Inferring Physical Units in B Models

Sebastian Krings and Michael Leuschel

Institut für Informatik, Universität Düsseldorf**
Universitätsstr. 1, D-40225 Düsseldorf
sebastian.krings@uni-duesseldorf.de, leuschel@cs.uni-duesseldorf.de

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. In this paper we present a technique that analyses the usage of physical units throughout a B machine, infers missing units and notifies the user of incorrectly handled units. The technique combines abstract interpretation with classical animation 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. The plugin uses a combination of abstract interpretation and constraint solving techniques. We provide an empirical evaluation of our technique, and demonstrate that it 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 would have been 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.

In this paper we propose a solution to this problem, by integrating an abstract interpretation technique into the ProB animator [14,15]. More precisely:

** Part of this research has been sponsored by the EU funded FP7 project 287563 (ADVANCE).

[1] See, however, [13].

– we provide an abstract semantics for B, where 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.

An introductory example can be found in Figure 1. It contains an extract of a simple B machine modeling a car. The current speed and position are stored in two variables. 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, it leads to wrong results for the position of the car.

Analyzing the physical units of measurement, the error is easy to detect. Looking at the units of `speed` and `tick_length`, 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.

```
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

2 Inference of Physical Units

Below, in Section 2.1 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.

We will mainly use the international system of units (SI) [20], but a user can also declare additional non-SI units. Afterwards, how we use abstract interpretation will be explained in Section 2.2. Section 3 will explain why we had to improve the technique with constraint solving. Empirical results will be presented in Section 4. We conclude with alternative approaches and related work in Section 5 and a discussion of our results and future work in Section 6.

2.1 Syntactic Extension of the B Language

Initially, the user must provide the physical units for certain variables as a starting point of our analysis. 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[2]. 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 `*/`. (Atelier-B will treat such a pragma as an ordinary comment.)

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

1. "`unit`", the pragma used to attach a physical unit to a B construct. This can be done either by specifying it by a B expression in an SI-compatible form or by using a predefined alias like "cm" instead of "10**-2 * m". The given unit has to be a valid SI unit [20]; i.e., a derived unit such as "m * s**-2" is acceptable. The usage is shown in Figure 2.
2. "`inferred_unit`", which works similar to `unit`. It is included in the pretty print 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 Figure 3.
4. "`unit_alias`", used to define new aliases for existing unit definitions.

2.2 Using Abstract Interpretation

Inferring units of measurement has a strong connection to type checking, which can be seen as a special kind of abstract interpretation [8]. 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 Figure 4. Initially, any type is still possible,

[2] Screenshots of input, output and errors messages can be found on `http://www.stups.uni-duesseldorf.de/models/sefm2013/screenshots`.

```
MACHINE UnitExample
VARIABLES
 /*@ unit 10 * m */ x,
 y
INVARIANT x:NAT & y:NAT & x>y
INITIALISATION x,y := 0,0
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

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:

- $\bot \in Types$
- $Bool \in Types$
- $String \in Types$
- $\mathbb{Z} \in Types$
- $Given \subseteq Types$ where $Given$ contains all the user-defined deferred, enumerated or parameter sets
- $x \in Types \land y \in Types \Rightarrow x \times y \in Types$
- $t \in Types \Rightarrow \mathcal{P}(t) \in Types$ [3]

Furthermore, if multiple types are inferred, there is a type error. This is denoted by the special type $\top$[4]. We define $Types_\top = Types \cup \{\top\}$. Note, that for Event-B, the Rodin tool also generates a type error if the inferred type still contains $\bot$. This can occur for a predicate such as $\{\} = \{\}$, where the type of $\{\}$ would be inferred as $\mathcal{P}(\bot)$.

[3] Functions and relations are stored as sets of couples.

[4] As the top element represents the least upper bound that matches two different types. However, only one type is acceptable for a correct model.

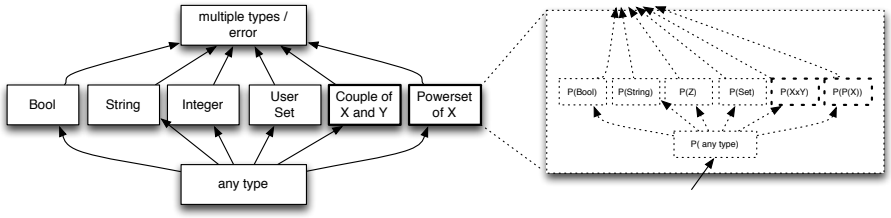

Fig. 4. B Type System and the relation $\sqsubset$

Basically, types are ordered using the relation $\sqsubseteq$, forming the lattice in Fig. 4 and defined using the following five rules. (We define $\sqsubset$ in the usual way: $s \sqsubset t$ iff $s \sqsubseteq t \wedge s \neq t$.)

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

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 couples remain, the integer type is replaced by an entire subdomain. An abstract integer value is now 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, u a SI base unit symbol and $e \in \mathbb{Z}$ the exponent of the unit.[5]

Definition 1. *A* unit *is a set of triples* $\{[10^{c_1} \times {u_1}^{e_1}], \ldots, [10^{c_k} \times {u_k}^{e_k}]\}$ *such that for all* $i \in 1..k$ *we have* $c_i \in \mathbb{Z}$, $e_i \in \mathbb{Z}$, u_i *being a base SI unit and* $\forall j \bullet j \in 1..k \wedge j \neq i \Rightarrow u_i \neq u_j$.

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.

Definition 2. *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.

Summarizing, our abstract interpretation framework for B uses the set of all possible B values as the concrete domain $\mathcal{C}$ and maps it to the abstract domain $\mathcal{A}$, which is recursively defined by

[5] For convenience, some SI derived units and units accepted for use with the SI standard (see [20]) are stored on their own rather than converting them.

- $boolean \in \mathcal{A}$
- $string \in \mathcal{A}$
- $\forall u \in Units \cup \{\bot_U, \top_U\} \Rightarrow \text{int}(u) \in \mathcal{A}$
- $\forall S \in Given, u \in Units \cup \{\bot_U, \top_U\} \Rightarrow \text{set}(S, u) \in \mathcal{A}$
- $x \in \mathcal{A} \land y \in \mathcal{A} \Rightarrow \text{couple}(x, y) \in \mathcal{A}$
- $t \in \mathcal{A} \Rightarrow \text{set}(t) \in \mathcal{A}$.

Note, that we need both $set(t)$ and $set(t, u)$: While the first is a set with elements that may hold a unit themselves, i.e. a set integers, the second has a unit directly attached to it, i.e. an enumerated set. The rules for the ordering of abstract values are as follows:

- $\bot_U \sqsubseteq_U u$ for any $u \in Units \cup \{\top_U\}$
- $u \sqsubseteq_U \top_U$ for any $u \in Units \cup \{\bot_U\}$
- $\bot_U \sqsubseteq_U \top_U$
- $t \sqsubseteq t$ 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 $u \sqsubseteq u'$
- $set(t, u) \sqsubseteq set(t, u')$ iff $u \sqsubseteq u'$
- $couple(s, t) \sqsubseteq couple(s', t')$ iff $s \sqsubseteq s' \land t \sqsubseteq t'$
- $set(t) \sqsubseteq set(t')$ iff $t \sqsubseteq t'$

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:

$$\alpha(x) = \begin{cases} boolean & \text{if } x \in \{\text{true}, \text{false}\} \\ string & \text{if type of } x \text{ is string} \\ int(unit) & \text{if } x \in \mathbb{Z} \text{ with an annotated unit} \\ int(\bot_U) & \text{if } x \in \mathbb{Z} \text{ without an annotated unit} \\ set(S, unit) & \text{if } x \in S, \text{ S annotated with unit} \\ set(S, \bot_U) & \text{if } x \in S, \text{ S without an annotated unit} \\ couple(\alpha(x_1), \alpha(x_2)) & \text{if } x \text{ is of type } x_1 \times x_2 \\ set(\alpha(x_1)) & \text{if } x \in \mathcal{P}(x_1) \end{cases}$$

with $S \in Given$ and

$$\gamma(y) = \begin{cases} \{\text{true}, \text{false}\} & \text{if } y = boolean \\ \{s | \text{type of s is string}\} & \text{if } y = string \\ S & \text{if } y = set(S, unit), \text{ with any unit} \\ \mathbb{Z} & \text{if } y = int(unit), \text{ with any unit} \\ \gamma(y_1) \times \gamma(y_2) & \text{if } y = couple(y_1, y_2) \\ \mathcal{P}(\gamma(y_1)) & \text{if } y = set(y_1). \end{cases}$$

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 couples.
3. Instructions like multiplication or division are able to generate new units based on the units of their operands.

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
    add $[10^{c_1} \times u^{e_1}]$ to p
  end
end
$p := p \cup x_2$

Algorithm 1: Abstract Multiplication

The first and second kind of operations can be implemented by unification or returning $\top$ if incompatible units are found. This 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 (unification-based) type inference. On the representation outlined above, multiplication is implemented by addition of the exponents of triples holding the same unit symbol. See Algorithm 1 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, in particular 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 as

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

Consequently, for the unit of $n \bmod m$

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

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 are certainly possible. Up to now, our empirical evaluation did not reveal any problems with the given definition.

We perform a fixpoint search by executing all operations of a B machine. Additionally, we evaluate properties and invariants in every iteration. Pseudocode can be found in Algorithm 2. For the example machine in Figure 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, $\perp_U$ is used. In the example, we set the initial state σ_0 to $\{(x, int(\{[10^1 \times m^1]\})), (y, int(\perp_U))\}$.
2. Evaluate the invariant on σ_0. The predicate $x > y$ allows us to infer the unit of y, updating $\sigma_0 = \{(x, int(\{[10^1 \times m^1]\})), (y, int(\{[10^1 \times m^1]\}))\}$. No incorrect usage of units is detected.
3. Execute `addToX` on σ_0:
   (a) Generate local state $\sigma_{IN} = \{(n, int(\perp_U)\} \cup \sigma_0$.
   (b) Evaluate $x + y = int(\{[10^1 \times m^1]\})) + int(\{[10^1 \times m^1]\})) = int(\{[10^1 \times m^1]\}))$
   (c) Substitute n by calculating the least upper bound of $\perp_U$ and $int(\{[10^1 \times m^1]\}))$. The resulting output state is $\sigma_{OUT} = \{(n, int(\{[10^1 \times m^1]\})), (x, int(\{[10^1 \times m^1]\})), (y, int(\{[10^1 \times m^1]\}))\}$.
4. Again, no incorrect usage of units is detected.
5. The next iteration executes `addToX` a second time. However, the state does not change and the fixpoint is reached.

3 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 Figure 5, where the variable x contains a length in meters and t holds a time interval in seconds. The unit of y should be inferred. Evaluating the expression $t := (x * y) * t$ needs several interpretation steps:

1. The interpreter computes the product of x and y. As $y = int(\perp_U)$, the interpreter can only return $int(\perp_U)$ as a result.
2. In consequence, the interpreter finds that $(x*y)*t = int(\perp_U)*t = int(\perp_U)* int(\{[10^0 \times s^1]\}) = int(\perp_U)$.
3. The assignment $t := (x * y) * t$ is evaluated by computing the least upper bound of t and $int(\perp_U)$, i.e., $int(\{[10^0 \times m^1]\})$. 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 $\perp_U$. Inside, the operation and the operands that lead to $\perp_U$ are stored. We implemented constraints for multiplication, division and exponentiation, as those can not be handled by unification alone.

In the example given in Fig. 5 two constraints are used to infer the unit of variable y. First, a constraint containing x and y is attached to the result of the inner multiplication. The result of the outer multiplication is annotated in

```
σ = {(identifier of x, α(x)) : x variable or constant}
evaluate properties / invariant (might replace ⊥_U by units in σ)
repeat
    foreach operation / event do
        update σ by executing operation / event:
            evaluate preconditions / guards (might replace ⊥_U by units 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 units in σ)
        if σ contains ⊤_U then
            report error
        end
        if invalid unit usage detected then
            report error
        end
        foreach variable holding a constraint do
            evaluate constraint if possible
        end
    end
until σ did not change in loop
```

Algorithm 2: Fixpoint Search

the same way. When computing the assignment to t, we know the unit that the outer multiplication has to return. We can use the domain operation for division of units to reverse the multiplications and compute the value of y.

In general, once a variable with an attached constraint is unified with another variable, the unit plugin has different ways to react:

- The other variable does not hold a physical unit at the moment. Hence, we can not solve the constraint.
- The other variable contains a physical unit. Now, we have to look at the variables inside:
  - If both variables are currently unset, there are multiple possible solutions. We again delay the computation to the next iteration of the fixpoint algorithm.
  - If one of the variables is unknown and the other one contains a unit, we can compute the missing unit.
  - If both are set, the constraint is dropped without further verification.
- Further unifications in the second step may trigger this process on another variable.

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, the constraint could only detect an error already reported. See Algorithm 2 for details.

```
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. 5. Machine requiring involved constraint solving

4 Empirical Results

Our empirical evaluation was based on three key aspects:

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

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[6].

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.

[6] The machines used in this case study can be downloaded from `http://www.stups.uni-duesseldorf.de/models/sefm2013/`.

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 Table 1. For comparison purposes, the table also lists the number of lines of code and the number of operations for each machine[7]. 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.

The second case study used a ClearSy tutorial on modeling in B[8]. 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 other units could be inferred[9]. In the process, no unit reached $\top_U$. The benchmarks are presented in Table 1, 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.

To evaluate the performance of the unit plugin on large scale examples, several B railway models from Alstom were used as benchmarks. 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.

---

[7] 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.

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

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

Currently, 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 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.

As a last 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 [6]. 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 [18]. 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 14 to 19 in Table 1.

5 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 classical

Table 1. Benchmarks

No.	machine	LOC	operations	iterations	time analysis	time annotating
1	Car	74	4	2	< 10 ms	≈ 30 min
2	TrafficLight	81	2	1	< 10 ms	≈ 20 min
3	System	322	20	2	50 ms	≈ 60 min
4	measure	42	2	1	< 10 ms	≈ 5 min
5	utils	24	2	1	< 10 ms	≈ 5 min
6	utils_i	38	2	1	< 10 ms	≈ 5 min
7	ctx	16	0	1	< 10 ms	≈ 5 min
8	ctx_i	16	0	1	< 10 ms	≈ 5 min
9	fuel0	64	2	2	< 10 ms	≈ 5 min
10	fuel_i	106	6	2	< 10 ms	≈ 5 min
11	compensated_gradient	3079	20	3	620 ms	≈ 45 min
12	vital_gradient	986	4	3	160 ms	≈ 45 min
13	sgd	773	0	2	170 ms	≈ 90 min
14	T_m0	115	6	3	20 ms	≈ 15 min
15	T_m1	179	11	3	30 ms	≈ 15 min
16	C_m0	108	4	3	20 ms	≈ 15 min
17	C_m1	141	4	3	20 ms	≈ 15 min
18	C_m2	162	4	3	40 ms	≈ 15 min
19	C_m3	228	7	3	90 ms	≈ 15 min

unification-based type inference (Hindley-Milner style) is not powerful enough due to the generation of new units, e.g., during multiplication.

A type checking approach for a modeling language is followed in [10]. 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 [16] and Charon [5].

Aside from specification languages, several extensions for general purpose language exist. Among others there are solutions for Lisp [9], C [11], C++ [21], Java [22] and F# [12].

In [12] and [23] the limitations of unification-based type inference mentioned above are solved by inferring new units as the solutions of a system of linear equations. In our approach these equations can be found in the constraints mentioned in Section 3.

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.

Another approach is followed in [19] and [17], providing an expressive type system containing physical units for Simulink, a modeling framework based on Matlab. The approach followed in [19] differs from the one implemented in this

paper. Instead of using abstract interpretation, the problem is translated into an SMT problem [7], which can be solved by a general purpose solver. In the approach presented in [17], the SMT backend is replaced by a set of constraints solvable by Gauss-Jordan elimination.

In addition to performing unit analysis, an SMT or constraint based approach makes 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.

6 Discussion and Conclusion

In conclusion, our first set of goals could be fulfilled. The newly developed plugin extends ProB by the ability to perform unit analysis for formal models developed in B or Event-B. 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. In future, we plan to support other languages, in particular TLA^+ and Z.

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 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 B 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 and ProB and is a useful addition to the existing B tools. It should be able to find errors which are not easily discoverable 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.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

2. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
3. J.-R. Abrial, M. Butler, and S. Hallerstede. An open extensible tool environment for Event-B. In *Proceedings ICFEM'06*, LNCS 4260, pages 588–605. Springer, 2006.
4. J.-R. Abrial, W. Su, and H. Zhu. Formalizing hybrid systems with Event-B. In *Proceedings ABZ'12*, LNCS 7316, pages 178–193. Springer, 2012.
5. M. Anand, I. Lee, G. Pappas, and O. Sokolsky. Unit & dynamic typing in hybrid systems modeling with CHARON. In *Computer Aided Control System Design*, pages 56–61. IEEE, 2006.
6. R.-J. Back, C. C. Seceleanu, and J. Westerholm. Symbolic simulation of hybrid systems. In *Proceedings APSEC'02*, pages 147–155. IEEE Computer Society, 2002.
7. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, University of Iowa, 2010. Available at `www.SMT-LIB.org`.
8. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings POPL'77*, pages 238–252. ACM, 1977.
9. R. Cunis. A package for handling units of measure in Lisp. *ACM SIGPLAN Lisp Pointers*, 5:21–25, 1992.
10. I. J. Hayes and B. P. Mahony. Using units of measurement in formal specifications. *Formal Aspects of Computing*, 7, 1994.
11. L. Jiang and Z. Su. Osprey: a practical type system for validating dimensional unit correctness of C programs. In *Proceedings ICSE'06*, pages 262–271. ACM, 2006.
12. A. Kennedy. Types for units-of-measure: Theory and practice. *Central European Functional Programming School*, pages 268–305, 2010.
13. L. Lamport and L. C. Paulson. Should your specification language be typed. *ACM Trans. Program. Lang. Syst.*, 21(3):502–526, 1999.
14. M. Leuschel and M. Butler. ProB: A model checker for B. In *Proceedings FME'03*, LNCS 2805, pages 855–874. Springer, 2003.
15. M. Leuschel and M. Butler. ProB: an automated analysis toolset for the B method. *Int. J. Softw. Tools Technol. Transf.*, 10(2):185–203, Feb. 2008.
16. Modelica Association. The Modelica Language Specification version 3.0, 2007.
17. S. Owre, I. Saha, and N. Shankar. Automatic dimensional analysis of cyber-physical systems. In *Proceedings FM'12*, LNCS 7436, pages 356–371. Springer, 2012.
18. A. Platzer. *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Springer, 2010.
19. P. Roy and N. Shankar. SimCheck: An expressive type system for Simulink. In *Proceedings NFM'10*, pages 149–160. NASA, 2010.
20. A. Thompson and B. N. Taylor. The International System of Units (SI). *Nist Special Publication*, 2008.
21. Z. Umrigar. Fully static dimensional analysis with C++. *ACM SIGPLAN Notices*, 29(September):135–139, 1994.
22. A. van Delft. A java extension with support for dimensions. *Software: Practice and Experience*, 29(7):605–616, 1999.
23. M. Wand and P. O'Keefe. Automatic dimensional inference. *Computational Logic: Essays in Honor of Alan Robinson*, pages 479–483, 1991.