Fast and Effective Well-Definedness Checking

Michael Leuschel

Institut für Informatik, Universität Düsseldorf
Universitätsstr. 1, D-40225 Düsseldorf
michael.leuschel@hhu.de

Abstract. Well-Definedness is important for many formal methods. In B and Event-B it ensures that certain kinds of errors (e.g., division by 0) cannot appear and that proof rules based on two-valued logic are sound. For validation tools such as ProB, well-definedness is important for constraint solving. B and Event-B establish well-definedness by generating dedicated proof obligations (POs). Unfortunately, the standard provers are not always very good at discharging them. In this paper, we present a new integrated technique to simultaneously generate and discharge well-definedness POs. The implementation contains a dedicated rule-based prover written in Prolog and supports B, Event-B and extensions thereof for data validation. We show that the generation and discharging is significantly faster than existing implementations in RODIN and Atelier-B and we show that a large number of POs are automatically discharged. The POs are fine-grained enough to provide precise source code feedback, and allow inspection of problematic POs within various editors.

1 Introduction and Motivation

Well-definedness is an important issue in formal methods. Various approaches exist to dealing with ill-defined expressions such as a division by zero or a function applied outside of its domain.

Three-valued logic is one such approach, but is rarely used in practice. Indeed, some important proof rules or techniques (e.g., proof by contradiction) are not sound in three-valued logic. This famous quote by the mathematician Hilbert is also relevant for automated provers: "*Taking the principle of excluded middle from the mathematician would be the same, say, as proscribing the telescope to the astronomer or to the boxer the use of his fists. To prohibit existence statements and the principle of excluded middle is tantamount to relinquishing the science of mathematics altogether.*".[1]

Another approach is to only allow total functions, but preclude any knowledge about the function's value for problematic inputs. E.g., in the case of division the expression $1/0$ would denote a number, but one has no knowledge about its value within a proof. In this approach we thus can neither prove $1/0 = 0$ nor $1/0 \neq 0$, but we can prove the predicates $1/0 = 1/0$ or $\exists y.y = 1/0$. This approach is convenient for constraint solving or model finding and is typically used

[1] Taken from https://en.wikipedia.org/wiki/Brouwer?Hilbert_controversy.

by SMTLib or Why3 [21]. Its main drawback is that problematic expressions such as 1/0 may lurk within a formal model without a user realising it.

The approach employed by the B-method is to generate well-definedness (WD) proof obligations (POs) [5] for all formulas under consideration. These POs can be generated by the RODIN platform [3] or by ATELIER-B (where they are called WD-lemmas). If they are discharged we know that the corresponding formulas are well-defined and that we can apply two-valued logic. This approach is good for automated proving, enabling to apply effective provers based on two-valued logic. Also, for users it is good to obtain feedback about ill-defined expressions, rather than silently giving them an arbitrary value. A false WD proof obligation pinpoints a potential error in formal model. There are unfortunately a few outstanding practical issues:

– discharging the WD proof obligations themselves is often quite time consuming, and the built-in provers of RODIN or ATELIER-B are not very good at discharging certain types of relevant goals (e.g., finiteness proof obligations or boundedness proof obligations for *min* and *max*).
– The POs generated by RODIN or ATELIER-B apply to entire formulas (e.g., invariants or guards), and it would be useful to be able to more precisely pinpoint problematic expressions and operators in the formal models. This is useful for user feedback, e.g., in an editor. Also, for constraint solving, ill-defined expressions pose a particular threat and also reporting well-definedness errors is very difficult. Here a precise annotation can help the constraint solver in knowing, e.g., which division is susceptible to divide by zero. We return to this in Section 6 below.
– for data validation [24], well-definedness is also an important issue. However, RODIN is missing some datatypes such as strings and sequences (the latter can be added via the theory plugin; but automated proof support is very limited). While ATELIER-B supports sequences and strings, its automated proof support for sequences is not very good. Furthermore, in practice, we use a few extensions to B which are not (yet) supported by ATELIER-B, such as the if-then-else and let for constructs expressions (see [17]).

Contributions In this article we present a new combined well-definedness PO generator and prover, which has been integrated into the PROB validation tool, and which

– is based on a fast algorithm to generate WD proof obligations and discharge them at the same time,
– deals with Event-B, classical B and PROB's extensions for data validation (or with any other formalisms which PROB translates internally to B, namely TLA+, Alloy and Z),
– provides proof support for the B sequence datatype and its many operators,
– produces precise error feedback, either in PROB's own editor, Atom or VS-Code to the end user,
– and can provide precise annotation of those operator nodes in the abstract syntax tree of a formal model which are susceptible to well-definedness errors.

Our prover can be seen as a specialized successor to the ML (mono-lemma) rule-based prover from ATELIER-B, dedicated to discharging WD POs.

2 Well-Definedness Proof-Obligations

We first recall the essential aspects of the well-definedness proof-obligations, as described in [5]. We suppose some familiarity with the B method. By *formula* we here mean either an expression (e.g, $x + 1$), a predicate (e.g., $x > 1$) or a substitution (e.g., $x := 1$). We denote logical equivalence of predicates by $\equiv$.

With each formula f we associate a well-definedness predicate $WD(f)$. $WD(f)$ is defined inductively over the structure of f and can be seen as a syntactic operator: it maps a formula f to a predicate. For Event-B the rules can be found in Section 5 of [27] in the RODIN handbookor partially in section 5.2.12 of [2].

Here are two such rule for division and function application, where the type of f is $\mathbb{P}(D_f \times R_f)$:

$$WD(a \div b) \equiv WD(a) \land WD(b) \land b \neq 0$$
$$WD(f(a)) \equiv a \in dom(f) \land f \in D_f \nrightarrow R_f$$

Here is a generic rule for binary operators $\circ$ which do not have their own WD condition; they only require their arguments to be well-defined for the entire formula to be well-defined:

$$WD(a \circ b) \equiv WD(a) \land WD(b)$$

Similarly, an integer literal or a variable has no well-definedness condition, i.e., the WD predicate is true:

$$WD(x) \equiv \top \quad \text{for integer literals or variables x}$$

We thus have for example $WD(10 \div (x \div y)) \equiv y \neq 0 \land x \div y \neq 0$.

$\mathcal{L}$ and $\mathcal{D}$ and Connectives An important aspect arises is the treatment of the logical connectives. There are in fact two approaches [5, 8] to computing WD:

- the left-to-right approach $\mathcal{L}$ which requires that well-definedness of a formula must be established by predicates on its left,
- and the more flexible $\mathcal{D}$ approach, which does not impose a strict left-to-right examination of the predicates.

RODIN uses the $\mathcal{L}$ approach, meaning that:

$$WD(P \land Q) \equiv WD(P) \land (P \Rightarrow WD(Q))$$

In other words, Q must not be well-defined if P is false. Similarly,

$$WD(P \lor Q) \equiv WD(P) \land (P \lor WD(Q))$$

Note that the RODIN handbook uses $\mathcal{L}(.)$ to denote this left-to-right WD-condition. [8] uses Δ_P^{MC}, where MC stands for Mc Carthy (see also [9] for CVC-lite).

The more powerful $\mathcal{D}$ approach [5] uses the following rule

$$\mathcal{D}(P \wedge Q) \equiv (\mathcal{D}(P) \wedge \mathcal{D}(Q)) \vee (\mathcal{D}(P) \wedge \neg P) \vee (\mathcal{D}(Q) \wedge \neg Q)$$

In [8] this operator is written as Δ_P^K instead, where K stands for Kleene.

Given $P = (x > 0 \wedge 100/x < 50)$ and $P' = (100/x < 50 \wedge x > 0)$ we have that $WD(P) \equiv \top$ and $WD(P') \equiv (x \neq 0) \not\equiv \top$ but $\mathcal{D}(P) \equiv \mathcal{D}(P') \equiv \top$. The $\mathcal{D}$ approach is more powerful, and is commutative wrt $\wedge$ and $\vee$ but suffers from an exponential blowup of the size of the WD proof obligations. It is not used in practice.[2] We will also use the $\mathcal{L}$ approach in the remainder of this article.

B vs Event-B For classical B the conditions are associated with each operator in [1] or the ATELIER-B handbook [12]. The treatment of substitutions is handled in [8]. There are some subtle differences in the WD conditions of B and Event-B. E.g., exponentiation is less permissive in Event-B than inclassical B: $(-2)^3$ is allowed in classical B, but not well-defined in Event-B (cf, page 43, Table 5.2 in [27]). For modulo `-3 mod 2 =-1` is well-defined and true in Event-B, but is not well-defined in classical B. But this is not due to a difference in the WD condition, but due to the fact that `-3 mod 2` is parsed as `-(3 mod 2)` in RODIN and `(-3) mod 2` in ATELIER-B.[3]

There is, however, no fundamental difference in the derivation of the WD proof obligations for predicates and expressions (but classical B has many more substitutions, see Sect. 3). Our implementation has a flag indicating the language mode (B, Event-B, Z, Alloy or TLA+), to appropriately adapt the POs.

There is, however, one fundamental difference between RODIN and ATELIER-B. RODIN adds the goals of WD proof obligations to subsequent proof obligations. The motivation is to avoid having to re-prove the same goal multiple times. This technique is not described in the [2, 27], but can be found in [26]. In ATELIER-B this technique is not applied.

In the example below, we have that the WD PO for axiom `axm2` in RODIN is $f \in \mathbb{Z} \leftrightarrow \mathbb{Z} \Rightarrow f \in \mathbb{Z} \rightarrow\!\!\!\!\!+ \mathbb{Z} \wedge 2 \in dom(f)$. This PO cannot be proven (and the model contains a WD error), but its goals $f \in \mathbb{Z} \rightarrow\!\!\!\!\!+ \mathbb{Z}$ and $2 \in dom(f)$ are added by RODIN to the hypotheses of the PO for the theorem `thm1`, meaning that it can be trivially proven with the `hyp` rule (which checks if a hypothesis is on the stack).

```
1 context Test_WD_Hyp
2 constants f
3 axioms
```

[2] [13] discusses combining power of $\mathcal{D}$ with efficiency of $\mathcal{L}$, but is not used in practice as far as we know. It seems to require one to establish the truth or falsity of individual formulas, which may not be easily feasible in practice.

[3] The RODIN handbook requires modulo arguments to be non-negative, which is correct; [27] is in error.

```
4     @axm1 f : INT <-> INT // f is a relation
5     @axm2 f(2) = 3        // this can give rise to a WD error
6     theorem @thm1 f : INT +-> INT // can be proven with hyp in Rodin
7     theorem @thm2 f(2) = 4
8   end
```

Listing 1.1. WD Event-B Rodin Example

This optimisation also means that discharging all *WD* POs is very important in RODIN: one simple error like using f(2) can be used to prove arbitrary goals (e.g., above one can easily prove a theorem `22=33`).

In our algorithm we do not use this optimisation of RODIN. As we will see, our algorithm is fast enough without it, and we also want to establish well-definedness for each program point in isolation and detect all sub-expressions which are potentially ill-defined, not just the first one. For example, in RODIN the well-definedness PO of theorem `thm2` is proven. This is particularly relevant when we want to use the information for a constraint solver: it has to know for every program point whether it is guaranteed to be well-defined or not.

3 Fine-grained WD Proof Obligations

Below we define our more fine grained way of computing well-definedness proof obligations. Rather than computing one proof obligation for an entire formula (such as an invariant or axiom), we will derive *multiple* proof obligations for individual operators within each formula. Our formalization thus uses a relation rather than a function taking a formula and producing a single PO. Our formalization also manages explicitly a single hypothesis environment, rather than putting hypotheses piecemeal into the formulas. The reasons will become apparent later: our formalisation manages the hypotheses like a *stack* and will correspond to an efficient implementation in Prolog.

Our PO generation is formalised using the following ternary relation: $H \otimes F \twoheadrightarrow P$ means that given the current hypotheses H the formula F gives rise to a proof obligation P. P will always be a predicate of the form $\mathit{Hypotheses} \Rightarrow \mathit{Goal}$.

For example, we will have that:

- $H \otimes (10 \div b) \div c \twoheadrightarrow H \Rightarrow b \neq 0$ and
- $H \otimes (10 \div b) \div c \twoheadrightarrow H \Rightarrow c \neq 0$.

We will first provide a generic rule for all binary operators (such as $\div$, $+$, $\cup$) which always require *both* arguments to be well-defined without additional hypotheses. We first define the direct well-definedness condition *WDC* for every such operator, ignoring *WD* conditions of arguments. Here are a few rules, where the type of f is $\mathbb{P}(D_f \times R_f)$:

$$\mathit{WDC}(a \div b) \equiv b \neq 0$$

$$\mathit{WDC}(a + b) \equiv \top$$

$$\mathit{WDC}(f(a)) \equiv a \in \mathit{dom}(f) \wedge f \in D_f \mathrel{+\!\!\!\rightarrow} R_f$$

$$WDC(\mathit{first}(f)) \equiv f \in \mathit{seq}(R_f) \land f \neq \varnothing$$

$$WDC(\mathit{inter}(a)) \equiv a \neq \varnothing$$

We can now provide three generic inference rules for all those binary operators BOP where no hypotheses are added or removed for discharging the well-definedness of its arguments:

$$\frac{}{H \otimes a \circ b \ \twoheadrightarrow\ H \Rightarrow WDC(a \circ b)} \ \circ \in BOP$$

$$\frac{H \otimes a \ \twoheadrightarrow\ PO}{H \otimes a \circ b \ \twoheadrightarrow\ PO} \ \circ \in BOP \qquad \frac{H \otimes b \ \twoheadrightarrow\ PO}{H \otimes a \circ b \ \twoheadrightarrow\ PO} \ \circ \in BOP$$

For unary operators UOP such as -, union, inter, conc we have the following rules:

$$\frac{}{H \otimes \circ a \ \twoheadrightarrow\ H \Rightarrow WDC(\circ a)} \ \circ \in UOP$$

$$\frac{H \otimes a \ \twoheadrightarrow\ PO}{H \otimes \circ a \ \twoheadrightarrow\ PO} \ \circ \in UOP$$

The logical connectives are now dealt with as follows. The equivalence $\Leftrightarrow$ can simply be treated by the BOP inference rules with $WDC(P \Leftrightarrow Q) = \top$. Similarly, negation can be treated as a unary operator with $WDC(\neg P) = \top$.

Let us now deal with conjunction. Here we can observe that the predicate P is pushed onto the hypotheses H for the second argument Q:

$$\frac{H \otimes P \ \twoheadrightarrow\ PO}{H \otimes P \land Q \ \twoheadrightarrow\ PO} \qquad \frac{H \land P \otimes Q \ \twoheadrightarrow\ PO}{H \otimes P \land Q \ \twoheadrightarrow\ PO}$$

The implication has exactly the same inference rules:

$$\frac{H \otimes P \ \twoheadrightarrow\ PO}{H \otimes P \Rightarrow Q \ \twoheadrightarrow\ PO} \qquad \frac{H \land P \otimes Q \ \twoheadrightarrow\ PO}{H \otimes P \Rightarrow Q \ \twoheadrightarrow\ PO}$$

For disjunction the negation of P is pushed onto the hypotheses for Q:

$$\frac{H \otimes P \ \twoheadrightarrow\ PO}{H \otimes P \lor Q \ \twoheadrightarrow\ PO} \qquad \frac{H \land \neg P \otimes Q \ \twoheadrightarrow\ PO}{H \otimes P \lor Q \ \twoheadrightarrow\ PO}$$

Here we clearly see the difference with the classical formalization of the WD operator in the literature, which inserts a hypothesis into a disjunction of the resulting PO formula: $WD(P \lor Q) \equiv WD(P) \land (P \lor WD(Q))$.

The treatment of quantifiers requires the renaming operator $\rho_V(H)$ which renames all variables in the Hypotheses which clash with variable in V to fresh new variables. With this operator we can produce the rules for existential and universal quantification:

$$\frac{\rho_V(H) \otimes P \ \twoheadrightarrow\ PO}{H \otimes \exists V.P \ \twoheadrightarrow\ PO}$$

$$\frac{\rho_V(H) \otimes P \Rightarrow Q \ \twoheadrightarrow \ PO}{H \otimes \forall V.P \Rightarrow Q \ \twoheadrightarrow \ PO}$$

We have similar rules for other quantified operators, such as $\bigcup$ or $\bigcap$.

Proof Obligations for Substitutions The Atelier-B handbook does not detail how well-definedness is established for substitutions, and this aspect is not relevant in rodin. For our tool we first developed our own *WD* proof rules and then discovered that [8] contains *WD* rules for substitutions, which seem mostly to have been taken over in Atelier-B. Some constructs like parallel composition or CHOICE are simply "transparent" for *WD* computation and can be treated in the same way as the binary operators above. This means that in a parallel construct $P \parallel Q$ each branch must be well-defined on its own: one cannot make use of guards in P to prove $WD(Q)$ or vice-versa. Some examples in Sect. 5.3 incorrectly rely on the guards in P to establish well-definedness in Q.

Some constructs like IF-THEN-ELSE or SELECT are similar to conjunction in that hypotheses are added for certain subgoals.

For the assignment, B also allows to assign to functions and nested functions and to records and nested records. Here are the rules for these cases.

$$\frac{H \otimes E \ \twoheadrightarrow \ PO}{H \otimes x := E \ \twoheadrightarrow \ PO} \ x \text{ is a simple variable}$$

$$\frac{H \otimes E \ \twoheadrightarrow \ PO}{H \otimes r(x) := E \ \twoheadrightarrow \ PO} \qquad \frac{H \otimes r \ \twoheadrightarrow \ PO}{H \otimes r(x) := E \ \twoheadrightarrow \ PO}$$

$$\frac{H \otimes E \ \twoheadrightarrow \ PO}{H \otimes r'f(x) := E \ \twoheadrightarrow \ PO} \qquad \frac{H \otimes r \ \twoheadrightarrow \ PO}{H \otimes r'f(x) := E \ \twoheadrightarrow \ PO}$$

The above means that for the assignment $r(x)(y) := E$ we require E to be well-defined and we require that $x \in dom(r)$ and that r is a partial function.

For the WHILE substitution we need to know the variables x that are modified in the loop. We adapted the proof rule from [8] to our formalism (i.e., the invariant I is pushed onto the stack for P, V and B and P is pushed onto the stack for discharging B):

A tricky aspect is the sequential composition. [8] contains a few specific rules and was trying to avoid having to apply the weakest-precondition computations in full.[4] We have adapted a few of the rules from [8], the most used one being $WD(x := E \ ; \ Q \ ) = WD(E) \wedge WD([x := E]Q))$ where $[x := E](P) = P[E/x]$. Note that [8] also contains a rule for parallel composition followed by sequential composition which is wrong. There is also a special rule for a WHILE loop in the LHS of a sequential composition which we have not implemented. The rules implemented thus far proved sufficient for many applications.

[4] Atelier-B now uses full WP calculus (private communication from Lilian Burdy).

4 Fast Integrated POG and Prover

One can notice that in the above proof rules for $H \otimes F \twoheadrightarrow P$ the hypotheses H are passed through to subarguments of F and sometimes a new hypothesis is added. This means that a lot of proof obligations will share common hypotheses. In RODIN each well-definedness PO is discharged on its own and a new prover instance is launched for every PO. This is not very efficient, especially when the number of hypotheses becomes large (cf. Section 5). In ATELIER-B the hypotheses can be numbered and shared amongst proof obligations, which is useful when discharging multiple proof obligations in one tool run. However, for every PO the hypotheses must still be assembled.

One key idea of this paper is to discharge the POs in the same order they are generated by our POG rules and to treat the hypotheses as a *stack*. E.g., when one enters the right-hand side of a conjunction we *push* the left-hand side as a hypothesis onto the stack, when leaving the conjunction we *pop* this hypothesis again. The pushing of a new hypothesis can also conduct a few proof-related tasks, like normalization and indexing.

Another insight of this paper is that in the Prolog programming language the popping can be done very efficiently upon backtracking: the Prolog virtual machine is optimised for these kinds of operations and does them in a memory and time efficient way.

Below we show our implementation of the above POG rule for the conjunction. The Prolog predicate `compute_wd` encodes our relation $Hypotheses \otimes A \wedge B \twoheadrightarrow PO$ with some additional arguments (for source code locations, typing and options). You can see that there is a call to push A onto the hypothesis stack, but no pop operation, which is performed upon backtracking.

```
1  compute_wd_aux(conjunct(A,B),_,_,_,_,Hypotheses,Options,PO) :- !,
2     (compute_wd(A,Hypotheses,Options,PO)
3      ;
4      push_hyp(Hypotheses,A,Options,NewHyp),
5      compute_wd(B,NewHyp,Options,PO)).
```

Listing 1.2. Prolog clause for processing the conjunction

The `push_hyp` predicate will also filter useless hypotheses and normalise the useful ones. It also performs indexing to ensure that subsequent proving steps can be performed efficiently. For commutative operators this may mean to store a hypothesis twice. Our technique will ensure that this overhead is only incurred once for all proof obligations having that particular hypothesis on the stack.

For quantifiers we need to provide the renaming mechanism $\rho_V(H)$. To avoid traversing all hypotheses upon every clash, our implementation of $\rho_V(H)$ actually stores a list of variable clashes and renamings. The renamings are not applied to the existing hypotheses, only to new hypotheses and the final goal of the PO.

In essence, the main ideas for obtaining a fast and effective proof obligation generator and prover are:

– use Prolog pattern matching on the syntax tree to implement the POG generation,

– combine proof-obligation generation and proving in a single traversal, discharging POs in the same order they are generated,
– organize the hypothesis as a stack, use Prolog backtracking for popping from the stack,
– pre-compile the hypotheses to enable logarithmic lookup of hypotheses,
– use a rule-based prover in Prolog which only uses such logarithmic lookups in hypotheses, performs rewrite steps using Prolog unification and limiting non-determinism as much as possible.

Normalization and Lookup of Hypotheses Normalization is employed by many provers, e.g., it is used in Atelier-B to minimize the number of proof rules that have to be implemented (see Chapter 3 of Interactive Prover manual of [12]).

Our rules are different from the ones in [12], as we are also concerned with ensuring logarithmic lookup of hypotheses. Our hypotheses are stored as an AVL tree using the normalised Prolog term as key. AVL trees are self-balancing binary search trees with logarithmic lookup, insertion and deletion (see, e.g., Section 6.2.3 of [20]). We have used the AVL library of SICStus Prolog 4, and implemented a new predicate to enable logarithmic lookup if the first argument of the top-level operator is known, but the second argument may be unknown (making use of lexicographic ordering of Prolog terms).

Predicate	Normalization	Additional Hypotheses	Conditions
$x > n$	$x \geq n+1$		if n is a number
$n > x$	$x \leq n-1$		if n is a number
$x > y$	$x \geq y \wedge x \neq y$	$y \leq x$	otherwise
$x < n$	$x \leq n-1$		if n is a number
$n < x$	$x \geq n+1$		if n is a number
$x < y$	$x \leq y \wedge x \neq y$	$y \geq x$	otherwise
$A \subset B$	$A \subseteq B \wedge A \neq B$	$B \supseteq A$	

Table 1. A few normalization rules and the generation of additional hypotheses

All hypothesis lookups in our prover are logarithmic (in the number of hypotheses); no lookup requires a linear traversal. Some hypotheses are stored multiple times to enable this logarithmic lookup based on first argument: the predicate $a = b$ is also stored in the form $b = a$ if the term b is susceptible to be looked up. The predicate $a < b$ may result in three hypotheses being added: $a \leq b, b \leq a, a \neq b$. $a \neq b$ is only stored once, as upon lookup time both arguments are known. The Table 1 shows some of our normalization rules.

Table 2 shows the lookups that are made by our prover in the hypothesis stack. As mentioned, the first argument A is always known. All other hypotheses not occuring in Table 2 are not pushed onto the stack (in proving mode), as they would never be used anyway.

Predicates Supported by the Prover The rule-based prover contains various Prolog predicates for proving a few core B predicates, namely those listed in Table 3.

Patterns for Lookups	
$\mathit{finite}(A)$	$A \in B$
$A = B$	$A \neq B$
$A \leq B$	$A \geq B$
$A \subseteq B$	$A \supseteq B$

Table 2. Lookups made in the Hypothesis Stack (A is always known, B known for $\neq$)

The following Prolog clauses contain a small part of the `check_finite` predicate responsible for proving the B *finite* predicate. The first argument is the B expression which is the argument to the *finite* operator, the second argument is the hypothesis stack while the third argument is a proof tree constructed by the prover (for subsequent inspection or validation).

```
1  check_finite(bool_set,_,bool_set) :- !.
2  check_finite(empty_set,_,empty_set) :- !.
3  check_finite(intersection(A,B),Hyps,intersection(D,PT)) :- !,
4      (  D=left, check_finite(A,Hyps,PT) -> true
5       ; D=right,check_finite(B,Hyps,PT)).
6  check_finite(set_subtraction(A,_),Hyps,set_subtraction(PT)) :- !,
7      check_finite(A,Hyps,PT).
8  check_finite(range(A),Hyp,ran(PT)) :- !, check_finite(A,Hyp,PT).
```

Listing 1.3. Some Prolog clauses for checking the finite B predicate

The clauses encode the axioms $\mathit{finite}(BOOL)$ and $\mathit{finite}(\varnothing)$ as well as the proof rules that $\mathit{finite}(A \cap B)$ holds if either $\mathit{finite}(A)$ or $\mathit{finite}(B)$ and that $\mathit{finite}(A \setminus B)$ or $\mathit{finite}(ran(A))$ hold if $\mathit{finite}(A)$.

The proof rules and derived Prolog clauses are written such that matching of B predicates (like `intersection` or `set_subtraction` above) always occur at the top-level of the formulas. This ensures that we can use efficient and simple Prolog unification for the proof rules and that Prolog's argument indexing often results in constant time lookup of possible matching proof rules.

B Predicates handled by the Prover	
$A \subseteq B$	$A \in B$
$A \leq B$	$A \neq B$
$A \in T \mathrel{+\!\!\!\rightarrow} T'$	(functional)
$A^{-1} \in T \mathrel{+\!\!\!\rightarrow} T'$	(injective)
$A \in \mathit{seq}(T)$	(is sequence)
$\mathit{finite}(T)$	
$dom(A) = D$	$dom(A) \subseteq D'$
$ran(A) = R$	$ran(A) \subseteq R'$

Table 3. Prolog prover predicates, where T and T' are maximal type sets

Ensuring Termination To avoid useless rewrites, our prover contains local loop checks within the predicates of Table 3. Some rewrites are also guarded by an occurs check, to prevent rewriting x to something like $rev(rev(x))$. Finally, a depth bound limits the number of equality and subset rewrites applicable within a particular proof or sub-proof. Currently the bound is set to allow 5 rewrites; increasing this bound only minimally increases the number of POs discharged in Section 5.3.

Implementation within ProB The prover has been integrated into the ProB validation tool. On the one hand, this has eased the implementation, as part of ProB's infrastructure (parser, typechecker, static rewriter) could be re-used. For the rules concerning substitutions (Sect. 3), we also reused the code for computing written variables. On the other hand, we also plan to use the output of the prover for ProB's constraint solver; see Section 6. Finally, this also enabled to make the prover available within RODIN, as part of the ProB-Disprover plugin. This is particularly useful for discharging POs which pose problems to other provers (e.g., for *min*, *max* and *card*). Existing integrations with editors, such as Atom and VSCode could also be easily extended to highlight potential WD issues.

5 Benchmarks

Below we provide a variety of benchmarks. Section 5.1 contains artificial benchmarks to measure scalability compared with Atelier-B and RODIN. In Sect. 5.2 we examine a few specific POs extracted as regression tests, while in Sect. 5.3 we perform a more exhaustive evaluation on over 6000 models from the ProB examples repository. All experiments were run on a MacBook Pro with 2.8 GHz i7 processor, 16GB of RAM and running macOS 10.14.6. For the experiments we have used version `1.10.0-beta2` of the command-line version `probcli` with the flags `-wd-check -silent`, which runs our PO generator and prover on the provided model and prints a summary information (the `-silent` flag prevents the output of source locations for the undischarged POs) available at:

`https://www3.hhu.de/stups/downloads/prob/tcltk/releases/1.10.0-beta2/`

5.1 Artificial Benchmarks

We next present the following artificial benchmark model template, where Nr is parameter which we have instantiated to various values between 100 and 8000 below. Atelier-B generates $3Nr$ proof obligations while our implementation generates $6Nr$, as we check separately for every function application that ff is a function and that the argument is in the domain of ff.

```
1 MACHINE FunNrWD
2 CONSTANTS ff
3 PROPERTIES
4     /* axm0 */ ff : 1 .. Nr --> 1 .. 90
5   & /* axm1 */ ff(1) < 100
```

```
6    & /* axm2 */ ff(2) < 100
7    ...
8    & /* axmNr */ ff(Nr) < 100
9    & /* axm_nest_1 */ ff(ff(1)) < 100
10   ...
11   & /* axm_nest_Nr */ ff(ff(Nr)) < 100
12 INITIALISATION skip
13 END
```

Listing 1.4. Artificial Benchmark Template

AtelierB We have loaded the above model into the 64-bit version 4.6.0-rc4 of Atelier-B for macOS. The timings for Atelier-B were obtained using a stop-watch, after the models had been loaded and typechecked. The timings of our implementation were taken within ProB, using walltime for the total time needed to generate and discharge the POs. The time needed to parse and load and typecheck the machine was *not* measured for either tool.

	Atelier-B			ProB WD	
Nr	POG	Proof F0	Discharged	POG + Proof	Discharged
100	4 sec	13 sec	100 %	0.035 sec	100 %
200	40 sec	62 sec	100 %	0.041 sec	100 %
500	error	-	0 %	0.058 sec	100 %
1000	error	-	0 %	0.083 sec	100 %
2000	error	-	0 %	0.139 sec	100 %
4000	error	-	0 %	0.252 sec	100 %
8000	error	-	0 %	0.478 sec	100 %

Table 4. Artificial WD Benchmark FunNrWD (classical B)

For Nr=100 and Nr=200 our implementation is a few orders of magnitude faster. Atelier-B ran into a "memory overflow (max expansion reached)" in default settings for Nr=200. After increasing the "m" parameter by a factor of 100 we managed to generate the proof obligations for Nr=200. But for Nr=500 we were not successful (4.25 GB memory were used; error generated after about 90 seconds, we tried to increase the memory allowance as much as the UI would let us).

Note that the first run of the WD prover within ProB is always a bit slower (probably due to JIT startup time). Indeed, the second run of the WD prover is considerably faster (18 ms for Nr=200). This is beneficial when checking multiple models (such as in Sect. 5.3) or when ProB is left open while working on a model.

Rodin We encoded the above B machines in Event-B and used rodin version 3.4. We used a stop watch to measure the POG (building) time and the auto prover time ("Retry Auto Provers" command).

As Table 5 shows, rodin is initially slower than Atelier-B, but is able to process larger models. However, the proving time is quite considerable as every

Nr	RODIN POG	Auto Prover	Discharged	PROB WD POG + Proof	Discharged
100	3 secs	2 min 28 sec	49.5 %	0.036 sec	100 %
200	8.5 sec	6 min 05 sec	24.5 %	0.044 sec	100 %
500	47 sec	17 min 10 sec	9.7 %	0.063 sec	100 %
1000	1 min 25 sec	+/- 45 min	5.2 %	0.121 sec	100 %

Table 5. Artificial WD Benchmark FunNrWD (Event-B)

proof obligation is sent to a new instance of the provers. For Nr=100 it is about an order of magnitude slower than ATELIER-B and three orders of magnitude slower than our technique, and discharges only half of the POs. For Nr=1000 it is about 23,000 times slower than our technique. We did try to prove some of the POs in RODIN by hand. For `axm110` PP needs to be interrupted but ML and Z3 can be used to prove it. For `axm_nest_999` the ML prover fails, Z3, veriT, and CVC4 run into timeouts and PP needs to be interrupted.

5.2 A few selected POs

The regression test 2018 of PROB contains 189 well-defined formulas which were collected from existing models, leading to 413 POs. Of course this test is biased, as it contains the regression tests for our prover. However, these regression tests were usually extracted from existing models, and were written to cover a large class of typical *WD* situations arising in practice. Here we wish to show that our prover does treat some naturally occurring WD POs better than the standard provers mono-lemma ML and the predicate prover PP in their default settings.

For the experiments we used PROB's atelierb_provers_interface module which calls KRT with the options `-a m1500000 -p rmcomm`. The results are summarised in Table 6. Our prover discharges all 413 POs in 47 ms, with a maximum walltime of 5 ms per PO. The maximum walltime of ML was about 18 seconds for the PO

```
f : 1,3,5,7,9 --> 1,3,5,7,9 => 5 : dom(f).
```

If we deduct the minimum walltime of 0.26 sec of ML (which is probably due to the overhead of starting a new ML for each PO) we obtain a runtime of around 45 seconds for all POs.

For PP some POs seemed to run into an infinite loop and we interrupted the prover on 18 occasions, e.g., for the PO

```
10 / f(a) = 10 / a & a : NATURAL1 & b : NATURAL &
    f : NATURAL1 --> NATURAL1 => b + 1 : dom(f).
```

The longest successful run of PP was around 49 sec for the PO

```
x' : 2 .. 8 => %x.(x : 2 .. 8|10 / x) : (INTEGER) +-> (INTEGER).
```

PP was not able to prove e.g. the POs

```
s : perm(1 .. 10) => s : (INTEGER) +-> (INTEGER)
x : POW(1 .. 2) => finite(x).
```

We have also tried to use Z3 4.8.8 via the translation [23] available in PROB. This translation does not support sequences and all B operators, and also un-

fortunately terminated after 23 POs (with an uncaught "datatype is not well-founded" exception). We tried to selectively skip over some POs, without success. In future it would be good to try the translation to SMTLib from [14] (but which would not support sequences either).

Prover	Proved	Unproved	Ctrl-C	Min.	Max.	Total	w/o Min.
ProB-WD	413	0	0	0.000 sec	0.006 sec	0.047 sec	0.047 sec
ML	190	223	0	0.260 sec	18.725 sec	152.633 sec	45.253 sec
PP	230	165		0.092 sec	49.144 sec	222.940 sec	186.600 sec
			18			1017.406 sec	-
Z3	9	14	0	0.007 sec	2.520 sec	crash	-

Table 6. POs Extracted from ProB Regression Tests

5.3 Benchmarks from ProB Examples

The ProB source code is accompanied by a large selection of models, which are used for regression tests. In a recent effort, the public part of these models have been made available for reproducible benchmark efforts and other research uses at: `https://github.com/hhu-stups/specifications`

For this article we have extracted the parseable and type-correct B and Event-B specifications to evaluate our tool. The scripts to run our tool are available in the folder `benchmarks/well-definedness` of the above repository. The summary is in Table 7; the detailed results for the 2579 B and 760 Event-B models can be found in the above repository. We also ran the experiments on the private B machines (.mch files) and Event-B files in the ProB examples. These are summarised in Table 8.

Formalism	Files	Total POs	Discharged	Perc.	Runtime	Avg. per File
B	2579	106784	90357	84.62 %	4.46 secs	1.7 ms
Event-B	760	42824	38847	90.71 %	1.10 secs	1.4 ms

Table 7. ProB WD on Public Benchmarks using ProB Examples

Formalism	Files	Total POs	Discharged	Perc.	Runtime	Avg. per File
B	3370	354769	288968	81.45 %	38.67 secs	11.5 ms
Event-B	145	32647	27202	83.32 %	1.01 secs	7.0 ms

Table 8. ProB WD on Private Benchmarks from ProB Examples

The performance exceeded our initial hopes and one could run this analysis as part of the ProB loading process without users noticing a delay: 82.9 % of the more than half a million POs from 6000 models were discharged in less than 50 seconds. For the public Event-B models the tool managed to discharge over 30000 POs per second. The maximum runtime was 0.310 secs for one file (a formal model of the Z80 processor with many equalities in the invariants).

The precision of the analysis is also very satisfactory. For many models 100 % of the POs are discharged, e.g. all 114 for a Paxos model by Abrial or all 118 for MovingParticles, an encoding [25] of an ASM machine in Event-B (whose WD POs which are tedious to discharge in RODIN). The analysis has also uncovered a considerable number of real well-definedness issues in existing models. In terms of the true POs, the discharge percentage of our tool should be noticeably higher. Indeed, we checked the unproven POs of the public Event-B models with ML; it managed to discharge only 7 % of them (i.e., an additional 0.55% overall).

6 Discussions and Outlook

Explanations for Performance What can explain the big performance difference of our tool compared to Atelier-B and RODIN? Some reasons have already been mentioned earlier:

- the combined PO generation and proving in one go definitely reduces some overhead,
- no overhead of calling an external prover (relevant compared to RODIN),
- no need to transmit or load hypotheses for a PO, all hypotheses are pre-compiled on the stack,
- efficient popping of hypotheses using Prolog's backtracking,
- only logarithmic hypotheses lookups are performed in the prover and useless hypotheses are not stored in stack.

Part of the performance also comes from the special nature of WD POs. Indeed, one could try to implement our proof rules as custom proof rules for ML, which would probably boost its benchmark results. Indeed, Atelier-B uses the *theory language* to express proof rules, which can be viewed as domain specific logic "programming language" tailored to B and proof. While Atelier-B comes with a custom developed compiler — the *Logic Solver* — it seems like it cannot compete with state-of-the-art Prolog compilers. A small experiment consisted in summing the numbers from 1..500000 in the theory language (written by Thierry Lecomte) and in Prolog. Using `krt` in Atelier-B 4.3.1 this task runs in over 6 seconds, while SICStus Prolog perform the same task in 0.001 seconds. Thus some of the performance is certainly due to implementing our proof rules in Prolog. The drawback of Prolog is that it has more limited matching (i.e., unification), namely only at the top-level of a Prolog term. This meant that we had to repeat some rewriting rules multiple times (for each predicate in Table 3).

RODIN uses external provers such as ML, PP or Z3 [14], and also the TOM rewriting library[6]. RODIN's internal sequent prover, however, seems to have been developed using hand-written matching, which is probably much less ef-

ficient than in Prolog or a dedicated term rewriting system. The hand-written solution is also very verbose: the equivalent of the last line 8 of our Prolog prover in Listing 1.3 is a file `FiniteRan.java`[5] with 82 lines of code (9 lines are copyright notice). The Prolog code is also very flexible (e.g., it can be used for finding proofs but also for re-playing or checking proofs if the proof tree argument is provided).

We are not the first to use Prolog to implement a prover [7, 15, 29]. An open question is whether using term rewriting [19] would be an even better approach. As mentioned above, Prolog unification is more limited, but very efficient.[6] Within a term rewriting system we could simplify our prover code, possibly use AC unification and avoid duplication of rewrite rules. An interesting topic for future research would be to port our prover to such a term rewriting system (like Maude).

WD and Constraint Solving Well-definedness is important for constraint solving in ProB's kernel. Indeed, constraint propagation can be much more effective if one assumes well-definedness. Take for example the predicate $x \in 1..10 \land y \geq 0 \land z = x \div y \land z > 10$. If we assume well-definedness of $x \div y$, we can infer that $z \in 1..10$ and hence realise that the constraint has no solution. If on the other hand, we wish to detect WD errors, the constraint solver has to delay until it knows whether y is 0 or not. In case $y = 0$ one can produce an error message, and if $y > 0$ the constraint is unsatisfiable. The detection of well-definedness errors is made more complicated by the fact that a solver does not necessarily treat predicates from left-to-right.

This is the reason many constraint solvers ignore well-definedness errors (see also [16]). E.g., in SMTLib or the finite domain constraint library CLP(FD) of SICStus Prolog, a division by zero simply results in an unsatisfiable predicate (and not in an error).This is not the approach used by ProB: it tries to detect well-definedness errors, but unfortunately is not guaranteed to detect all WD errors, because some of the checks would be prohibitively expensive at solving time. Particularly, within nested set comprehensions such well-definedness issues can cause unexpected results. The techniques of this article will allow us to implement a much better approach:

- if all WD POs are discharged, we know that no WD errors can arise. We can then perform stronger constraint propagations in the ProB kernel.
- if not all WD POs are discharged, we can turn full WD checking at runtime for those places where the POs have not been discharged.

Outlook Concerning our proof obligation generator we plan to extend as needed to cover substitutions in classical B more precisely. Full coverage will, however,

[5] See FiniteRan.java in `org.eventb.internal.core.seqprover.eventbExtensions` at `https://sourceforge.net/p/rodin-b-sharp/rodincore`

[6] The missing occurs check in Prolog is not an issue, because we use the ground representation for the B formulas, and hence any variable in a proof rule is always instantiated to a ground term.

also require a full implementation of the weakest-precondition computation. The generator is currently tailored for use within ProB; for usage outside of ProB, we will need to allow to preserve the interleaved exact order of theorems and invariants for RODIN models or the order of included invariants in B. It would also be benefical to extract the proof status information from RODIN and Atelier-B; this will further improve performance and precision and give users a fallback solution in case our prover is not powerful enough.

The prover itself can be further improved, in particular cycle detection can be made more efficient. We also plan to provide a "strength" option to enable more non-deterministic proof rules, at the cost of runtime. The quantifier instantiations and treatment of implications can also be extended.

It would be useful to visualize the proof tree constructed by our tool, and display the useful hypotheses for a particular PO. The proof tree could also be checked by a second tool, for validation purposes. Similar to what was done for ML, we could also attempt to prove all our rewrite rules using another prover.

About 10 years ago Abrial proposed [4] an outline for a new improved prover P3 for the B method. The results of this paper could be an encouragement to try and develop this successor to ML and PP using Prolog, possibly incorporating ideas from SMT solvers into the Prolog prover as shown in [18, 28]. Maybe our approach could also be used to provide an easily extensible, yet efficient prover, for RODIN's theory plugin [10].

Summary In summary, we have developed a new fast and effective integrated proof obligation generator and prover for well-definedness. It can deal with B sequences and with various extensions of the B language. It has been integrated into the ProB validation tool, and is able to analyse formal models effectively and quickly, with average runtimes below 0.01 seconds for over 6000 benchmark models. Our technique is orders of magnitude faster than existing implementations in Atelier-B and RODIN. The output of our tool can be inspected either within ProB or within the Atom and VSCode editors, which proved to be useful to detect a considerable number of errors in existing models. The prover is also available in RODIN via ProB's Disprover [22] plugin. In future, the output of the prover will be used by ProB's constraint solver to improve performance and to better detect well-definedness errors at solving time.

Acknowledgements A big thanks go to Philipp Körner for scripts for extracting benchmark specification list, Thierry Lecomte for writing the sum Logic Solver example, Sebastian Stock for the VSCode integration and David Geleßus for the ProB2-UI integration. I wish to thank Jean-Raymond Abrial, Lilian Burdy, Michael Butler, Stefan Hallerstede and Laurent Voisin for useful feedback, Atelier-B and RODIN implementation details and pointers to related research. In particular Laurent Voisin provided many useful hints and corrections.

References

1. J.-R. Abrial. *The B-Book*. 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 Z. Liu and J. He, editors, *Proceedings ICFEM'06*, LNCS 4260, pages 588–605. Springer-Verlag, 2006.
4. J.-R. Abrial, D. Cansell, and C. Métayer. Specification of the automatic prover P3. In *Proceedings of the 10th Workshop on Automated Verification of Critical Systems (AVoCS'10) and the Rodin User and Developer Workshop*, September 2010. Slides available at https://wiki.event-b.org/images/Rodin2010-sld-abrial.pdf.
5. J.-R. Abrial and L. Mussat. On using conditional definitions in formal theories. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *Proceedings ZB'2002*, LNCS 2272, pages 242–269. Springer-Verlag, 2002.
6. E. Balland, P. Brauner, R. Kopetz, P. Moreau, and A. Reilles. Tom: Piggybacking rewriting on java. In F. Baader, editor, *Proceedings RTA 2007*, volume 4533 of *Lecture Notes in Computer Science*, pages 36–47. Springer, 2007.
7. B. Beckert and J. Posegga. leantap: Lean tableau-based deduction. *J. Autom. Reasoning*, 15(3):339–358, 1995.
8. P. Behm, L. Burdy, and J. Meynadier. Well defined B. In D. Bert, editor, *Proceedings B'98*, LNCS 1393, pages 29–45. Springer, 1998.
9. S. Berezin, C. Barrett, I. Shikanian, M. Chechik, A. Gurfinkel, and D. L. Dill. A practical approach to partial functions in cvc lite. *Electronic Notes in Theoretical Computer Science*, 125(3):13 – 23, 2005.
10. M. J. Butler and I. Maamria. Practical theory extension in event-b. In Z. Liu, J. Woodcock, and H. Zhu, editors, *Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, LNCS 8051, pages 67–81. Springer, 2013.
11. M. J. Butler, K. Schewe, A. Mashkoor, and M. Biró, editors. *Proceedigns ABZ 2016*, LNCS 9675. Springer, 2016.
12. ClearSy. *Atelier B, User and Reference Manuals*. Aix-en-Provence, France, 2009. Available at `http://www.atelierb.eu/`.
13. Á. Darvas, F. Mehta, and A. Rudich. Efficient well-definedness checking. In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings IJCAR 2008*, LNCS 5195, pages 100–115. Springer, 2008.
14. D. Déharbe, P. Fontaine, Y. Guyot, and L. Voisin. Integrating SMT solvers in rodin. *Sci. Comput. Program.*, 94:130–143, 2014.
15. M. Fitting. leantap revisited. *J. Log. Comput.*, 8(1):33–47, 1998.
16. A. M. Frisch and P. J. Stuckey. The proper treatment of undefinedness in constraint languages. In I. P. Gent, editor, *Proceedings CP 2009*, LNCS 5732, pages 367–382. Springer, 2009.
17. D. Hansen, D. Schneider, and M. Leuschel. Using B and prob for data validation projects. In Butler et al. [11], pages 167–182.
18. J. M. Howe and A. King. A pearl on SAT and SMT solving in Prolog. *Theor. Comput. Sci.*, 435:43–55, 2012.
19. J. Hsiang, H. Kirchner, P. Lescanne, and M. Rusinowitch. The term rewriting approach to automated theorem proving. *J. Log. Program.*, 14(1&2):71–99, 1992.
20. D. Knuth. *The Art of Computer Programming, Volume 3*. Addison-Wesley, 1983.

21. N. Kosmatov, C. Marché, Y. Moy, and J. Signoles. Static versus dynamic verification in why3, frama-c and SPARK 2014. In T. Margaria and B. Steffen, editors, *Proceedings ISoLA 2016*, LNCS 9952, pages 461–478, 2016.
22. S. Krings, J. Bendisposto, and M. Leuschel. From failure to proof: The ProB disprover for B and Event-B. In R. Calinescu and B. Rumpe, editors, *Proceedings SEFM 2015*, LNCS 9276, pages 199–214. Springer, 2015.
23. S. Krings and M. Leuschel. SMT solvers for validation of B and Event-B models. In E. Ábrahám and M. Huisman, editors, *Proceedings IFM 2016*, LNCS 9681, pages 361–375. Springer, 2016.
24. T. Lecomte, L. Burdy, and M. Leuschel. Formally checking large data sets in the railways. *CoRR*, abs/1210.6815, 2012. Proceedings of DS-Event-B 2012, Kyoto.
25. M. Leuschel and E. Börger. A compact encoding of sequential asms in event-b. In Butler et al. [11], pages 119–134.
26. F. Mehta. A practical approach to partiality - A proof based approach. In S. Liu, T. S. E. Maibaum, and K. Araki, editors, *Proceedings ICFEM 2008*, LNCS 5256, pages 238–257. Springer, 2008.
27. C. Métayer and L. Voisin. The Event-B mathematical language. Available at `http://wiki.event-b.org/index.php/Event-B_Mathematical_Language`, 2009.
28. E. Robbins, J. M. Howe, and A. King. Theory propagation and reification. *Sci. Comput. Program.*, 111:3–22, 2015.
29. R. F. Stärk. The theoretical foundations of lptp (a logic program theorem prover). *The Journal of Logic Programming*, 36(3):241 – 269, 1998.

A Screenshots and Links

Instructions for running well-definedness checking in ProB can be found at: `https://www3.hhu.de/stups/prob/index.php/Well-Definedness_Checking`.
The source code of the well-definedness proof obligation generator and prover is available in the files `well_def_analyser.pl`, `well_def_hyps.pl`, `well_def_prover.pl` as part of ProB's source code:
`https://www3.hhu.de/stups/prob/index.php/Download#Sourcecode`.

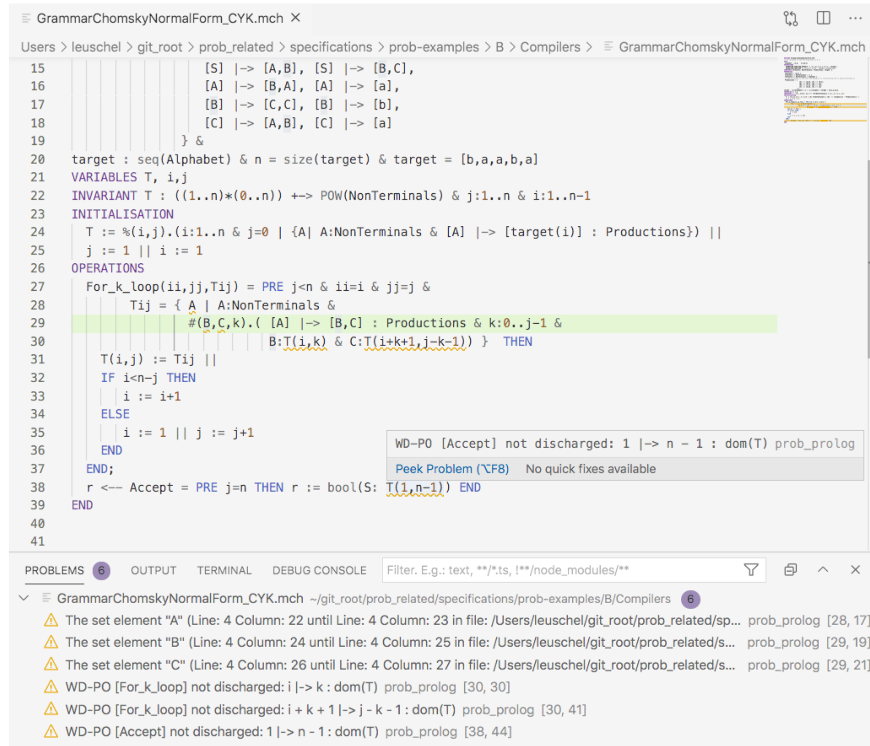

Fig. 1. B/ProB Plugin for VisualCode, highlighting undischarged WD POs. There are missing invariants about T which prevent discharging the POs.

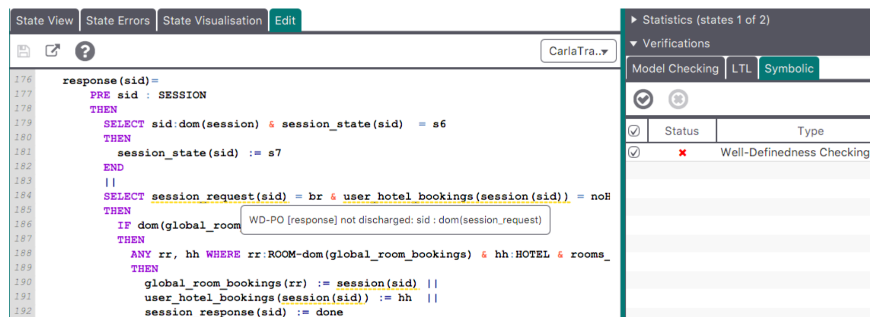

Fig. 2. Display of undischarged WD POs in the ProB2-UI. The author of the model was unaware that the guard of the first SELECT cannot be used to prove the POs in question.

```
ProB 1.10.0-nightly: [NFA_to_DFA_Unicode.mch]
21  Σ = {0,1} ∧
22  S ⊆ Z ∧
23  F ⊆ Z ∧
24  δ ∈ (Z×Σ) → ℙ(Z) ∧
25
26  /* rekursive Definition der erweiterten Übergangsfunktion */
27  δs ∈ (ℙ(Z)×seq(Σ)) → ℙ(Z) ∧
28  δs = λ(Z',s).(Z'⊆Z ∧ s∈seq(Σ) |
29            IF s=[] THEN Z'
30            ELSE  ∪(z).(z∈Z'|δs(δ(z,first(s)),tail(s))) END)
31  ∧
32
33  /* die vom Automaten generierte Sprache */
34  L = {ω|ω∈seq(Σ) ∧ δs(S,ω) ∩ F ≠ ∅}
35
36  ∧
37
38  /* Nun der A
```

State Properties
MAXINT = 214748364
MININT = -214748364
card(Z) = 4

WD PO List

WD Proof Obligations List (16/16 DISCHARGED)

PO Label	Discharged	Proof Obligation Goal	Source
AXM	yes	s : seq1((INTEGER))	at line 30:35 - 30:43
AXM	yes	z |-> first(s) : dom(δ)	at line 30:31 - 30:44
AXM	yes	δ : (Z*INTEGER) +-> (POW(Z))	at line 30:31 - 30:44
AXM	yes	s : seq1((INTEGER))	at line 30:45 - 30:52
AXM	yes	δ(z |-> first(s)) |-> tail(s) : dom(δs)	at line 30:28 - 30:53
AXM	yes	δs : (POW(Z)*seq(INTEGER)) +-> (POW(Z))	at line 30:28 - 30:53
AXM	yes	S |-> ω : dom(δs)	at line 34:19 - 34:26
AXM	yes	δs : (POW(Z)*seq(INTEGER)) +-> (POW(Z))	at line 34:19 - 34:26
THM-AXM	yes	{z_0} |-> [0,1,1,1] : dom(δs)	at line 45:2 - 45:20
THM-AXM	yes	δs : (POW(Z)*seq(INTEGER)) +-> (POW(Z))	at line 45:2 - 45:20
THM-AXM	yes	{z_0} |-> [] : dom(δs)	at line 46:2 - 46:13
THM-AXM	yes	δs : (POW(Z)*seq(INTEGER)) +-> (POW(Z))	at line 46:2 - 46:13
THM-AXM	yes	{z_0} |-> [1] : dom(δs)	at line 47:2 - 47:14
THM-AXM	yes	δs : (POW(Z)*seq(INTEGER)) +-> (POW(Z))	at line 47:2 - 47:14
Go	yes	cur |-> [input] : dom(δs)	at line 53:37 - 53:52
Go	yes	δs : (POW(Z)*seq(INTEGER)) +-> (POW(Z))	at line 53:37 - 53:52

Fig. 3. Display of WD POs in the ProB Tcl/Tk UI. Here all POs are discharged for a machine involving a recursive function and sequence operators.