in a new layer added to the domain shown in Fig. 9. The new units are added in between $\bot_U$ and the fully inferred units above it.

Now, we have a representation for an abstract unit like "length", namely $\{10^{unknown} \times m^1\}$. To integrate abstract units into our abstract interpretation framework, only a few changes are necessary: While the abstraction function α as well as the concretization function γ do not need to be changed aside from extending the set of all units, we have to extend the definition of the least upper bound.

If we compute the least upper bound of two units, we have to be able to compute a unit less abstract than both of the units. In case one of the units is still set to $\bot$, we can simply return the other one. If one of the units does already hold $\top$, we can only return $\top$. If the units are equal, the least upper bound is equal to both of them.

However, if one of the units is abstract and not identical to the other one, computing the least upper bound has to be done in a more elaborate fashion. To do so, we compare the units triple by triple and replace "unknown" where possible. Our implementation is shown in Algorithm 6. It can be seen as an extension to the unit equivalence algorithm shown in Algorithm 1.

A concluding example can be found in Fig. 14. It features an abstract variable `abstract_length` that stores

```
Data: Units x1 ∈ Units, x2 ∈ Units
Result: Refining Least Upper Bound lub ∈ Units
lub := ∅
foreach triple t1 ∈ x1 do
    if there is a triple t2 ∈ x2 with
     symbol(t1) = symbol(t2) then
        if exponent(t1) ≠ exponent(t2) then
            return ⊤
        end

        if prefix(t1) = unknown ∨ prefix(t1) = prefix(t2)
         then
            lub := lub ∪ {t2}
        else
            if prefix(t2) = unknown then
                lub := lub ∪ {t1}
            else
                return ⊤
            end
        end
        x2 := x2 \ {t2}
    else
        return ⊤
    end
end
if x2 ≠ ∅ then
    return ⊤
else
    return lub
end
```

Algorithm 6: Refining Least Upper Bound

any length. Furthermore, there a two concrete variables `distance` and `area` with the appropriate units attached to them. The two operations highlight how our inference algorithm works on abstract units:

- If the operation `correct` is executed, the interpreter has to compute the sum of `abstract_length` and `distance`. As `abstract_length` holds the abstract unit $10^{unknown} \times m^1$ and `distance` holds the concrete unit $10^0 \times m^1$, the least upper bound is $10^0 \times m^1$ according to Algorithm 6. Obviously, no unit error occurs.
- On the contrary, executing the operation `wrong` does result in a unit error: `area` holds the concrete unit $10^0 \times m^2$. Hence, in Algorithm 6 the else branch is executed and $\top$ is returned as the resulting least upper bound of `abstract_length` and `area`.

7 Empirical Results

Our empirical evaluation examines the following three key aspects:

- the effort needed to annotate the machines and debug them if necessary;

Table 1 Benchmarks - Analysis

Classical B					
No.	machine	LOC	# operations	# iterations	time analysis
1	Car	74	4	2	< 10 ms
2	TrafficLight	81	2	1	< 10 ms
3	System	322	20	2	50 ms
4	measure	42	2	1	< 10 ms
5	utils	24	2	1	< 10 ms
6	utils_i	38	2	1	< 10 ms
7	ctx	16	0	1	< 10 ms
8	ctx_i	16	0	1	< 10 ms
9	fuel0	64	2	2	< 10 ms
10	fuel_i	106	6	2	< 10 ms
11	compensated_gradient	3079	20	3	620 ms
12	vital_gradient	986	4	3	160 ms
13	sgd	773	0	2	170 ms
14	params_scn_f6_372_bis	99	0	2	< 10 ms
15	actions_scn_f6_372_bis	526	17	2	25 ms
16	params_custom_unit	99	0	2	20 ms
17	actions_custom_unit	526	17	2	35 ms
Event-B					
No.	machine	LOC	# operations	# iterations	time analysis
18	T_m0	115	6	3	20 ms
19	T_m1	179	11	3	30 ms
20	C_m0	108	4	3	20 ms
21	C_m1	141	4	3	20 ms
22	C_m2	162	4	3	40 ms
23	C_m3	228	7	3	90 ms
TLA$^+$					
No.	machine	LOC	# operations	# iterations	time analysis
24	Clock	18	1	1	240 ms
25	WaterTank	37	1	1	260 ms

– additionally, the number of iterations performed and the time spent in search for a fixpoint was of particular interest;
– the accuracy of the abstract interpretation.

To evaluate the performance of the unit analysis plugin, several case studies on different crafted and real world example machines were performed. Most of the machines used in the case studies can be found online[9]. We will discuss six of them in the following sections.

7.1 Traffic Light Warning System

The first case study is based on an intelligent traffic light warning system. The traffic light broadcasts information about its current status and cycle to oncoming cars using an ad-hoc wireless network. The system should warn the driver and eventually trigger the brakes, in case the car approaches a traffic light and will not be able to pass when it would be still allowed.

[9] `http://www.stups.uni-duesseldorf.de/models/sosym_units/`

After the annotations were done, the plugin reported an incorrect usage of units. The underlying cause was the definition

$$ceil\_div(a,b) == \frac{a}{b} + \frac{b-1+a \bmod b}{b},$$

a ceiling division that rounds the result up to the next integer value. It was introduced to keep the approximation of breaking distances sound.

The expected result for the unit of ceil_div is the unit of a regular division, that is the unit of a divided by the unit of b. However, the definition above does not lead to a consistent unit. Thus, the former definition of ceil_div was not convenient for use with the unit plugin. It was changed to

$$ceil\_div(a,b) == \frac{a}{b} + \frac{\min(1, a \bmod b)}{(b+1) \bmod b},$$

which leads to the expected result.

Furthermore, the speed of the car was stored as a length and implicitly used as a “distance per tick”. Our plugin discovered that the speed variable could not be associated with any suitable unit without giving further errors.

Table 2 Benchmarks - Annotating

Classical B					
No.	machine	LOC	# constants & variables	thereof annotated	time annotating
1	Car	74	6	2 (≈ 33%)	≈ 30 min
2	TrafficLight	81	6	2 (≈ 33%)	≈ 20 min
3	System	322	13	4 (≈ 31%)	≈ 60 min
4	measure	42	3	1 (≈ 33%)	≈ 5 min
5	utils	24	0	0 (0%)	≈ 5 min
6	utils_i	38	0	0 (0%)	≈ 5 min
7	ctx	16	3	1 (≈ 33%)	≈ 5 min
8	ctx_i	16	3	1 (≈ 33%)	≈ 5 min
9	fuel0	64	6	1 (≈ 17%)	≈ 5 min
10	fuel_i	106	6	1 (≈ 17%)	≈ 5 min
11	compensated_gradient	3079	766	-[a]	≈ 45 min
12	vital_gradient	986	263	-[a]	≈ 45 min
13	sgd	773	251	-[a]	≈ 90 min
14	params_scn_f6_372_bis	99	34	10 (≈ 29%)	≈ 20 min
15	actions_scn_f6_372_bis	526	51	10 (≈ 20%)	≈ 20 min
16	params_custom_unit	99	34	10 (≈ 29%)	-[b]
17	actions_custom_unit	526	51	10 (≈ 20%)	-[b]
Event-B					
No.	machine	LOC	# constants & variables	thereof annotated	time annotating
18	T_m0	115	9	3 (≈ 33%)	≈ 15 min
19	T_m1	179	13	2 (≈ 15%)	≈ 15 min
20	C_m0	108	13	2 (≈ 15%)	≈ 15 min
21	C_m1	141	16	2 (≈ 13%)	≈ 15 min
22	C_m2	162	17	2 (≈ 12%)	≈ 15 min
23	C_m3	228	19	2 (≈ 11%)	≈ 15 min
TLA$^+$					
No.	machine	LOC	# constants & variables	thereof annotated	time annotating
24	Clock	18	3	3 (100%)	≈ 1 min
25	WaterTank	37	7	2 (≈ 29%)	≈ 15 min

[a] Several of the constants and variables inside these models are dimensionless and used for program counters, track ids and so on. Hence, the number of annotated variables is not representative as a benchmark and is therefore left out.

[b] Same as above with meters replaced by the custom unit "yards". Hence, no time was measured.

To overcome the problem we introduced a new variable `TICK_LENGTH`, set it to one and annotated it with seconds. With the speed being stored in meters per second and the position of the car in meters, a multiplication with `TICK_LENGTH` fixes the unit usage.

Regarding the performance factors mentioned above, the number of iterations and the computation time was measured. Furthermore we timed annotating the machine and correcting unit errors if necessary. The results are listed in benchmarks 1 to 3 in Tables 1 and 2. For comparison purposes, the table also lists the number of lines of code and the number of operations for each machine[10]. No variables contained $\top_U$, so the abstract interpretation did not lead to a loss of precision.

The effort needed to annotate and correct the model was reasonably low, in particular when compared with the time needed to create the model in the first place. The evaluation also showed that it is easy to split developing the model and performing unit analysis.

7.2 ClearSy Tutorial

The second case study used a ClearSy tutorial on modeling in B[11]. It contains both abstract and implementation machines (all in all seven B machines). The system uses several sensors to estimate the remaining amount of fuel in a tank.

The first step was to annotate all variables with their respective units. When no error was found, the number of pragmas was gradually reduced, to measure the efficiency of our approach with less user input available. Eventually, we only needed one pragma for the abstract and one for the implementation machine. All

[10] Both were counted on the internal representation of the machines. Thus, the metrics include code from imported machines. Comments are not counted, as they are not in the internal representation. However, new lines used for pretty printing are counted.

[11] The tutorial including the machines can be found at `http://www.tools.clearsy.com/wp1/?page_id=161`.

other units could be inferred[12]. In the process, no unit reached $\top_U$. The benchmarks are presented in Table 1 and Table 2, rows 4 to 10: again, the computation time is very low and only two iterations are needed to fully infer the units of all variables. The additional step of introducing an implementation level did not lead to longer computation times. No significant annotation work was needed on the implementation machine, once the abstract machine had been analyzed.

7.3 Hybrid Systems

For the next case study, we used some of the Event-B hybrid machines described in [4]. Hybrid systems usually consist of a controller working on discrete time intervals, while the environment evolves in a continuous way. To deal with the challenge of analyzing both a discrete and a continuous component simultaneously, time is modeled by a variable called "now". It can be used as input to several functions mapping it to a real-world observation, taken from the environment at that moment in time. Hence, this approach is an addition to the former case studies using different techniques.

From the three models described in [4], two were used as case studies: the hybrid_nuclear model and the hybrid_train model. The hybrid_nuclear model was originally introduced in [7]. It models a temperature control system for a heat producing reactor that can be cooled by inserting one of two cooling rods once a critical temperature is reached. The hybrid_train example was originally developed in [31]. It features one or more trains running on the same line. Each train receives a point m on the track where it should stop at the least.

The machines with less abstraction introduced hybrid components by using functions as explained above. The unit plugin stores these functions as mappings from one unit to another. Hence, to be able to fully analyze the usage of units inside a machine, there have to be annotations on both the discrete and the continuous variables.

In the train models, the variables holding speed and position were annotated in the abstract model. In the more concrete model, the acceleration was stored as $\frac{m}{s}$ while one of the time variables was annotated as seconds. Both configurations lead to full inference of the used units through all variables and constants. No unset variables or variables with multiple inferred units occurred.

In the hybrid_nuclear models, different combinations of annotating one of the temperatures and one of the time variables were tried. Regardless of the combination, once both a temperature and a time were annotated, all other units could be inferred. The belonging benchmarks are 18 to 23 in Tables 1 and 2.

[12] The exception being variables and sets belonging to the system's status. Here, no unit of measurement applies and no unit was inferred.

7.4 Alstom Models

To evaluate the performance of the unit plugin on large scale examples, several B railway models from Alstom were used as benchmarks. The models precisely simulate the behavior of trains in order to fine-tune parameters of control systems. As most of these machines are confidential, neither source code nor implementation details can be provided.

During the evaluation, the plugin showed some difficulties in handling large B functions or relations of large cardinality. Mainly, this is because for every new element that is added to a relation, the plugin tries to infer new units for range and domain. In almost all cases this does not modify the currently inferred units. In a future revision, the plugin might rely more on information from the type checker to reduce the number of inferences.

Furthermore, lookup of global variables and their units slowed the interpreter down. When accessing elements of deferred or enumerated sets, the machine had to be unpacked frequently. To overcome this limitation, certain units are now cached to reduce the lookup time. As discussed in 3.2, there is no way to annotate both range and domain of a function or relation at once, as this would require another pragma or at least a second variant of the unit pragma. Therefore, they have to be annotated on their own. Our evaluation shows that this is possible without substantial rewriting of a machine.

Examples 11 to 13 in Table 1 and 2 shows the benchmark results for some of the Alstom machines. Total lines of code and number of operations are again given to ease comparison with the former case studies. As can be seen, our analysis scales to these large, industrial examples.

7.5 Alstom Models 2

We performed a second case study on machines taken from Alstom transportation modeling an aspect of the European Train Control System (ETCS). In this case study, classical B is used to model a single railway track. At its sides, signals are placed at various positions. The signals might allow the trains to pass through without interruption. However, in case a signal turns to red, the train has to stop within a certain area in front of the signal.

The model then places several trains on the track following each other. Each train can be assigned a speed. Obviously, there are several safety aspects the model has to care for, e.g., the trains may not be in the same section of the track. The goal is to trigger de- and acceleration of each train in order to find a configuration that minimizes breaking maneuvers and maximizes the throughput of the track.

The system consists of a machine called "params" that is used to set up necessary constants. It is accompanied by "actions" that includes variables and operations on them.

This time, the intended physical units of measurement were already given by the implementors of the machines. However, they were stored in simple comments and thus hidden from analysis.

We selected this case study, because it employs a feature not used in the former studies: user-defined units. The feature was originally added to our plugin to support non-standard units like the once used in railway applications.

To show that adding units does not slow down our analysis, we benchmarked two versions of "params" and "actions". In the first version, we use meters for every distance. In the second version, we introduced yards by adding the global pragma
`/*@ new_unit yards */`. Please note that, as explained in Sect. 3.2, this just introduces a new unit symbol "yards". There is no conversion rule between meters and yards[13].

Both variants were able to incorporate the units given by the implementors. Annotating was straightforward and no errors could be found using the analysis. There was no notable delay introduced by utilizing user-defined units.

7.6 TLA$^+$ Machines

The TLA$^+$ Clock model mentioned in Sect. 3.3 represents a simple clock storing the time in three integer variables used for hours, minutes and seconds. In this case study, we verify the consistent usage of units throughout the machine. The critical part is the `Clock_next` predicate.

It contains three conjuncts, each updating one of the variables:

- `seconds' = (seconds + 1) % 60` sets the new value of seconds. Following the inference rules explained in Sect. 4.1 the unit of `(seconds + 1) % 60` is equal to the unit of `(seconds + 1)` and therefore the unit of seconds. Units are used consistently.
- `hours' = (IF minutes # 59 THEN hours ELSE (hours + 1) % 24)` and
- `minutes' = (IF seconds # 59 THEN minutes ELSE (minutes + 1) % 60)` dispatches on a variable that uses a different unit than the variable that is updated. We will look at the translation to explain how our plugin evaluates this to be correct.

`minutes' = (IF seconds # 59 THEN minutes ELSE (minutes + 1) % 60)` is translated into the equivalent B expression $(\lambda t.(t \in \{1\} \land seconds \neq 59 | minutes) \cup (\lambda t.(t \in \{1\} \land \neg seconds \neq 59 | (minutes{+}1) \bmod 60)))(1)$ as explained in [18][14].

The analysis correctly computes that the unit of *minutes* and (*minutes*+1) mod 60 is the unit of *minutes* (min). Following, the abstract element representing the lambda expressions is set(pair(int($\perp_U$),int(min)), a set of pairs that each contain an untyped integer and an integer representing minutes. If we apply this to 1, an integer, the result is an integer representing minutes. This is compatible to the unit of the variable `minutes` and hence no unit error is found.

The benchmarks can be found in Tables 1 and 2, row 24. Time needed for the translation from TLA$^+$ to B is included in the analysis time and takes up most of the time spend. Lines of code and number of operations are counted on the translated machine.

Aside from the Clock model we performed a second benchmark on a simple water tank modeled in TLA$^+$. The model stores the amount of water in a tank, an upper and a lower threshold (all in m^3), a steady outflow and a pump controlled inflow (both in $\frac{m^3}{s}$). With the constant outflow, the water level sinks until the lower threshold is reached. This should trigger the pump, causing the water level to rise again. Upon other goals, the model should verify that a certain minimal amount of water is always present.

Annotating the model with physical units was again quite straight forward. This time, the length of a tick or an update operation was already encoded as a constant and set to one, much as we did in the Traffic Light Warning System. However, we encountered a unit error. The multiplication with the tick length was missing in the water level's update operation. As the duration of an update was set to a single second, the missing multiplication did not cause errors detectable by common model checking. Initializing the model with a different

[13] Upon request of our industrial customers we since added several custom units to the units included by default. Conversion rules were added as well.

[14] A relation constructed as the set union of the two lambda expressions. Depending on the condition of the if expression, one of the lambda expressions is empty while the other one contains exactly one element: the result of the if expression. We apply the relation to 1 to extract this element.

or even a nondeterministically chosen tick length would have made the error detectable by model checking.

We fixed the model in order to perform our benchmarks. The results are again presented above. Fixing the error took quite some time as the source-linked error information for TLA$^+$ is not as good as the one we implemented for B and Event-B. This will be improved in future versions.

8 Alternative Approaches and Related Work

Aside from the idea to use abstract interpretation, an extension of the type checking capabilities of ProB was initially considered. This approach would act more like a static analysis of the B machine, rather than interpreting it (abstractly) while observing the state space. Note, however, that simple syntactical unification-based type inference (Hindley-Milner style) is not powerful enough due to the generation of new units, e.g., during multiplication.

In order to use a unification-based algorithm the simple syntactic unification would have to be replaced by equational unification, i.e. unification with respect to a given theory [6]. Inside this theory, the computation of units under operations like multiplication and division could be axiomatized. Other operations like addition or subtraction could still be handled by using type variables where necessary. With the usual operations, the set of physical units forms an abelian group [12], over which equational unification is decidable [6]. Furthermore, most general unifiers can be computed for this theory and a unification algorithm can be implemented in a straight-forward fashion [21]. This technique is widely used in languages that support unit analysis.

In contrast to the interpreter based approach, implementing an extended type checker would possibly have resulted in less implementation work. On the downside, it would not be able to animate or to reason about intermediate states. In contrast, the interpreter based approach can also be used as an interactive aid while debugging errors.

A type checking approach for a modeling language is followed in [19]. The authors describe a language extension for Z adding physical units. The correct usage of units is verified by static analysis. Support for physical units is also present in the specification languages Modelica [29] and Charon [5]. While the Modelica standard supports physical units of measurement directly in the specification language, not all tools implementing the standard support all features of it. Furthermore, the standard does not describe how units should be inferred or which algorithms should be used.

Aside from specification languages, several extensions for general purpose language exist. Among others there are solutions for Lisp [15], C [20], C++ [37], Java [16] and F# [21].

In [21] and [38] the limitations of syntactical unification-based type inference are solved by inferring new units as the solutions of a system of linear equations as explained above. In our approach these equations can to some extent be found in the constraints mentioned in Sect. 5.

Another approach is followed in [34] and [30], providing an expressive type system containing physical units for Simulink, a modeling framework based on Matlab. The approach followed in [34] is different from the one implemented in this article and from the equational unification based algorithms. Instead of using abstract interpretation, the problem is translated into an SMT problem [8], which can be solved by a general purpose solver.

In addition to performing unit analysis, an SMT or constraint based approach could make it easier to generate test cases that verify the required properties. In particular, calculating the unsatisfiable core makes it possible to generate minimal test cases for certain errors. However, in contrast to the abstract interpreter based approach, verifying intermediate states and performing animation involve multiple reencodings of the problem to SMT-LIB.

In a recent followup [30], the authors present their new tool "DimSim" that is able to handle compositional analysis of Simulink as well as underspecification. In this version, the problem is translated into a set of constraints or equations solvable by Gauss-Jordan elimination. Hence, it can be seen as an implementation of equational unification.

9 Discussion and Conclusion

In conclusion, our approach of combining abstract interpretation with constraint solving works reasonably well. It is competitive with the equational unification based approaches mentioned in Sect. 8. The newly developed plugin extends ProB by the ability to perform unit analysis for formal models developed in B or Event-B. In connection with the translation from TLA$^+$ to B [18] it enables ProB to analyze units in TLA$^+$ machines. We provide source-level error feedback to the user and usually a small number of annotations is sufficient to infer the units of all variables and check the consistency of a machine. Screenshots of the tool integration showing input, output and error messages linked to the source code can be found in Figures 15, 16 and 17.

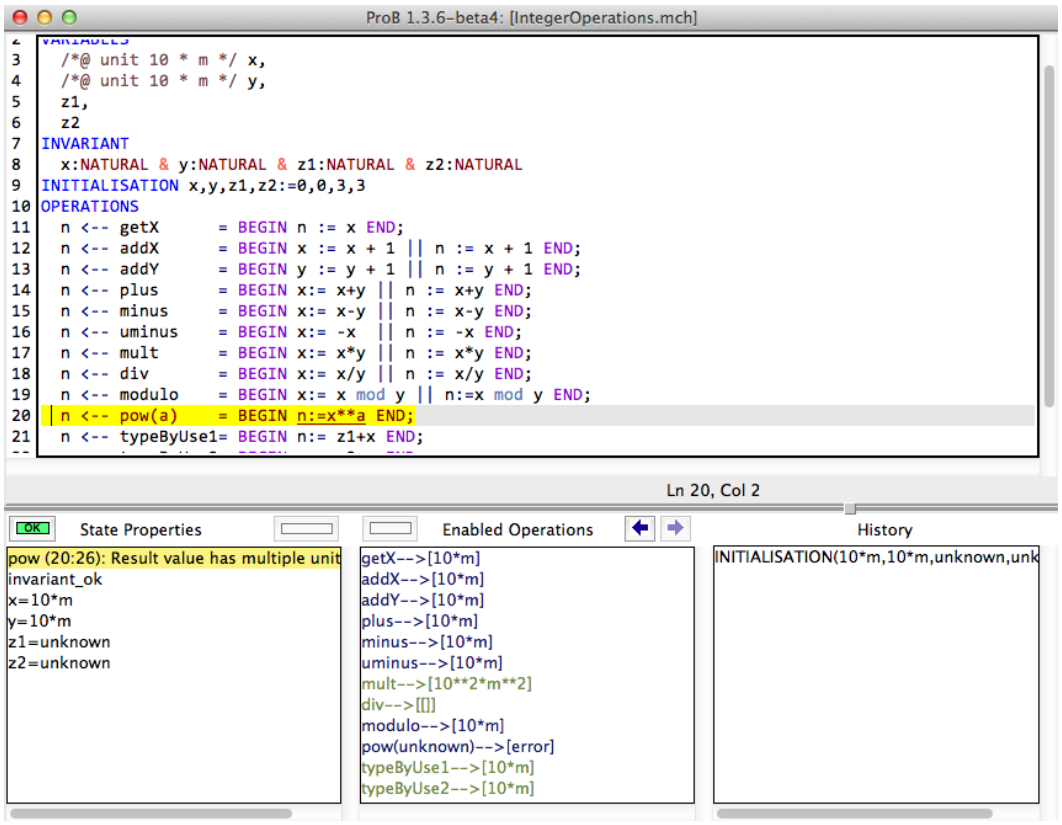

Fig. 15 Screenshot ProB with source-level feedback

Fig. 16 Rodin Integration: Variables and Units

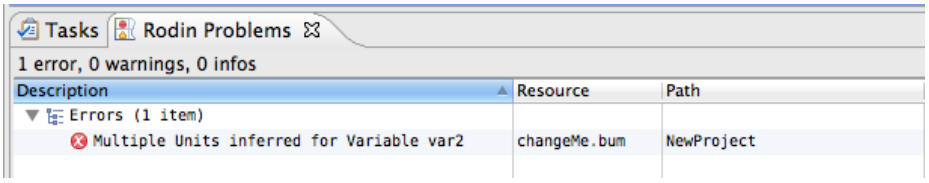

Fig. 17 Rodin Integration: Error Markers

Figure 15 shows the source-linked error reporting for a B model inside the Tcl/Tk based ProB standalone. Here, ProB reports that the return value of an operation has multiple units attached to it. Once the user clicks on the error message in the lower left panel, the corresponding source code lines are highlighted. Furthermore, the different units inferred can be seen as well.

Figure 16 shows how error reporting is done for an Event-B model inside the Rodin platform. We display both the annotated as well as the inferred unit. In case an error is detected, the incompatible units are presented to the user. Furthermore, the error reporting capabilities of Eclipse are used to attach error markers to different source code positions as shown in Fig. 17.

In future, we plan to support other languages, in particular CSP ‖ B and Z. Furthermore, the integration of TLA$^+$ is not as user-friendly as it is for B and Event-B. Improving error output and source-linked information is one of our next targets.

As anticipated, the plugin is able to infer units of constants and variables and handle their conversions. Additionally, user controlled unit conversions can be performed and are fully integrated with the analysis tools.

Furthermore, the extension of B and TLA$^+$ by pragmas leaves all machines usable by the different tools and tool sets without limitations. Deploying unit analysis does not interfere with any step of a user's usual workflow.

Most machines only needed a few iterations inside the fixpoint algorithm. Furthermore, the top element was only reached in machines containing errors. Thus, the selected abstract domain seems fitting for the desired analysis results.

While the overall performance generally matches the expectations, there is still room for improvement. Especially on large machines, computations should be refined. Yet, more input from industrial users is needed first, both in form of reviews and test reports as well as in form of case studies and sample machines.

We plan to further investigate the usage of constraints to speed up unit inference. In particular, an in-depth comparison with the SMT and constraint based approaches will be performed. This comparison will focus both on speed as well as on the completeness of the resulting unit information.

All in all, the unit analysis plugin extends the capabilities of B, TLA$^+$ and ProB and is a useful addition to the existing tools. It should be able to find errors which can not be discovered easily by the existing tools and might lead to errors in a future implementation. The technique scales to real-life examples and the animation capabilities aid in identifying the causes of errors.

Acknowledgements We are grateful to reviewers of SEFM and SoSyM for their useful feedback, which helped to improve the paper. Our thanks also go to Luis-Fernando Meija for providing us with interesting industrial case studies.

References

1. Abrial, J.R.: The B-Book. Cambridge University Press (1996). DOI 10.1017/CBO9780511624162
2. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
3. Abrial, J.R., Butler, M., Hallerstede, S.: An open extensible tool environment for Event-B. In: Z. Liu, J. He (eds.) Proceedings ICFEM'06, LNCS 4260, pp. 588–605. Springer-Verlag (2006). DOI 10.1007/s10009-010-0145-y
4. Abrial, J.R., Su, W., Zhu, H.: Formalizing hybrid systems with Event-B. In: Proceedings ABZ'12, LNCS 7316, pp. 178–193. Springer (2012)
5. Anand, M., Lee, I., Pappas, G., Sokolsky, O.: Unit & dynamic typing in hybrid systems modeling with CHARON. In: Computer Aided Control System Design, pp. 56–61. IEEE (2006)
6. Baader, F., Snyder, W.: Unification theory. Handbook of automated reasoning **1**, 445–532 (2001)
7. Back, R.J., Seceleanu, C.C., Westerholm, J.: Symbolic simulation of hybrid systems. In: Proceedings APSEC'02, pp. 147–155. IEEE Computer Society (2002)
8. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. Tech. rep., Department of Computer Science, University of Iowa (2010). Available at `www.SMT-LIB.org`
9. Boute, R.T.: The Euclidean Definition of the Functions Div and Mod. ACM Trans. Program. Lang. Syst. **14**(2), 127–144 (1992)
10. Bridgman, P.: Dimensional Analysis. Yale University Press (1922). URL `http://books.google.de/books?id=vehfnkmJIlkC`
11. ClearSy: Atelier B 4.1 Release Notes. Aix-en-Provence, France (2009). Available at `http://www.atelierb.eu/`
12. Collins, J.B.: A mathematical type for physical variables. In: S. Autexier, J. Campbell, J. Rubio, V. Sorge, M. Suzuki, F. Wiedijk (eds.) Intelligent Computer Mathematics, *Lecture Notes in Computer Science*, vol. 5144, pp. 370–381. Springer Berlin Heidelberg (2008)
13. Cousot, P.: Types as abstract interpretations. In: Conference Record of the Twentyfourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 316–331. ACM Press, New York, NY, Paris, France (1997)
14. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings POPL'77, pp. 238–252. ACM (1977)
15. Cunis, R.: A package for handling units of measure in Lisp. ACM SIGPLAN Lisp Pointers **5**, 21–25 (1992)
16. van Delft, A.: A Java extension with support for dimensions. Software: Practice and Experience **29**(7), 605–616 (1999)
17. Gibbings, J.: Dimensional Analysis. Springer London (2011)
18. Hansen, D., Leuschel, M.: Translating TLA+ to B for validation with ProB. In: Proceedings iFM'2012, LNCS 7321, pp. 24–38. Springer (2012)
19. Hayes, I.J., Mahony, B.P.: Using units of measurement in formal specifications. Formal Aspects of Computing **7** (1994)
20. Jiang, L., Su, Z.: Osprey: a practical type system for validating dimensional unit correctness of C programs. In: Proceedings ICSE'06, pp. 262–271. ACM (2006)
21. Kennedy, A.: Types for units-of-measure: Theory and practice. Central European Functional Programming School pp. 268–305 (2010)
22. Knuth, D.E.: The Art of Computer Programming, Volume 1: Fundamental Algorithms. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (1997)
23. Krings, S., Leuschel, M.: Inferring physical units in B models. In: Proceedings SEFM'2013, LNCS 8137. Springer (2013)

24. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
25. Lamport, L., Paulson, L.C.: Should your specification language be typed. ACM Trans. Program. Lang. Syst. **21**(3), 502–526 (1999)
26. Leuschel, M., Butler, M.: ProB: A model checker for B. In: Proceedings FME'03, LNCS 2805, pp. 855–874. Springer (2003)
27. Leuschel, M., Butler, M.: ProB: an automated analysis toolset for the B method. Int. J. Softw. Tools Technol. Transf. **10**(2), 185–203 (2008)
28. Lockwood, G.: Final Report of the Board of Injury: Investigating the Circumstances of an Accident Involving the Air Canada Boeing 767 Aircraft C-GAUN that Effected an Emergency Landing at Gimli, Manitoba on the 23rd Day of July, 1983. Minister of Supply and Services Canada (1985). URL `https://books.google.de/books?id=Ej5PAAAAMAAJ`
29. Modelica Association: The Modelica Language Specification version 3.0 (2007). URL `http://www.modelica.org/`
30. Owre, S., Saha, I., Shankar, N.: Automatic dimensional analysis of cyber-physical systems. In: Proceedings FM'12, LNCS 7436, pp. 356–371. Springer (2012)
31. Platzer, A.: Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics. Springer (2010)
32. Reps, T.W.: Program analysis via graph reachability. Information & Software Technology **40**(11-12), 701–726 (1998). DOI 10.1016/S0950-5849(98)00093-7. URL `http://dx.doi.org/10.1016/S0950-5849(98)00093-7`
33. Reps, T.W., Horwitz, S., Sagiv, S.: Precise interprocedural dataflow analysis via graph reachability. In: Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995, pp. 49–61 (1995). DOI 10.1145/199448.199462. URL `http://doi.acm.org/10.1145/199448.199462`
34. Roy, P., Shankar, N.: SimCheck: An expressive type system for Simulink. In: Proceedings NFM'10, pp. 149–160. NASA (2010)
35. Stephenson, A., LaPiana, L., Mulville, D., Rutledge, P., Bauer, F., Folta, D., Dukeman, G., Sackheim, R., Norvig, P.: Mars climate orbiter - mishap investigation report - phase i report (1999)
36. Thompson, A., Taylor, B.N.: The International System of Units (SI). Nist Special Publication (2008)
37. Umrigar, Z.: Fully static dimensional analysis with C++. ACM SIGPLAN Notices **29**(September), 135–139 (1994)
38. Wand, M., O'Keefe, P.: Automatic dimensional inference. Computational Logic: Essays in Honor of Alan Robinson pp. 479–483 (1991)